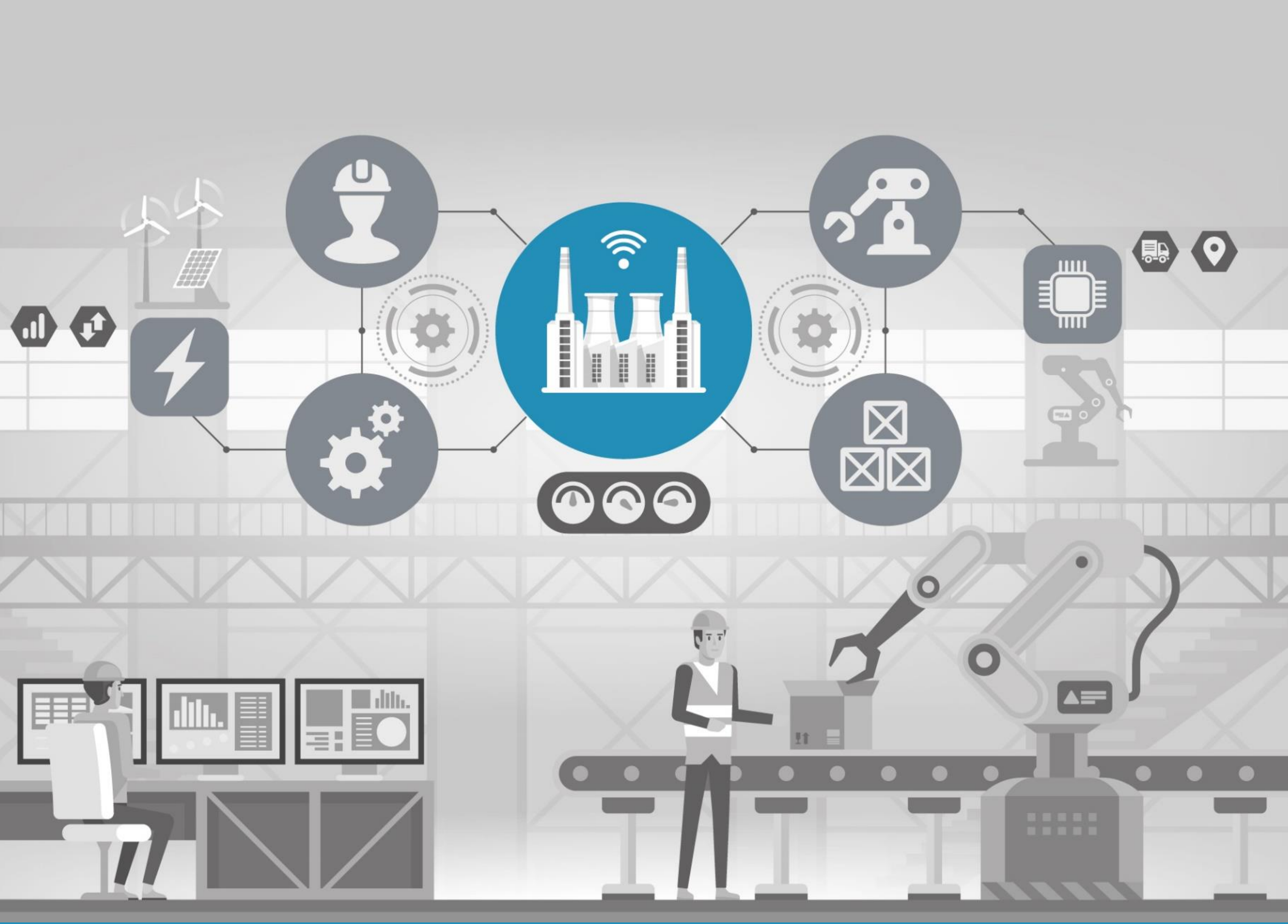


IIP-Ecosphere Whitepaper

IIP-ECOSPHERE PLATFORM HANDBOOK

Version 0.20

Holger Eichelberger, Amir Shayan Ahmadian,
Andreas Dewes, Marco Ehl, Monika Staciwa,
Miguel Gómez Casado

White Paper IIP-2021/005

IIP-Ecosphere
Next Level Ecosphere for
Intelligent Industrial Production

Disclaimer

Publication: August, 2021 on https://www.iip-ecosphere.eu/

DOI: 10.5281/zenodo.5168946

Gefördert durch:

Bundesministerium
für Wirtschaft
und Energie

Executive Summary

The IIP-Ecosphere platform is a central asset developed by the IIP-Ecosphere project. The core aim of the platform is to research and demonstrate novel platform concepts for Industry 4.0, e.g., asset administration shells as interfaces for software components and resources, unified edge deployment, an AI toolkit or seamless configuration of a platform from network settings via services up to applications running on the platform. This platform handbook provides insights into the rationales, ideas and concepts that make up the design and the realization of the IIP-Ecosphere platform, ranging from an overall layered architecture over a detailed discussion of the design and realization state of each layer up to cross-cutting mechanisms such as the configuration model or the related code/artifact generation.

This platform handbook addresses the technical side of the platform work in IIP-Ecosphere and builds on the intensive prior work on requirements (usage view and functional/quality requirements of the platform). This handbook shall provide means for deeper technical discussions with partners, stakeholders and interested parties, but also allow for a technical understanding to contribute to the platform, e.g., in terms of protocols, platform connectors, services or demonstration applications.

Acknowledgements: We are grateful to Dr. Christian Sauer and Ahmad Alomosh from the Software Systems Engineering Group of the University of Hildesheim for cross-reading this document and providing valuable feedback and ideas for improvement.

Contents

1 Introduction

1.1 Motivation and Goals

The digitalization of the industry increases the effectiveness of technical systems and related processes, but also affects the complexity of the realizing (software) systems. Currently, several approaches are developed in the fields of Internet-of-Things (IoT), Industrial Internet-of-Things (IIoT) or „Industrie 4.0“ (I4.0)[1]. To support the industrial transformation towards IoT, IIoT and I4.0, several software platforms were developed that provide different capabilities.

The vision of the BMWi-funded[2] project IIP-Ecosphere is to enable innovations in the area of industrial production based on connected, intelligent and autonomous systems in order to increase productivity, flexibility, robustness and efficiency of IIoT and I4.0. IIP-Ecosphere aims at creating a novel ecosystem for the "next level" of intelligent industrial production, not only for software-based systems, but also for the people involved in this kind of systems, e.g., automation engineers, software developers, AI experts, startups, venture capitalists, etc. On the software side, one core activity in IIP-Ecosphere is to research and to realize a virtual platform that connects factory installations across companies in a vendor-independent manner. In particular, the platform shall provide easy-to-use access to Artificial Intelligence (AI) in secure and flexible manner.

Towards the design of such a platform, we analyzed in [27] 21 IIoT platforms with specific relevance to IIP-Ecosphere and described in [8, 30] the requirements for the IIP-Ecosphere platform from two different perspectives, namely the usage view and the functional/quality requirements view. The next step is to turn the requirements into an architecture and to implement the platform. The resulting platform shall be open, extensible, vendor-neutral, secure, flexible, configurable, self-adaptive and based on relevant standards as well as on existing Open Source components. In particular, we aim at developing a **virtual platform**, i.e., a platform that utilizes existing, already installed solutions by integrating with them, using accessible output and resources, enhancing them with AI and, if desired, feeding back AI-enhanced information into utilized systems. Thus, we do not aim at replacing existing platforms as those mentioned in [27] rather than enhancing them. Moreover, we aim at demonstrating how research results, e.g., from systematic variability management, security or data protection, can lead to platform concepts that are currently rarely used in IIoT/I4.0 platforms. Besides the desirable abilities mentioned above, following the initial decisions made in [8, 30], the platform shall be service-based and virtualized through containers. One relevant I4.0 standard to integrate the parts and pieces of the platform is the **Asset Administration Shell** (AAS) [25] that we aim to apply as self-description and interface to software components across all platform layers. The consortium discussions regarding a vision of the IIP-Ecosphere platform also emphasized the need to directly communicate with production machines, in particular, to utilize edge devices and, if feasible, cloud technology (such as the upcoming Gaia-X[3]). This re-shaped the character of the envisioned platform from a purely virtual to a mixed-virtual platform with stronger aspects of a usual IIoT/I4.0 platform, in particular providing **uniform deployment of services** to heterogeneous execution resources such as edge devices, on-premise servers or clouds.

In this whitepaper, we aim at discussing and documenting the architecture and the implementation of the IIP-Ecosphere platform. This happens in an incremental[4] fashion as, we intentionally mix requirements, architecture and implementation activities in an agile manner. With this approach, we aim at synchronizing the requirements with the architecture and ensuring that the underlying

[1] Translates to some degree to IIoT in German-speaking areas in Europe, partly based on own standards.
[2] https://www.bmwi.de/Redaktion/DE/Publikationen/Technologie/ki-innovationswettbewerb.html
[3] https://data-infrastructure.eu
[4] Along the realization state, i.e. the releases of the platform software. The version number of this white paper reflects the software release version. Thus, at the beginning some sections may be rather empty.

implementation realizes and fits the architecture. Thus, this document documents the current state at hands, while we aim at updating this document as part of upcoming releases of the IIP-Ecosphere platform. In other words, in this document, we document and discuss the current state of the platform on a feasible level of detail, the underlying implementation, decisions we made and the tradeoffs that we faced. However, depending on the state of the implementation, this document is not meant to be complete but rather to be a "living document" that is updated incrementally.

It is important to mention that this document is also meant to be a basis for discussions with the respective teams in IIP-Ecosphere[5] (mainly Think Tank "Platforms" and KI-Accelerator) and with all kinds of platform stakeholders in order to help, improve, influence or integrate with development of the IIP-Ecosphere platform. So far, it helped to onboard various co-workers and stimulated detail decisions and clarifications.

1.2 Interaction with other initiatives

Work on the IIP-Ecosphere platform is influenced by interaction with other initiatives, in particular

- The IIP-Ecosphere IIoT platform overview [27] indicating challenges and potential for future AI-based I4.0 plattforms.
- Interactions with other funded projects: DaPro[6], BaSys[7], FabOs[8], Service-Meister[9].
- Internal IIP-Ecosphere stakeholders with interest in validating the platform (in conjunction with their own approaches) or for contributing components, e.g., dedicated services. In the remainder of the funded time of IIP-Ecosphere, the "AI accelerator" work package plans to make contributions in terms of customizable (AI) services. First steps in this direction have been done, e.g., in terms of feasibility studies with the IIP-Ecosphere demonstrators or a collection of candidate services for generalization.

1.3 Structure of the document

A typical first section of a platform handbook could be a summary of the requirements to be realized. As stated in Section 1, the IIP-Ecosphere team summarized the results of the requirements collection for the platform in two other whitepapers, namely the usage view [30] and the functional/quality requirements view [8]. For pragmatic reasons, these two documents have been prepared partially before and partially while designing the platform architecture, so that they are synchronized with the work described here. In order to avoid inconsistencies, we are not repeating the requirements in this document rather than referring to [8, 30] through requirements identifiers defined there.

In Section 2 we introduce the tooling that is used for developing the architecture model and the implementation. A brief discussion of the tooling and the rationales for certain decisions is relevant at that point as the decisions significantly interact with the modeling concepts, i.e., affect the set of concepts that we practically can use for specifying, describing or realizing the architecture. Moreover, Section 2 already indicates that the ultimate outcome of our work is not "just" an architecture rather than an implemented and working platform.

In Section 3 we introduce and discuss the architecture of the IIP-Ecosphere platform, ranging from the UML profiles used, over the lower transport up to user-defined applications. This section is not only intended to present the architecture as it was designed rather than also the tradeoffs that we faced and the decisions that we made towards the actual architecture. In Section 4, we summarize

[5] See https://www.iip-ecosphere.eu/ for a summary of the project structure.
[6] http://dapro-projekt.de/
[7] https://www.basys40.de/
[8] https://www.fab-os.org/
[9] https://www.servicemeister.org/

architectural constraints that must be obeyed by the implementation. In Section 5, we discuss the representation of the platform components in terms of Asset Administration Shells, which are used as a uniform way to represent interfaces and communication among components.

One aim of the platform work in IIP-Ecosphere is to research concepts on systematically and consistently configuring such a platform, ranging from network settings over available resources or services up to the wiring of re-usable parts and pieces to IIoT-applications. In Section 6, we elaborate the structure of and the concepts of the model to specify decisions that must be made to turn alternative or generic components into an installable platform with user-defined applications. We will also discuss, how to utilize such a model, not only to validate configuration decisions, but, in particular, to automatically generate platform instances, artifacts or glue code as one means of supporting platform users to create IIoT-applications.

In Section 7, we discuss mechanisms ensuring the security of the platform. In Section 8, we detail how to obtain, install, instantiate and use (depending on the implementation state) the IIP-Ecosphere platform. In Section 9 we summarize steps on how to extend, contribute to or use the IIP-Ecosphere platform.

In particular for Sections 3 to 9 it is important to recall that the IIP-Ecosphere platform is currently under agile and incremental development, i.e., while some sections are detailed and an implementation is provided for the respective components, other components are still in planning and not yet realized. The design and implementation state will change and evolve over time as the architecture and the implementation will do. To detail the respective realization state, we will refer to the requirements in terms of realized, modified or, if needed, even deferred or excluded requirements.

Ultimately, in Section 10 we will summarize and conclude this document. In Section 11 we list references to other work that we rely on.

2 Tooling and Basic Technical Decisions

Tooling is an important topic when creating an architecture and when implementing it in terms of executable code. In this section, we briefly describe the tooling decisions made by the involved partners, as they affect the available options for modeling the architecture and for realizing it.

The **architecture** is designed using the Unified Modeling Language (UML) [24]. We will not give an introduction to UML in this document rather than assuming that the reader is sufficiently familiar with UML. As tool support, we use Eclipse Papyrus[10]. While there is a broad range of modeling tools available, in particular commercial ones, we decided to use Papyrus for two major reasons:

1) During architecture modeling already concepts for security and data privacy shall be integrated and the architecture shall be evaluated in this direction. Therefore, we will use UMLsec [19] as well as a specific security profile developed for IIP-Ecosphere. UMLsec has been successfully applied with Papyrus and with the Eclipse UML modeling tools, advocating Papyrus/Eclipse as a natural choice for our work. For applying the security concepts, the respective UML profiles must be installed and integrated into the model. For an automated security analysis, the additional Eclipse-based CARiSMA[11] tool must be installed.
2) In contrast to commercial software, Papyrus is available to the interested public as it is released as Open Source. This facilitates platform work, as we plan to release the UML model of the IIP-Ecosphere platform as part of one of the platform releases. Moreover, as it is based on Eclipse, further available tools and model translations from the Eclipse ecosystem may be utilized.

Although Papyrus offers various UML modeling capabilities, in particular the behavioral modeling for state machines, sequence or communication diagrams are currently not completely stable. This, however, affects the available options and concepts for modeling the platform architecture. Thus, in some cases, more recent modeling concepts could have been used that are not available for this reason. Unfortunately, the realization state of Papyrus also affects the layout of the included diagrams, which could be presented in more pleasing manner would some more diagramming functionality be available. This is also true for the Papyrus diagram export, which so far produces only formats (bitmap, SVG) that unfortunately can only hardly (or with some inconvenient transformation steps) be used with Microsoft Word. Thus, we include UML figures taken from the architecture model as bitmaps into this document.

Along with the architecture and the design of individual components, also architectural constraints arise, e.g., that in particular for alternative components, dependencies to underlying libraries must be private to the respective platform component, i.e., and globally used by other platform components. We will discuss the architectural constraints of the IIP-Ecosphere platform in Section 4 as a specific summary of the architecture section. Section 3 may already indicate or mention such constraints.

For **implementing** the architecture, we must integrate existing components and take into account that in particular AI services will be realized in different programming languages.

- For the **Java** components, we rely on Eclipse (so far 2021-03, version 4.19.0) with Maven[12], Git[13] and checkstyle[14] integrations. Fundamental technical decisions are documented along with the code. As we use Maven for the platform installation, a Java Development Kit (JDK) is

[10] https://www.eclipse.org/papyrus/ version 4.8
[11] https://rgse.uni-koblenz.de/carisma/
[12] https://maven.apache.org/
[13] https://git-scm.com/
[14] https://checkstyle.sourceforge.io/

required rather than a Java Runtime Environment (JRE). We just mention some of the decisions here: The dependency management and the build process are specified in Maven. Templates for code formatting and validation of the formatting are available for checkstyle in the source code repository as part of the Eclipse project for managed platform dependencies. A common logging framework was selected (`slf4j`) based on decisions of components to be integrated. Components of the IIP-Ecosphere platform are represented as individual Eclipse projects. For compliance with yet unknown edge devices, we require that (at least the lower, edge-related) layers are executable with Java 1.8 (as this is also the case for many available IoT libraries). This technical constraint may be relaxed for higher platform layers.

- While some AI methods may also be realized in Java, nowadays AI methods are frequently implemented based on **Python**. For Python services (as for Java-based services), a service execution environment is foreseen, which is responsible for the communication with related Java components, so that an AI developer does not have to work with both languages or protocol details. For the service environment, we rely on Python 3.9.6, a rather recent version as modern AI frameworks often also require a recent Python version.
- Some components require basic **technical settings** for their startup, e.g., the internet address of the AAS of the platform or basic security certificates to announce the own instance, to request or contribute information. The aim is to reduce such explicit setup information to a minimum as it is a source for inconsistencies. For this purpose, such information shall be managed centrally, instantiated into binary components or distributed via discovery protocols where feasible. Further information not required to startup a component shall be made available via the (joint) AAS of the platform. Technical settings that may be subject to modifications by administrators shall be represented in a uniform and human readable manner. For stored setup information we rely on Yaml[15], for machine-readable complex data in AAS on JSON[16].
- Components shall internally communicate via **interfaces** in order to reduce (accidental) dependencies. Alternative and optional components shall be realized as a kind of plugin and register themselves into the platform. On the Java side, we rely on the Java Service Loader (JSL) mechanism, which associates concrete implementations to their respective (descriptor) interfaces. The relation happens through a specific form of file that is evaluated by the JSL mechanism upon request. We use that mechanism to define, e.g., factory instances, to compose AAS but also to set up the component lifecycle, e.g., to handle the start and shutdown process.
- So far, no mechanisms to shield (the dependencies) of individual platforms against each other was necessary, as, e.g., technical dependency conflicts could be successfully resolved through global version restrictions. However, we are aware of the fact that in particular though external contributions, conflicts may arise that cannot be solved in this manner. Thus, for future releases, we plan to investigate, whether approaches like **OSGi** (Open Services Gateway Initiative) could help to avoid unintended or unexpected conflicts.
- All components shall provide sufficient **tests** for their functionality. Tests shall be executed during the **continuous integration** (CI) of the platform and also usual test metrics shall be recorded.

As stated in Section 1, for several reasons one objective of the IIP-Ecosphere platform is to use existing Open Source solutions wherever feasible. However, not all **Open Source licenses** are per se permissible

[15] https://en.wikipedia.org/wiki/YAML
[16] https://www.json.org/json-en.html

in industrial contexts. Therefore, the IIP-Ecosphere consortium has reviewed Open Source licenses and categorized them into four categories:

1) Usable without limitations.
2) Permissible, but potentially problematic.
3) Commercial licenses.
4) Not allowed in particular due to copy-left implications.

These categories shall be considered already during the design of the IIP-Ecosphere platform and may effectively limit potential candidates. Licenses of the first two categories may be used (with care), the remaining shall be avoided. This is in particular true for platform components that constitute mandatory core functionalities of the platform. Commercial licenses may be used depending on the decision of the installing organization. Components relying on commercial licenses shall be optional by default and, thus, their use is the decision of the using organization. Analogously, also software under not permissible licenses could be used in optional parts of the platform, but to avoid later license conflicts, licenses of category 4 shall be avoided wherever possible.

The source code of the IIP-Ecosphere platform is made publicly available in the **GitHub** space of IIP-Ecosphere[17]. Moreover, to foster transparency, the development of the IIP-Ecosphere platform happens in public. In later stages also the underlying architecture model shall be made available to support external and future developments after the project lifetime. As far as possible, components are subject to CI using the Jenkins server of the Software Systems Engineering (SSE) group at the University of Hildesheim. Upon successful builds, artifact snapshots are deployed by the CI processes to the Maven repository[18] of the SSE group. Java parts of stable releases are deployed to Maven central[19].

[17] https://github.com/iip-ecosphere/platform/
[18] https://projects.sse.uni-hildesheim.de/qm/maven/
[19] E.g., https://repo1.maven.org/maven2/de/iip-ecosphere/platform/, https://search.maven.org/artifact/de.iip-ecosphere.platform/transport

3 Architecture

The architecture of the IIP-Ecosphere platform aims at realizing the requirements collected in the project [8, 30] in terms of software. In this section, we discuss the design of the individual parts and components of the platform. Please note that as mentioned in Section 1, we follow a pragmatic agile approach to the development of the platform, which involves forward and feedback cycles among requirements, architecture and implementation. Thus, depending on the realization state, not all platform components may be completely described in this version of the document, i.e., we will work out sections incrementally depending on the realization state.

We start in Section 3.1 with an overview of the platform layers and dive then into their details. At the end of Section 3.1, we detail some further basic aspects, namely relation to reference architectures in Section 3.1.1, basics of asset administration shells in Section 3.1.2 and the virtual character of the platform in Section 3.1.3. Section 3.2 indicates the coarse-grained development streams. Section 3.3 takes up the general requirements from [8] as context for the platform architecture. As basis for the architecture description, we discuss in Section 3.4 the used UML profiles and go through the layers of the infrastructure, first as overview and then one section per layer, starting at the bottommost layer.

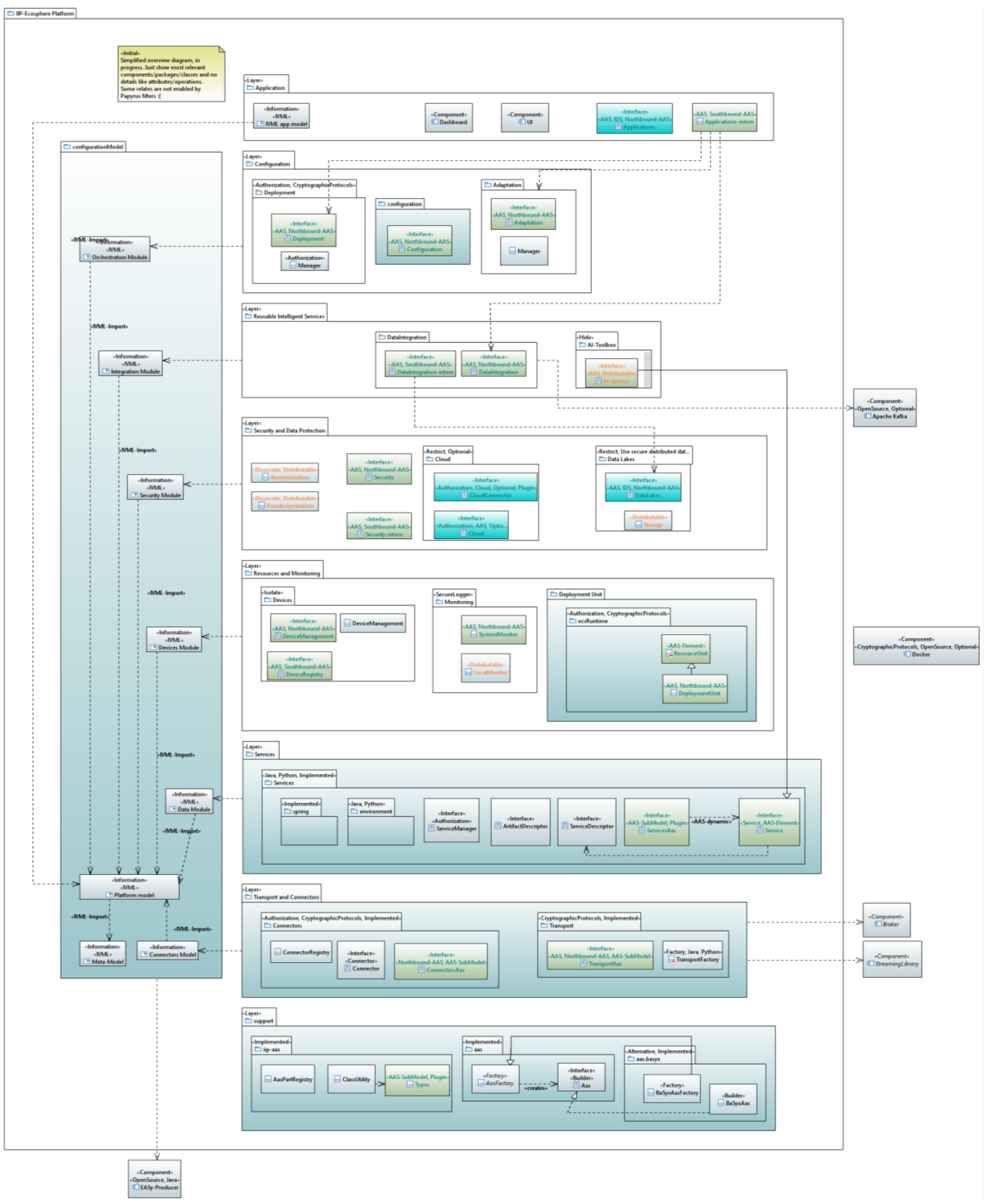
3.1 Overview

The overall architecture of the IIP-Ecosphere platform follows a layered style (see Figure 1 with only high-level relations shown) based on components and services (R4 in [8]). As far as feasible, we aim for a strict (logical) layering, so that for two adjacent layers l_l and l_u (with as “the lower layer” l_l being located below “the upper layer” l_u), only l_u (and not its transitive upper layers) shall access or call l_l directly. Moreover, there are also aspects that cross-cut visibly or invisibly in this layered structure.

- The interface description and the main call style of the IIP-Ecosphere platform is based on **Asset Administration Shells** (R7, [25]), in particular based on the “reference implementation” BaSyx[20]. An integration of AAS as well as support for realizing Administration Shells in IIP-Ecosphere style will form the bottom-most layer of the platform.
- In addition, the platform will contain an event-based **messaging** mechanism, e.g., a `Broker`, so that components and services can communicate among each other independent of the layering. Although this implies certain degrees of freedom and may be used to bypass R7 in exceptional cases, the event-based messaging shall not happen in an ad-hoc or chaotic manner undermining the layer structure. Further, uncontrolled messaging may accidentally overload the broker(s), in particular if the broker is involved in the processing of soft-realtime data streams (one potential manifestation of R10 [8]). As event-based communication and data streaming are essential to the platform, they occur on one of the fundamental layers (`Transport`) utilizing the external (abstract) components `Broker` and `StreamingLibrary`.
- **Variability management** and **consistent configuration** typically do also cross-cut layers, as variability instantiations may affect all components. This is already reflected in the requirements, where *configuration model* occurs in many different functional topics, see e.g., also for implicit information R8, R19f, R20, R28, R30, R31, R34, R40-R43, R62, R64, R73, R77, R80, R86, R89, R93-R101, R104, R107, R112, R119-R122, R131, R134 in [8]. Moreover, some layers require access to the configuration, in particular at runtime, e.g., to determine whether migrations of components are needed or how adaptations shall be enacted. However, also here a chaotic use of the configuration can easily lead to unmanageable dependencies. Therefore, we modularize the configuration along the layers (as indicated in Figure 1), and, if required, provide access to the individual configuration modules. Similarly, only some few

[20] https://www.eclipse.org/basyx/

selected mechanisms to instantiate variability shall be utilized, in particular code generation, generation of setup files and artifact selection while packaging.



IIP-Ecosphere Platform
«Initial» Simplified overview diagram, in progress. Just show most relevant components/packages/classes and no details like attributes/operations. Some relates are not enabled by Papyrus filters :(
«Layer» Application
«Information» «IVML» IVML app model
«Component» Dashboard
«Component» UI
«Interface» «AAS, IDS, Northbound-AAS» Applications
«AAS, Southbound-AAS» Applications-intern
configurationModel
«Layer» Configuration
«Authorization, CryptographicProtocols» Deployment
«Interface» «AAS, Northbound-AAS» Deployment
«Authorization» Manager
configuration
«Interface» «AAS, Northbound-AAS» Configuration
Adaptation
«Interface» «AAS, Northbound-AAS» Adaptation
Manager
«Information» «IVML» Orchestration Module
«IVML-Import»
«Information» «IVML» Integration Module
«Layer» Reusable Intelligent Services
DataIntegration
«Interface» «AAS, Southbound-AAS» DataIntegration-intern
«Interface» «AAS, Northbound-AAS» DataIntegration
«Hide» AI-Toolbox
«Interface» «AAS, Distributable» AI-Service
«Component» «OpenSource, Optional» Apache Kafka
«Layer» Security and Data Protection
«Information» «IVML» Security Module
«Dissociate, Distributable» Anonymization
«Dissociate, Distributable» Pseudonymization
«Interface» «AAS, Northbound-AAS» Security
«Interface» «AAS, Southbound-AAS» Security-intern
«Restrict, Optional» Cloud
«Interface» «Authorization, Cloud, Optional, Plugin» CloudConnector
«Interface» «Authorization, AAS, Optio...» Cloud
«Restrict, Use secure distributed dat...» Data Lakes
«Interface» «AAS, IDS, Northbound-AAS» DataLakes
«Distributable» Storage
«Layer» Resources and Monitoring
«Information» «IVML» Devices Module
«Isolate» Devices
«Interface» «AAS, Northbound-AAS» DeviceManagement
DeviceManagement
«Interface» «AAS, Southbound-AAS» DeviceRegistry
«SecureLogger» Monitoring
«AAS, Northbound-AAS» SystemMonitor
«Distributable» LocalMonitor
Deployment Unit
«Authorization, CryptographicProtocols» ecsRuntime
«AAS-Element» ResourceUnit
«AAS, Northbound-AAS» DeploymentUnit
«Component» «CryptographicProtocols, OpenSource, Optional» Docker
«Layer» Services
«Information» «IVML» Data Module
«Java, Python, Implemented» Services
«Implemented» spring
«Java, Python» environment
«Interface» «Authorization» ServiceManager
«Interface» ArtifactDescriptor
«Interface» ServiceDescriptor
«Interface» «AAS-SubModel, Plugin» ServicesAas
«AAS-dynamic»
«Interface» «Service, AAS-Element» Service
«Information» «IVML» Platform model
«Information» «IVML» Meta-Model
«Information» «IVML» Connectors Model
«Layer» Transport and Connectors
«Authorization, CryptographicProtocols, Implemented» Connectors
ConnectorRegistry
«Interface» «Connector» Connector
«Interface» «Northbound-AAS, AAS-SubModel» ConnectorsAas
«CryptographicProtocols, Implemented» Transport
«Interface» «AAS, Northbound-AAS, AAS-SubModel» TransportAas
«Factory, Java, Python» TransportFactory
«Component» Broker
«Component» StreamingLibrary
«Layer» support
«Implemented» iip-aas
AasPartRegistry
ClassUtility
«AAS-SubModel, Plugin» Types
«Implemented» aas
«Factory» AasFactory
«creates»
«Interface» «Builder» Aas
«Alternative, Implemented» aas.basyx
«Factory» BaSyxAasFactory
«Builder» BaSyxAas
«Component» «OpenSource, Java» EASy-Producer

Figure 1: Layered platform overview with indicating only relevant high-level relationships[21].

For short, the layers of the platform from bottom to top:

[21] Colors indicate the realization state and element categories. Green components indicate AAS components, turquoise layers/components are actually realized (at least in an initial version), red parts are so far not realized and may finally even be omitted (e.g., functions of semantic mapping and routing are already taken over by other components) and orange parts are currently in realization.

- **Support Layer:** The support layer realizes basic abstractions and helpful functions for the upper layers of the platform. The core aim is to reduce repetitions of non-trivial management functions or functions to create common AAS structures and to foster internal conventions, e.g., how to represent certain information in AAS. Moreover, it contains an abstraction of the underlying AAS implementation, serving for both, more flexibility (allowing to also use other implementations) and risk reduction.
- **Transport and Connectors Layer:** This layer is responsible for connecting devices among each other and with platform services using appropriate protocols and formats from the I4.0 domain. However, several protocols and formats impose different tradeoffs in functionality, performance, security and legal/normative impact. This layer integrates such protocols in a flexible manner. The role of the **Transport Component** is to abstract over relevant protocols such as MQTT[22], AMQP[23], or OPC UA pub/sub[24] to integrate the abstraction with the technology used for streaming (`StreamingLibrary`) and to provide an environment for protocol/format connectors. In contrast to recent platforms [27], where a single fixed transport protocol is not uncommon, we want to avoid making such basic decisions on behalf of the user already on this layer. Also for the streaming technology several candidate approaches with their tradeoffs are known. The idea is to prepare a flexible integration and to link this decision to the selected transport protocol. Similarly, connections to production machines and already installed platforms are abstracted by the **Connectors Component**. Such a `Connector` may utilize similar protocols as the Transport Component, but also protocols at higher semantic levels such as OPC UA providing an own information model shall be made available. In contrast to the Transport Component, which passes through given data, here only subsets of the data being available to a connector may be ingested in the platform and information/commands originating from the platform may be transported back, e.g., to reconfigure an underlying machine. The `Connectors` may optionally include functionality of the International Data Spaces (IDS)[25] for secure access to data.
- **Services Layer:** Openness and extensibility through services of different kinds, in particular AI services, are at the heart of the IIP-Ecosphere platform. To be useful for an application, services must be parameterized and orchestrated, e.g., their data (streams) must be connected to other services or connectors. While the interconnections will be handled by the Transport and Connectors Layer, the Services Layer defines the basic service interfaces (`Services`) as well as the services execution environments, e.g., for Java and Python.
- **Resources and Monitoring Layer:** To become effective, services must be deployed to resources/devices (in terms of a `Deployment Unit`) and monitored at runtime. In IIP-Ecosphere, deployment targets such as edge devices shall describe themselves in terms of AAS and perform a registration with the device registry (`Devices`), which reflects its data into the runtime structures of the platform. For deployment, the `Deployment Unit` (more precisely, the ECS runtime from [30]) receives commands via its AAS from the platform, downloads a container including the service implementations[26] and starts the container. Although techniques and frameworks for both tasks exist, according to our knowledge so far no AAS is used for this purpose. However, also the execution of the services in the container must be monitored, which may involve reusable monitoring probes provided by the platform as well as application-specific probes. The reusable mechanisms are provided by the `Monitoring`

[22] https://mqtt.org/
[23] https://www.amqp.org/
[24] https://opcfoundation.org/news/press-releases/opc-foundation-announces-opc-ua-pubsub-release-important-extension-opc-ua-communication-platform/
[25] https://www.internationaldataspaces.org/
[26] Assembling the containers is managed by the Configuration Layer as described below.

component, which (in terms of probes and signaling) is part of the service environment while the aggregation of the monitoring data happens on central IT level. The `Monitoring` component also uses the capabilities of the support layer (monitoring in terms of AAS) and the `Transport` and Connection layer (fast track signaling, alarms) and may issue alerts in generic as well as application-specific manner to further layers.

- **Security and Data Protection Layer:** Security and data protection in the IIP-Ecosphere platform encompass of two parts, 1) cross-cutting mechanisms that can be used to implement security and data protection in any component, e.g., authentication, and 2) centralized or distributable mechanisms to support security and data protection, e.g., services supporting data protection or data storage. While the cross-cutting mechanisms occur in all layers (directly or indirectly controlled through the platform configuration), this layer primarily focuses on the second part. Thus, it provides access to the overall security configuration, e.g., authentication tokens or cryptographic keys for accessing edge devices. Further, this layer realizes components (optionally) enhancing the security and data protection, e.g., stream-based services for `Anonymization` and `Pseudonymization`, external (`Cloud`) communication connectors and (optionally secure) `Data Lakes`. Data lakes/stores can be distributable components to be packed into deployment units, e.g., to buffer data on edge devices.
- **Reusable Intelligent Services Layer:** The components described so far (as well as not mentioned administrative services provided by the platforms) can be used to develop simple applications similar to existing platforms [27]. This layer shall pave the way for open, extensible and reusable intelligent services. The `Data Integration` collects data from running services (as defined during the orchestration) and integrates the data with additional information such as floor plans, order data etc. The integrated information may be stored in storages provided by the `Data Lake` component(s). The actual functionality of this component in the context of a running application is also defined in the platform configuration. Finally, the `AI-Toolbox` shall contain re-usable AI services that can be parameterized and orchestrated to form a running application.
- **Configuration Layer:** The configuration layer contains components to manage the platform configuration. The `Configuration` component is responsible for composing reusable and application-specific services and representing the information in terms of the application specific-modules of the platform configuration. The `Deployment` component is responsible for deciding which services shall be executed by which device (e.g., edge, server or cloud) depending on runtime information available in the platform configuration. Based on these decisions and device-specific information provided by a device AAS, deployment containers are created automatically and made available. Furthermore, the `Deployment` component shall take the dynamic state of the platform reflected in the platform configuration into account to optimize the container/service deployment at runtime, e.g., supported by generated service glue code or dynamic re-routing of data by the Transport and Connection Layer or the `Streaming Library`. In addition, the `Adaptation` component is responsible to decide about configuration changes to deployed services as well as selection of alternative services at runtime (supported by similar mechanisms as for runtime deployment adaptation).
- **Applications Layer:** Ultimately, a (simple) platform user interface (`UI`)[27] relying in particular on components of the `Configuration` layer as well as AI-enabled applications run on top of the platform. Applications are described by configuration modules and may ship with application-specific components, e.g., AI services or monitoring probes. Although not visible here, glue or transport code generated for orchestrated services implicitly belongs to the applications. The

[27] As discussed in [8], user interface and dashboards are formally out of scope of our funding contract. However, if feasible, we plan to realize at least a simple (Web) user interface in one of the next releases.

execution of the applications shall be visualized by (as far as feasible) generic Dashboard components. Further, external AAS-based access to selected aggregated information of the platform can be made available through secure mechanisms, e.g., IDS.

International Data Spaces (IDS) [16] is a virtual data space leveraging various standards, technologies, and governance models to enable secure and standardized data exchange in a trusted environment. IDS offers a decentralized data storage where several companies share data through IDS Connectors. Moreover, IDS allows to deploy various internal and external applications into the IDS Connectors in order to provide various services on top of data exchange processes. Furthermore, IDS introduces a so-called security profile indicating the capabilities of a Connector to maintain this secure and trusted. As discussed above, security is usually cross-cutting, i.e., while individual mechanisms may enhance or wrap IIP-Ecosphere platform connectors, e.g., to act as IDS connectors, other mechanisms may be more on the central side, such as an integration with the IDS data storage.

The full stack shown in Figure 1 is not required for all kinds of installations. E.g., on a resource such as an edge device, a cloud or a server, a **specialized runtime** is needed (ECS runtime from [30]) to take control over containers and services. The ECS runtime can be composed from a subset of the layers as indicated in Figure 2. The basic layers such as Support as well as Transport and Connectors must be present (from the Support Layer also mechanisms for dynamic network management). For managing containers, at least the deployment unit from the Resources and Monitoring Layer is needed. However, the Services Layer is optional for an ECS runtime, at least in the same container. If an ECS runtime installation also ships with all dependencies needed to run the configured services (e.g., Python and AI libraries), then it might make sense to also have the service manager from the Services Layer present. Otherwise, the Services Layer shall optionally be executable in an own container, based on the Support as well as Transport and Connectors Layers. This container would then be under the control of the ECS runtime, i.e., the local Resources and Monitoring Layer.

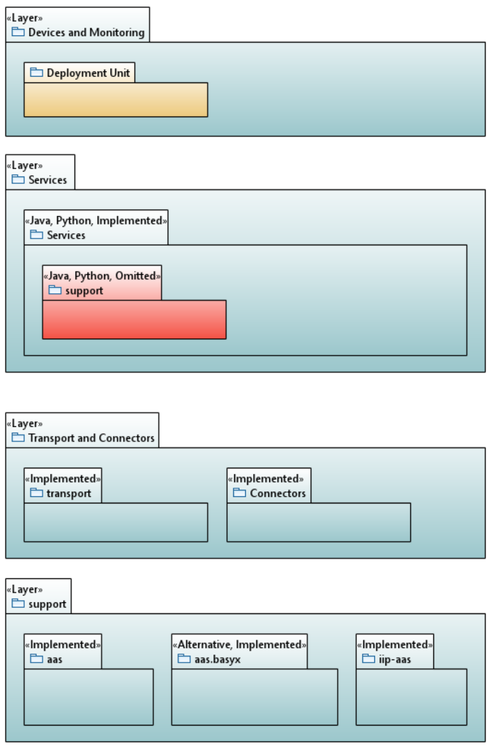


Figure 2: Layers and components required to build an ECS runtime.

The **Java 1.8 restriction** stated in Section 2 applies in particular to the layers shown in Figure 2 in order to enable compliance with unknown edge devices. Although execution on edge devices shall be virtualized in terms of containers (see [8], in particular R30), it may also be required in some settings that the ECS runtime is directly executed by a Java virtual machine on the edge device. As far as we could see at the point in time when designing the architecture of the IIP-Ecosphere platform, this is no significant limitation as relevant (client) libraries for AAS, IoT protocols or connectors or data streaming can be used there. However, we are also aware of the fact that in particular for testing further (broker) libraries may be required, where e.g., the restriction to Java 1.8 may not be fulfilled. This can be mitigated to some degree, as in an installation also equivalent functionality in terms of native programs or other programming languages are available and may be used. As also stated in Section 2, this constraint may be relaxed for the remaining layers shown in Figure 1.

3.1.1 Relation to Reference Architectures

IIP-Ecosphere aims at interrelating and adhering to reference architectures such as RAMI 4.0 [26]. Although we use an own naming of the platform layers, they map nonetheless to layers defined by RAMI 4.0 as summarized in Table 1. However, it is important to recall that the IIP-Ecosphere platform shall be a virtual platform, i.e., it shall in particular (be able to) build on existing installations without implementing a complete IIoT platform. Thus, it is not relevant to meticulously adhere to all RAMI levels, in particular not to the lower levels targeting field devices (as already scoped out in [8, 30]). In addition, our architecture includes some (crosscutting) layers that do not directly fit into the picture of RAMI[28], but are important to operations, research and contributions of IIP-Ecosphere.

Table 1: Mapping RAMI 4.0 and the IIP-Ecosphere architecture

RAMI 4.0 Axis	RAMI 4.0 Level	IIP-Ecosphere Layer/Component
Layers	Asset	Not in scope [8, 30], represented through edge AAS
	Integration	Support Layer, Transport and Connectors Layer
	Communication	Services Layer
	Information	Reusable Intelligent Services Layer
	Functional	Application Layer
	Business	On top of Application Layer via `Applications` AAS
Hierarchy Levels	Product	Not in scope, represented by data
	Field Device	Not in scope [8, 30], represented through edge AAS
	Control Device	ECS runtime [30] with deployed services, in particular Resources and Monitoring Layer with contributions from upper layers
	Station	ECS runtime [30], possibly with access to more powerful resources or UI capabilities for executing or controlling deployed services. Includies Resources and Monitoring Layer with contributions from upper layers
	Work Centers	Reusable Intelligent Services Layer, in particular `Data Integration` component
	Enterprise	Application Layer
	Connected World	On top of Application Layer via `Applications` AAS, including connected IIP-Ecosphere platforms
Life Cycle Value Stream	Type	Component and AAS types prescribing structures
	Instance	Deployed component and AAS instances

In term of the Industrial Internet Reference Architecture [18], this document can further be understood as a continuation of the usage view(point) [30], the functional view [8] In terms of a platform architecture as well as its implementation.

[28] Crosscutting aspects are better covered by IRA [18].

3.1.2 Asset Administration Shells

The IIP-Ecosphere platform aims at complying with, integrating of and extending existing standards and technologies in I4.0 (R7, R14). This applies to protocols, formats but also “interface” standards such as the Asset Administration Shells (AAS). For short and without aiming for a complete description, an AAS can be seen as information model, which consists of nested sub-models and asset specification. Sub-models may consist of typed properties, operations and heterogeneous collections of sub-model elements. AAS and sub-models can be classified as static (all information is determined when creating the AAS), dynamic (some information may change at runtime) or active (callable operations are provided). Similarly, properties and operations can be static or dynamic, whereby in the dynamic case both element types can be linked to an implementation, e.g., provided by a remote implementation server, and thus change value (access) or implementation over time.

According to the requirements (R7), the IIP-Ecosphere platform shall describe all (distributable) components, interfaces, functions and deployment targets in terms of AAS. Thus, each of the discussed layers will provide one or more AAS (sub-models) to link the layers against each other. As far as feasible, the IIP-Ecosphere platform will utilize existing approaches and standards to define the AAS, but also define own ones where needed, e.g., to characterize the capabilities of deployment targets such as edge, server or cloud devices [30]. We will detail the platform AAS and its structure in Section 5.

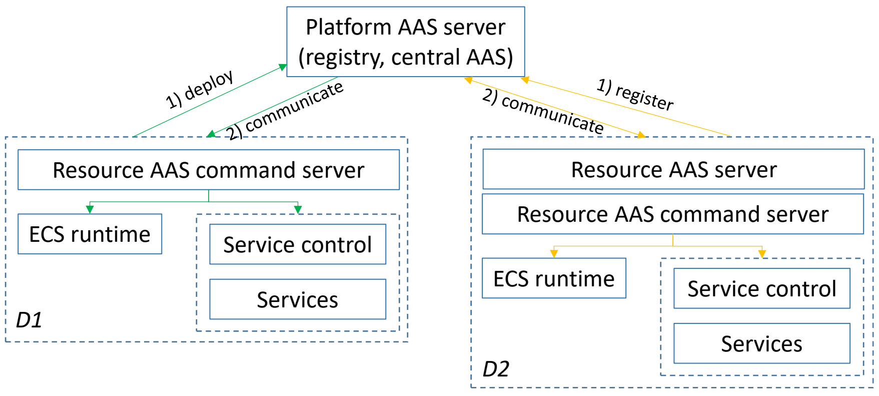


Figure 3: AAS deployment options (D1 remote deployment, D2 local deployment)

As typically several distributed compute resources are involved in a platform installation and each compute resource shall be described with an own AAS (model, sub-model or as part of joint model/sub-model), it is helpful to introduce now two basic AAS and component deployment patterns. Figure 3 illustrates the central IT side (the “Platform AAS server”) and two distributed resources *D1* and *D2*, e.g., edge devices. An AAS can be served locally and only be registered in a central registry or it can be deployed remotely to a central server. Serving an AAS locally requires a related web server process (“Resource AAS server” in D2), i.e., a further process to be executed on a resource. Deploying an AAS centrally avoids such local server processes, but may lead to increased communication with the central server and, in the case of dynamic or active AAS that allow for dynamic properties and operation calls, also to redirections of requests via the central server to the resource. To handle requests of dynamic or active AAS, the resource must run a (further) server instance, the “Resource AAS command server”. A similar server process must exist on the central IT side of the Platform AAS server to offer dynamic properties or operations. In the resource case, this “Resource AAS command server” may forward operations to further processes, or, if the processes are already known when the resource AAS is constructed, also specific server processes, e.g., for the service control running in an own container, can be linked to the AAS and directly contacted to serve AAS requests.

3.1.3 Virtual Character of the Platform

As stated in Section 1, the IIP-Ecosphere platform shall be designed as a **virtual platform** (R3), i.e., a platform that offers services on top of existing already installed platform functionality. The idea is that the Connectors component in the Transport and Connection Layer map relevant underlying platform information and functionality into the IIP-Ecosphere platform. Where feasible, this mapping shall happen in the form of AAS as it allows for an overarching information model, but also further approaches like OPC UA or MQTT may be used. We see here three alternatives, focusing on AAS as the default approach, potentially using a transport protocol like MQTT for high-speed data connections:

1. An underlying platform provides its own AAS and manages the access to selected functionality and data. Theoretically, this AAS could be mapped side-by-side into the AAS of the IIP-Ecosphere platform. Then, layers such as deployment device management, or monitoring could directly utilize the information. Therefore, a standardized AAS structure for manufacturing platforms would be desirable, but such a standard currently does not exist.
2. The AAS connector of the IIP-Ecosphere platform can map the AAS of the underlying platform into the format of IIP-Ecosphere. Of course, this adds additional overhead and in some cases a mapping may not be possible at all.
3. One of the other IIP-Ecosphere connectors provides a protocol that allows mapping the underlying platform and its operations into the IIP-Ecosphere AAS format. This approach may require manual programming, while the second approach might be realized easier through mapping and code generation.

Besides having the AAS of an underlying platform available, relevant components of the IIP-Ecosphere platform, in particular the resource management and monitoring component are required to operate with multiple AAS instances (for now based on the IIP-Ecosphere AAS structure).

3.2 Development Streams

Realizing the IIP-Ecosphere platform in one big shot is not realistic. As already indicated in the previous sections, we rely on incremental and agile development, so that delayed concepts, designs and results can be integrated flexibly, e.g., after initial experimentation with the available platform components. For the increments, we identified three development streams as indicated in Figure 4.

In the first stream, we aim at the basic functions, i.e., support for creating AAS, data transport, data connectors, basic service interfaces and management as well as the environment for running services on edge, cloud or server installations (ECS runtime). In agile spirit, these realizations must be functional and tested but not complete, e.g., it is more important to start/stop dependent services/containers rather than to perform a runtime migration of services or containers.

In the second stream, advanced functions are added and functionality missing from the first development stream may be realized. At latest, missing functionality will be integrated with the improved and advanced functions in the third development stream.

We do not indicate a detailed time schedule for the streams or functionalities here. The first development stream was completed in Spring 2021, the first release of the second stream shall be available in Summer 2021 (along with this version of the handbook) while the second development stream shall mostly be completed until end of 2021. Each stream shall be manifested by at least one release of the platform. At the point in time when this document is published, the first development stream (basic functions) is completed and basic versions for selected components of the second stream (configuration and the configuration model, artifact generation and service/resource monitoring) are available.

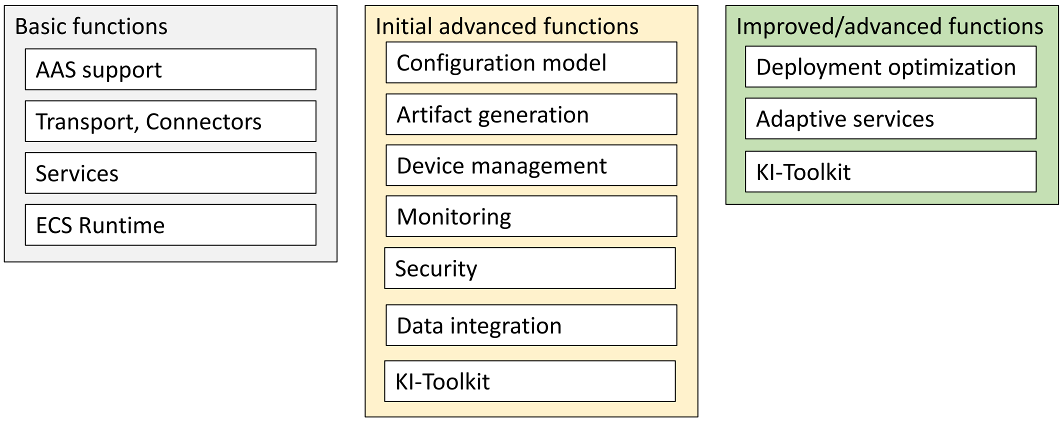


Figure 4: Development streams for the IIP-Ecosphere platform

3.3 Overall Requirements

In general, all platform layers and components discussed below must take the following general requirements from [8] into account:

Table 2: General platform requirements in [8]

Requirement	Summary
R1	Vendor and technology neutral platform
R2	Use of standards
R3	Design as a virtual platform
R4	Design based on components and services
R5	Use of Open Source, with respect to the licensing rules of IIP-Ecosphere
R6	Open for optional/commercial components
R7	Use of AAS for interfaces
R8	Use of systematic variant management techniques
R9	Means for availability
R10	Soft real-time processing (<100 ms) for production-critical functions
R11	Documentation (also in terms of this handbook)
R12	Documentation of processing steps (of applications, supporting data privacy)

As already indicated in Table 2, [8] also specifies quality requirements such as R10. Besides security and data protection requirements, there are also data frequency and volume requirements that are not so obvious, in particular as they are assigned to specific topics/components of the architecture in [8]. To provide an overview, we discuss them here on a global level for the entire platform.

In Table 3, we summarize the cross-cutting quality requirements, i.e., in particular those that may require specific considerations regarding time-critical functionality such as the (stream) processing or data transport. Although the IIP-Ecosphere platform aims at the deployment of components to edge devices, both, the services as well as the platform operations there belong to the IT realm so that OT requirements such as R35 or the OT sensor sampling frequency mentioned in R28 do not directly apply. However, a machine pulse of 8 ms (R28) as well as an hourly throughput of 7 GByte as well as an expected size of data items with 50 values (R19a) are highly relevant for judging the performance of the IIP-Ecosphere platform. As also mentioned in [8], not all data volume and frequency requirements were indicated while collecting the requirements from the partners, i.e., the platform shall aim for even higher speed (such as a 50 ms cycle time) or a throughput of 600 GByte per day.

It is also important to recall from [8], that the IIP-Ecosphere platform is primarily responsible for its mechanisms and included services, i.e., providers for services to be packaged with the platform will have to obey the quality requirements in [8] and in particular Table 5. Further, as also discussed in [8], the platform is not responsible for the quality of external services, e.g., application-specific or user-specific services (while measures may apply to report or terminate services that potentially taint given runtime requirements).

Table 3: Overview of (global) quality requirements on data frequency and volume

Requirement	Summary
R10	Soft realtime, response time < 100 ms for production critical functionality
R19a	Sample data set of 50 values of different types all 20-30 s
R19e	Output data shall be provided all 5 s
R21	Low impact on data throughput
R22	Overall platform throughput of 500 GByte per year
R28	OT sensor sampling frequency 0.2 ms, machine pulse 8 ms, step pulse 5 s, process pulse 25 s (mentioned in the explanation of the cloud requirement R28)
R35	OT sampling frequency of 2 ms
R91	7 GByte per hour as input for data integration, which may be aggregated to 2 Gbyte per hour.

As an illustration, we discuss the quality requirements now in terms of hypothetical numbers. From the data transport perspective, the requirements command that each machine can ingest a data item with around 50 values each 8 ms, i.e., 125 messages per second. This leads to at least 450.000 messages per hour (per machine/edge device). If we assume a size of 654 Byte payload (actual size of a simple JSON serialization of such as message), a source produces around 280 Mbyte per hour (just focusing on the raw data payload, i.e., not on additional information, e.g., for routing or meta-information as stated in R79). On a platform-level (R91, R22), aggregating components of the IIP-Ecosphere platform will have to cope with multiple parallel streams of this kind, which requires 26 such streams to reach the requested 7 Gbyte (in a real setting with payload and overhead). Of course, the distribution may be different, i.e., more streams at lower ingestion frequency or less streams at maximum frequency, potentially with image payloads, to reach several hundreds of GBytes per hour.

In the discussion of the individual layers/components, we will refer to these general requirements and re-iterate the argumentation only for affected layers or layers that already have been (initially) evaluated.

3.4 UML Profiles

The IIP-Ecosphere architecture model is based on three UML profiles, the IIP-Ecosphere profile introduced in Section 3.4.1, the UMLsec [19] profile for security modeling in Section 3.4.2 as well as the security and privacy profile in Section 3.4.3. All three profiles aim at classifying and defining orthogonal information to be attached to individual modeling elements. While the IIP-Ecosphere profile as well as the security and privacy profile are mostly of descriptive nature, i.e., indicate additional information such as open source licenses and component versions, the UMLsec profile is the basis for automated security analyses of UML models using the CHARiSMA tool.

3.4.1 IIP-Ecosphere Profile

The aim of the IIP-Ecosphere profile is to classify and categorize modeling elements in the IIP-Ecosphere architecture, i.e., to express orthogonal semantics in a uniform manner. We will now briefly discuss the individual concepts and parts of the profile.

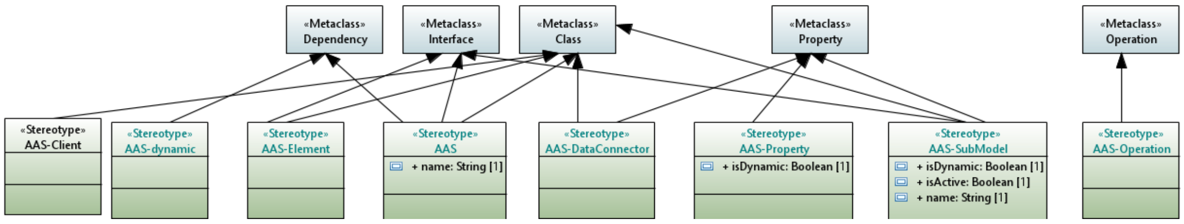


Figure 5: AAS stereotypes in the IIP-Ecosphere profile (comments cropped).

One cornerstone of the IIP-Ecosphere platform is the exploration and use of **Asset Administration Shells** (AAS, R7 in [8]). The partners decided to use AAS in particular to describe interfaces of the platform (internal, external) and communication with these interfaces in a standard-based uniform approach[29]. Thus, from an architectural point of view, it is relevant to model (structural) AAS aspects. We use AAS in terms of classes, interfaces and operations tagged by the stereotypes depicted in Figure 5. A class/interface can be marked by the «AAS» stereotype to express that there shall be an AAS providing access to the contained data. An «`AAS-DataConnector`» is a communication endpoint[30], e.g., for soft-realtime (streaming) connections. Such endpoints that are currently not part of the AAS standard[31]. An «`AAS-Property`» is a static or dynamic attribute of an AAS. UML properties may also indicate that a substructure (i.e., an «`AAS-SubModel`») shall be exhibited by an AAS. Moreover, AAS may describe functional interfaces using the «`AAS-Operation`» stereotype.

Moreover, an «`AAS-Client`» is per se not an AAS element. In the IIP-Ecosphere platform, an `AAS-Client` is a supporting class implementing how to access properties or how to execute operations. These classes shall be defined along with the respective AAS and can be tested directly against the AAS.

As our approach to modeling is pragmatic and agile, we do not aim at covering all possible aspects of AAS. Please note that the stereotypes just indicate that the respective information shall be represented in a realizing AAS. We do neither model the concrete names used in a realizing AAS, the completeness of models or sub-models nor any sequence of contained AAS elements. Besides properties that can change their value at runtime or sub-models that occur on demand, dynamic relations among AAS elements can be modeled by dependencies marked with the `AAS-dynamic` stereotype.

A second cornerstone are **services**, in particular to encapsulate platform functionality or (re-usable) AI methods. According to the profile (Figure 6), a service can be modeled as an interface (with implementation aspects hidden) or as a class (i.e., a namespace with properties and operations). Related to services are (platform) **connectors** that ingest data into the platform or are involved in offloading data/processing to other platforms or a cloud. Typically, for one connector type multiple alternatives are offered and also additional connectors can be added (openness, e.g., R14, R16). To indicate these elements, the profile contains a generic «`Connector`» stereotype that can also be used to indicate Cloud connectors and (for security) optional IDS connectors.

[29] Design guidelines for AAS must still be agreed upon by the partners or discussed with other projects. Although this affects the implementation, the actual AAS design guidelines are outside the scope of the architecture, i.e., we focus here just the relevant aspects such as properties, operations and links/dependencies.

[30] Following [30], we do not use the term "endpoint" in this document rather than "data connector". For links among data connectors and endpoints, we use the terms "relation" [30], "data flow" or "data path" [8].

[31] There is ongoing work on standardizing communication endpoints. If possible, we will adopt this upcoming standard in a later release.

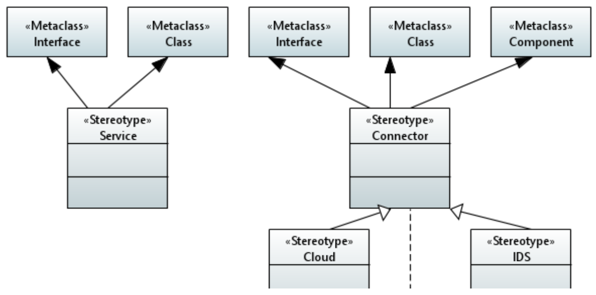


Figure 6: Service and connector stereotypes in the IIP-Ecosphere profile (comments cropped).

In IIP-Ecosphere, services shall be deployed in terms of virtualized **containers**. Therefore, implementing elements can be marked as «`Container`» (Figure 7). Further, besides services, individual platform components can be marked as «`Distributable`» while parts not marked as «`Distributable`» shall remain part of a central platform installation.

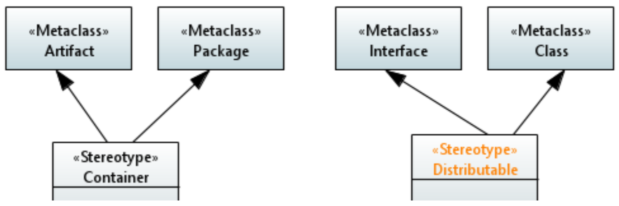
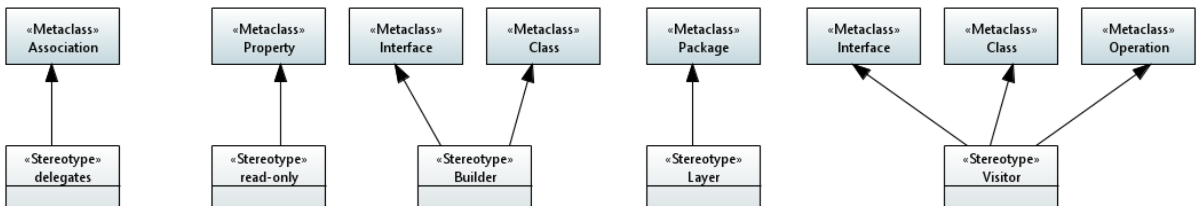


Figure 7: Container and distribution stereotypes in the IIP-Ecosphere profile (comments cropped).

To simplify the models, i.e., to avoid repetitively modeling of typical mechanisms or collaborations, the profile allows indicating **architecture, design or implementation patterns**[32], to explain/detail a model element in an uniform manner, but also to guideline the implementation. Figure 8 depicts simple patterns like architectural layers[33], delegation of control to another element via an association, read-only attributes (without corresponding setter)[34], builder pattern[35] (or classes that shall use this pattern to realize read-only attributes) or visitor pattern[36]. Figure 9 illustrates stereotypes for marking an object factory[37] (an exchangeable mechanism that creates instances) or plugins as one form to extend platform functionality at defined points. A «`Plugin`» is detailed by a «`PluginType`» providing more information on how to implement/register the plugin. The default type is JSL, the Java Service Loader[38], a simple mechanism on an implementation to its (descriptor) interface without direct dependencies in code. These patterns support the openness of the platform, e.g. extensibility for optional components in R6 [8].



[32] An important reference here is the GoF book [11], but for simplifying the understanding, we just provide some Web references.
[33] https://en.wikipedia.org/wiki/Multitier_architecture
[34] UML and Papyrus offer a read-only meta-property of the meta-class `Property`. However, displaying this information in the diagram is tedious, so we just define the corresponding stereotype `read-only`.
[35] https://en.wikipedia.org/wiki/Builder_pattern
[36] https://en.wikipedia.org/wiki/Visitor_pattern
[37] https://en.wikipedia.org/wiki/Factory_method_pattern
[38] https://docs.oracle.com/javase/9/docs/api/java/util/ServiceLoader.html

Figure 8: Basic architecture/implementation patterns in the IIP-Ecosphere profile (comments cropped).

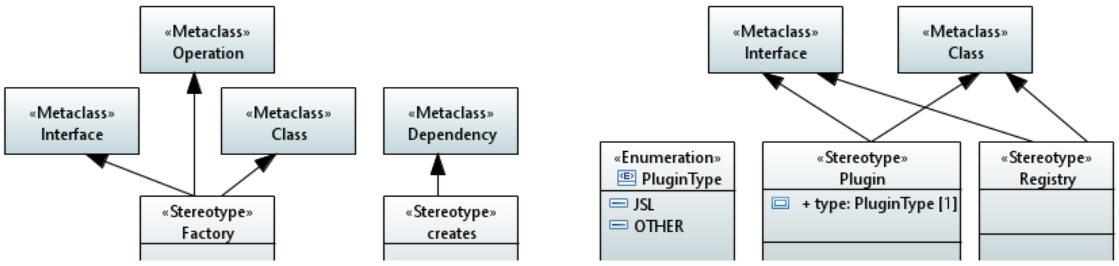


Figure 9: Factory and plugin/registration patterns in the IIP-Ecosphere profile (comments cropped).

Due to the AI nature of IIP-Ecosphere it is important to recognize that code written in various **programming languages and under licenses** must be integrated (R5, R6, R113 in [8]). The stereotypes in Figure 10 allow indicating these two dimensions also to locate potential pitfalls. External components not marked with the `OpenSource` stereotype are meant to be commercial/proprietary and shall be only used as alternatives rather than mandatory or default components, i.e., their installation shall be left to the user's choice. «`OpenSource`» components are characterized by their license(s) and their `version`. If no version is indicated, no decision was made so far, i.e., the component was not integrated so far.

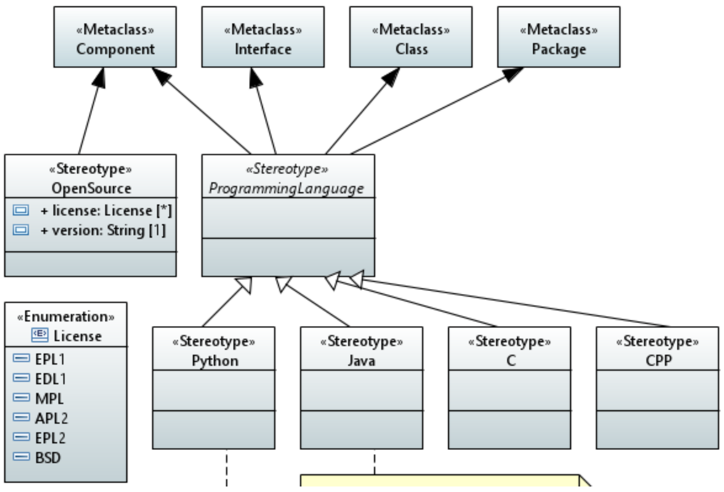


Figure 10: Licenses and programming languages in the IIP-Ecosphere profile (comments cropped).

Within the architecture model, it is also relevant to mark the **maturity status** of individual parts, e.g., to distinguish initial models from detailed models that are actually implemented (Figure 11). Some parts (within models) may not have been realized so far and can be marked with «`Omitted`». The maturity status can be attached to a model or individual modeling elements if applicable, but in particular also to comments. We use comments to document the aim/contents of each UML diagram (often cropped in this document) and may then attach the maturity stereotype to that comment.

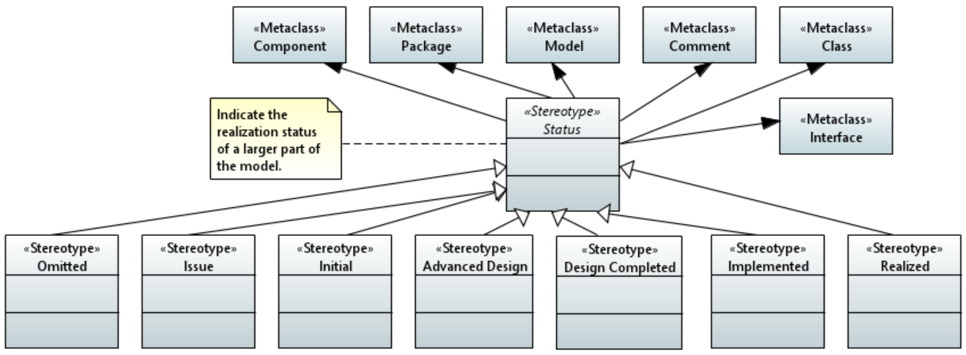


Figure 11: Maturity status for comments, packages or models.

Another special contribution of the IIP-Ecosphere platform is to aim for an encompassing and consistent **configuration model** that ranges from devices over services to service orchestration and covers static pre-runtime and runtime aspects, e.g., for deployment optimization or self-adaptation (R120-R126, R107 in [8] as listed in Section 3.1). We use the Integrated Variability Modeling Language (IVML) [7] to describe the configuration model and we indicate aspects of the variability modeling in the architecture models. IVML is realized in terms of the EASy-Producer [31] toolset, an external open source component that we integrate into the IIP-Ecosphere Platform. For short, the IVML configuration meta-model of IIP-Ecosphere (represented as information items marked with the `IVML` stereotype shown in Figure 12) defines the structure, configuration options and validity criteria for all potential platform instances. The configuration (also an IVML model) instantiates the meta-model and details the configuration decisions for a specific platform instance, e.g., on which server the platform AAS will be located, which concrete services are available etc. One particular architectural aspect is the structure of the IVML (meta-)model and its relation to the layers of the platform. The (meta- and configuration) model consists of individual modules (called projects). We specify this decomposition of the configuration model into modules (represented as information items tagged with «`IVML`») in terms of dependencies decorated with «`IVML-Import`». Ultimately, mechanisms of EASy-Producer will validate the configuration and automatically modify, include, exclude, generate or package artifacts that finally make up the configured platform instance.

A further architecturally important aspect are the components and classes realizing the variations defined in the configuration model (i.e., the implementation parts to be included, excluded, modified, linked with glue code etc.). We use the Software Product Line [32] notion of «`Alternative`» or «`Optional`» artifacts and mark the respective components using the stereotypes shown in Figure 12. Examples are alternative transport connectors (one must be selected) or optional components (that can be part of the platform instance or not) such as IDS or cloud connectors. As these stereotypes are intended to be illustrative and explanatory rather than for defining a configuration or an artifact model, we do not include further variability details as done in typical variability profiles, e.g. in [33, 12]. In particular, components marked with «`Alternative`» or `Optional` shall be designed and implemented carefully with respect to their dependencies, i.e., leaving out an «`Optional`» component or replacing an «`Alternative`» component must not render a platform instantiation invalid unless the governing configuration is invalid.

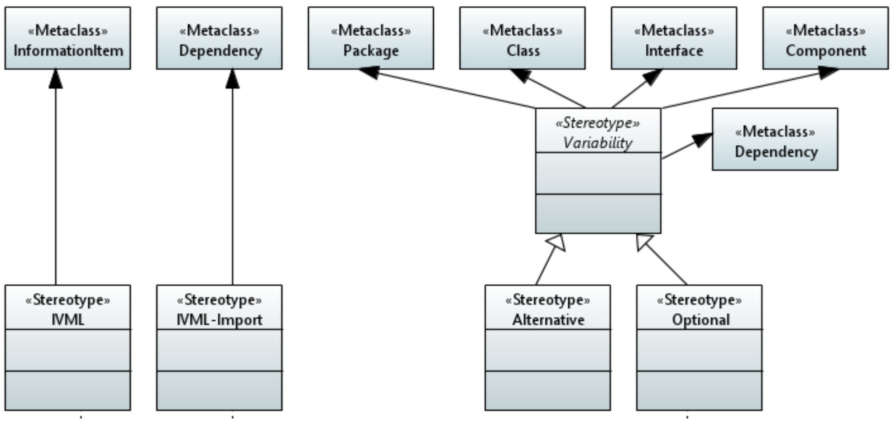


Figure 12: Configuration modeling and variability management stereotypes (comments cropped).

One form of instantiating a configuration model is to **generate code**, e.g., data transport code or glue/binding code between a hand-crafted implementation and platform interfaces to ease the development of consistent applications. To indicate that parts of the architecture are intentionally left open as they will be filled through generation from the platform configuration model during platform instantiation, we mark these parts by the «`Generated`» stereotype shown in Figure 13.

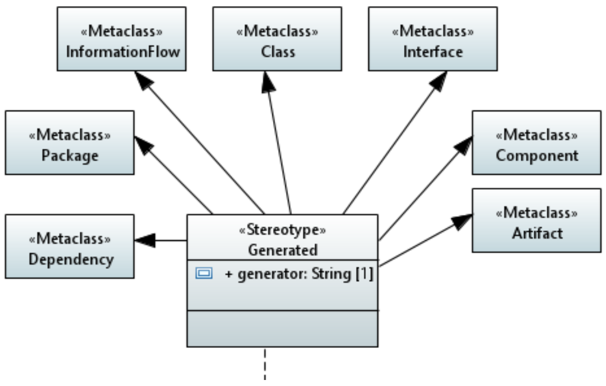


Figure 13: Stereotype for generated code (comments cropped).

The IIP-Ecosphere platform shall provide **self-adaptive capabilities at runtime** based on the (runtime part of the) configuration model (see R102-R109 and R120-R126 in [8]). Examples of platform elements that could be adapted are container deployment locations or actual AI services used in the same deployment location. To indicate model elements that are related to adaptation, the IIP-Ecosphere profile defines the stereotype «`Adaptation`» (Figure 14). One particular example for the application of this stereotype is to mark states in a state diagram that would not be needed without self-adaptation functionality, but which are required for self-adaptation.

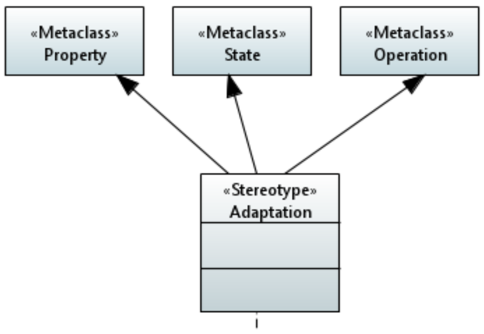


Figure 14: Marking model elements as support for self-adaptation.

In some situations, we also include **experimental** components in the architecture, in particular to introduce a certain concept that is used later on in the discussion, e.g., in a validation. To indicate such experimental components, we utilize the stereotype shown in Figure 15. For various reasons, experimental components may not be found in the IIP-Ecosphere source code repository and they may also not be subject to the continuous integration.

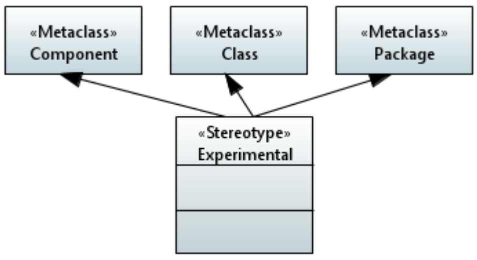


Figure 15: Marking experimental components

In Papyrus, it is possible to define a stylesheet to adapt the formatting of modeling elements based on the applied stereotypes. We will use this mechanism to mark important stereotypes, e.g., issue comments or omitted elements, in a uniform manner such as uniform fill or text colors.

3.4.2 UMLSec Profile

UMLsec[39] provides a model-based approach to develop and analyze security critical-software, in which security requirements such as confidentiality, integrity, and availability are expressed within UML diagrams. The UMLsec language is provided as a UML profile and can be imported into existing UML tools. In UMLsec, different stereotypes and tags are used to annotate UML diagrams with security properties. UMLsec provides various security checks to ensure the annotated properties. The CARiSMA tool performs the corresponding security checks. The idea of UMLsec is to provide maximal analysis power while allowing to use everyday development tools for the development process.

While the UMLsec profile is defined as a light-weight UML extension, it is also possible to define it using heavyweight extensions to specify the change of semantics. One can make use of an extended metamodel (analysis model). This analysis model provides the possibility of more complex analysis by extending the basic UML metamodel.

As mentioned above, UMLsec provides different security checks to verify whether a security property in a system is violated, and a security mechanism is needed to restore it. In this document, we explain two security checks, namely *secure links* and *secure dependency*. *Secure links* is used for the description and the analysis of secure data flows over connections between the artifacts in a UML deployment diagram, which describes the physical layer of a system. *Secure dependency* ensures that various dependencies between interfaces in a structure of a system model respect the security requirements of the data communicated across them.

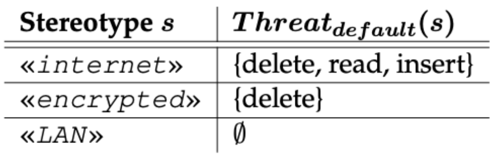
3.4.2.1 Secure Links Check

The physical layer of a system is modeled by a deployment diagram, including physical nodes, the communications between them (modeled by links), the (software) artifacts and the dependencies between the artifacts. The *secure links* annotation enables one to ensure the security of communications in a physical layer.

In UMLsec, to perform a security check, *adversary patterns* are required. Such patterns specify the potential access paths threatened by a certain attacker. Table 4.1, represents the *default* adversary, as

[39] https://rgse.uni-koblenz.de/jj/umlsec/

an example of an adversary pattern. For a given adversary of type A, the set $\text{Threat}_A(s)$ specifies which kinds of actions the adversary can apply to a node or a link marked with the stereotypes. For example, considering an unencrypted *internet* communication link, the default attacker ($\text{Threat}_{\text{default}}$(internet)) can *delete*, *read* and *insert* messages transmitted over this link.



Stereotype s	$Threat_{default}(s)$
«*internet*»	{delete, read, insert}
«*encrypted*»	{delete}
«*LAN*»	$\emptyset$

Figure 16: The UMLsec default adversary pattern

The stereotype «`secure links`» implies the following conditions: for each dependency annotated with stereotype $s \in$ { «`secrecy`», «`integrity`», «`high`» } between two artifacts deployed on two nodes n, m, we have a communication link l between n and m with stereotype t such that:

- s = «`high`», implies that $\text{threat}_A(t) = \emptyset$,
- s = «`secrecy`», implies that read $\notin \text{threat}_A(t)$, and
- s = «`integrity`», implies that insert $\notin \text{threat}_A(t)$.

For instance, if a communication link between two nodes n, m are annotated with «`internet`», and the dependency between two artifacts a_1 (deployed on node n) and a_2 (deployed on node m) are annotated with «`high`», then the security constraint associated with the stereotype «`secure links`» is violated: the dependency annotated with «`high`» demands that the set of threats of an adversary is empty, however, the communication link is annotated with «`internet`», meaning that the adversary is capable of reading, deleting, or inserting messages over the link between n and m. Consequently, the security requirement of the communications is not supported.

3.4.2.2 Secure Dependency Check

In UML, a dependency between two model elements is a relationship that denotes a model element requires other model elements for its specification or implementation. In other words, the complete semantics of the client element is either semantically or structurally dependent on the definition of the supplier element. The stereotype «`secure dependency`» implies that the security requirements have to be supported by both sides of the dependency (respective classifiers) and the dependency itself.

Later in this document (within the data security layer) we describe the UMLsec profile within the architecture model of the IIP-Ecosphere. Furthermore, we show how a CARiSMA check can be performed on such models to verify the security level of the architecture models.

3.4.3 Security and Privacy Profile

To enhance the security and privacy of the platform, we create a dedicated Security and Privacy UML profile. The purpose of this UML Profile is to provide a catalog of security and privacy mechanisms to annotate corresponding UML models and the architecture of the platform. With such an annotation process, we can express appropriate mechanisms on a high abstraction level. In this way we give an overview of a complete security and privacy framework to the developers and designers of the system. Furthermore, this dedicated profile enables privacy and security audits. Audits lead to increased software quality.

The catalog represented by the profile introduces a means to structure privacy and security mechanisms in multiple abstraction levels. Namely, design strategies, sub strategies, patterns, and privacy enhancing technologies (PET) that can be applied to design of the platform.

The feature model in Figure 17 shows the four abstraction levels of the privacy and security concept. A feature model describes a set of features and their relations, here privacy and security features and their relations. The main structure of the feature model is hierarchical. But there are relations that do not follow the hierarchical structure. Some features may require other features. For example, `Authorization` requires `Authentication`. Other features may exclude each other, for example, `Anonymity Set` and `Notify`.

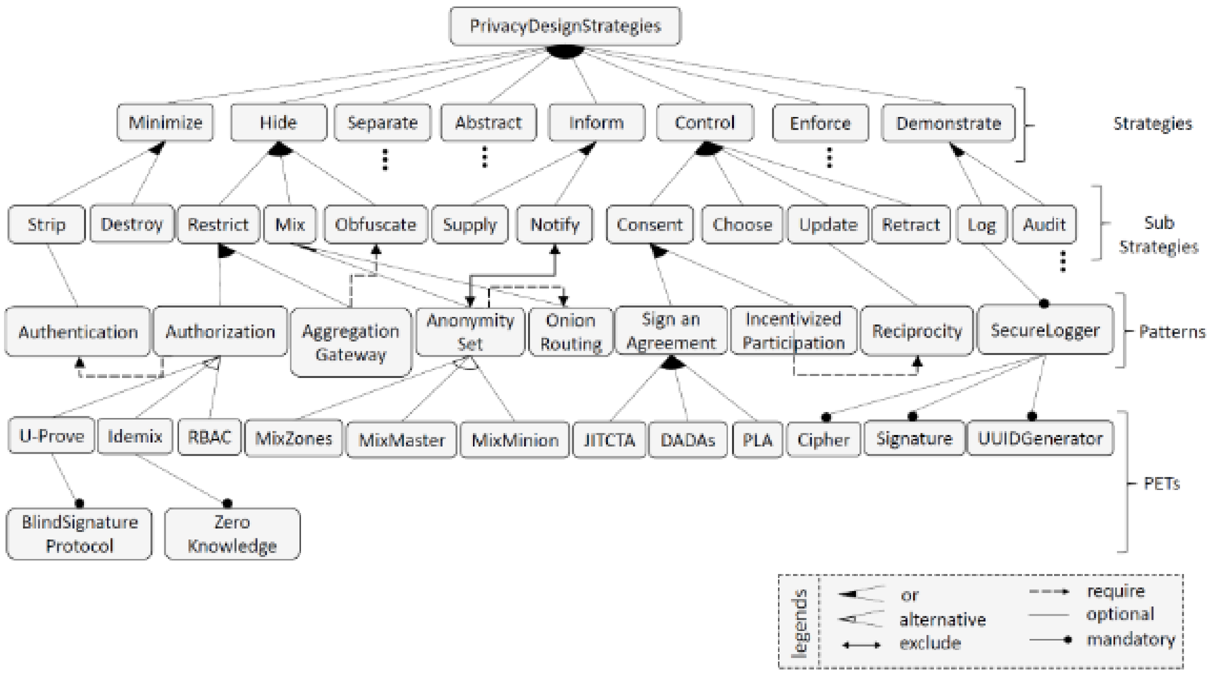


Figure 17: An excerpt of the feature model including privacy design strategies, sub-strategies, privacy patterns, and PETs (cf. [1] Figure 6.5).

Figure 18 shows the UML privacy and security profile we created based on the feature model shown in Figure 17. We adopted the hierarchical structure of the feature model and recreated it in terms of a UML profile. The profile has the same 5 levels as shown in the feature model, the root level `PrivacySecurity`, the `Strategy` level, `Sub Strategy` level, the `Pattern` level and the PET level.

For each level we defined to what elements in the model the stereotype can be annotated with that stereotype. For example, we can use the «`Privacy`» security stereotype to annotate components and packages. Stereotypes from the strategy level, like `Hide`, can additionally be used to annotate classes and interfaces.

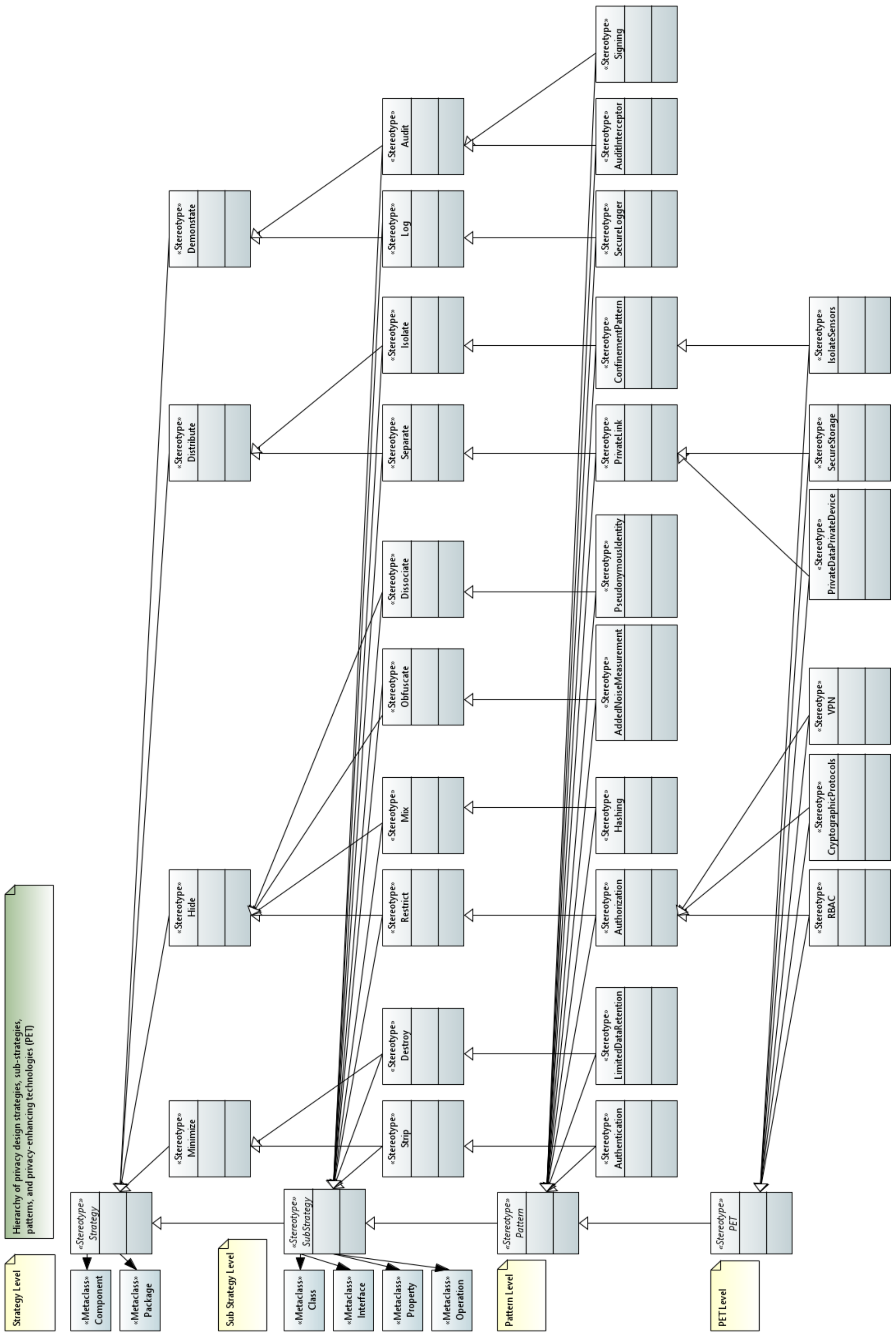


Figure 18: The Privacy and Security UML Profile (excerpt, cropped).

We discuss now specific examples on how the stereotypes from the UML privacy and security profile can be used. In IIP-Ecosphere, the profile can be used by business partners to communicate with each other about business secrets or to communicate with expensive production equipment. Unauthorized access to the system can cause severe damage to the companies using and trusting it.

We will now use the stereotypes to annotate our model with the role-based access control (RBAC) stereotype. In RBAC, the access rights are assigned to roles. Then individuals are assigned to the roles. This has multiple advantages over assigning roles directly to individuals. RBAC is a privacy enhancing technology. In our hierarchy, the RBAC PET is located in the `Minimize` Strategy, the `Restrict` Sub Strategy and the `Authorization` Pattern.

Figure 19 shows an interface (to be introduced in Section 3.6.3) annotated with the «`Authorization`» stereotype, and in that interface the `write` method is annotated with the RBAC and the Log stereotype.

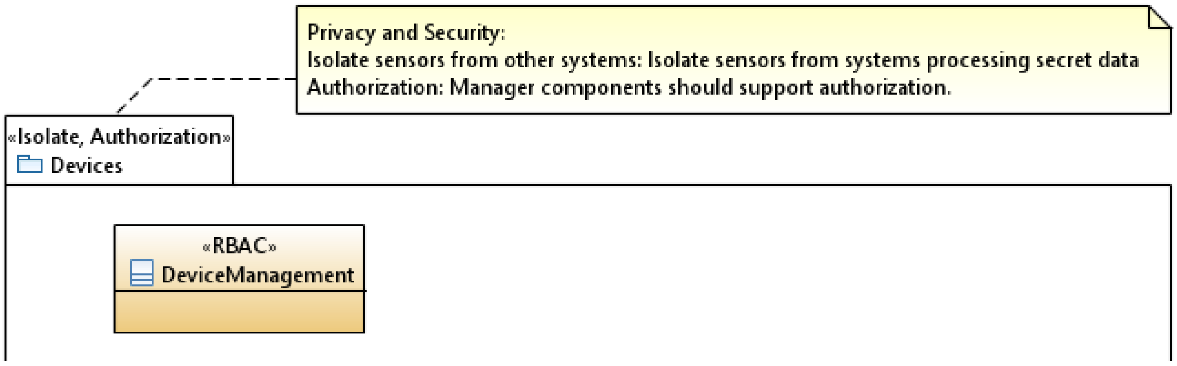

Privacy and Security:
Isolate sensors from other systems: Isolate sensors from systems processing secret data
Authorization: Manager components should support authorization.
«Isolate, Authorization»
Devices
«RBAC»
DeviceManagement

Figure 19: Interface annotated with Privacy and Security stereotypes

Figure 20 shows how the serialization package (to be introduced in Section 3.6.2) is annotated with the «`Hashing`» and «`Signing`» stereotypes. The contents of the package has been omitted in order to focus on the stereotype application. Serialization is one important part of storing, loading and transmitting data. With «`Hashing`» we can increase the integrity and with signing we can verify the origin of the data.

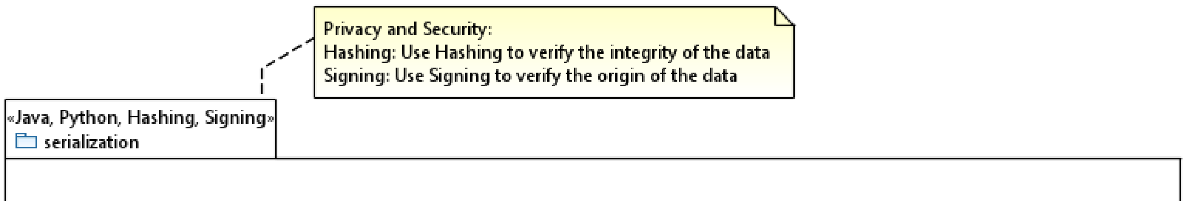

Privacy and Security:
Hashing: Use Hashing to verify the integrity of the data
Signing: Use Signing to verify the origin of the data
«Java, Python, Hashing, Signing»
serialization

Figure 20: Package annotated with Privacy and Security stereotypes

Table 4 shows an excerpt of strategies, sub strategies, pattern and PETs that are suitable for the system. The design strategies, patterns, and privacy enhancing technologies are based on the work of Ahmadian [1]. The strategies are adapted from Hoepman [15].

Table 4: Design strategies, patterns, and privacy enhancing technologies for the IIP-Ecosphere architecture. (cf. [1] Appendix F).

Strategy	Sub Strategy	Pattern	PET
Minimize	*Strip*	Authentication	
	Destroy	Limited Data Retention	
Hide	*Restrict*	Authorization	RBAC, Cryptographic Protocols, VPN
	Mix	Hashing	
	Obfuscate	Added Noise Measurement	
	Dissociate	Pseuodnymous Identity	
Separate	*Distribute*	Private link	Private Data private Device, Secure Storage
	Isolate	Confinement Pattern	Isolate Sensors from System
Demonstrate	*Audit*	Audit interceptor	
		Signing	
	Log	Secure logger	

3.5 Support Layer

The Support Layer aims at providing useful common functions and abstractions to ease the realization of the IIP-Ecosphere platform. Thus, it is more a support library than a full layer, i.e., it does not provide an own AAS representing the interface of the layer. However, even as a support library it is used by the Transport and Connection layer, i.e., the support functionality logically forms an own layer. Below, we detail the AAS abstraction in Section 3.5.1, the network manager in Section 3.5.2 and the lifecycle support in Section 3.5.3.

3.5.1 Asset Administration Shell Abstraction

A core aim of the Support Layer is to abstract over the used AAS implementation. This allows for flexibility (the AAS implementation can be exchanged), but also to mitigate risks of impacts by the currently evolving AAS standard and its implementations. Thus, the abstraction described here aims at supporting the application of AAS for the description of interfaces (R7), the application of standards (R2) and enables openness for different AAS implementations, including potential upcoming commercial implementations (R6). Further, an abstraction contributes to the IIP-Ecosphere goal of increasing interoperability, as currently several AAS implementations do exist that potentially do not interoperate (see LNI Testbed Asset Administration Shell[40]). Thus, an abstraction also mitigates development risks, as the current rather dynamic external implementation activities may lead to partially disruptive technical changes.

Figure 21 depicts the three parts of the support layer. The core is the `aas` component, which defines the IIP-Ecosphere abstraction of AAS. The `iip-aas` component on top utilizes the AAS abstraction to add further functionality that eases the realization of the IIP-Ecosphere platform, e.g., mechanisms how to dynamically link alternative and optional AAS sub-models of different components into the platform AAS. We employ BaSyx as the default AAS implementation of the IIP-Ecosphere platform. The `aas.basyx` component implements the interfaces defined by the `aas` component and provides a factory implementation to transparently instantiate abstract concepts based on the underlying BaSyx implementation. As BaSyx ships with a large number of dependencies and not all of these dependencies may be needed on an edge device, e.g., when deploying an AAS remotely to a central server (cf. Section 3.1.2) persistent storage to a database is not needed, we aim for a dependency-reduced `aas.basyx` component and an `aas.basyx.server` component including all dependencies.

[40] https://lni40.de/lni40-content/uploads/2020/11/AAS-testbed.pdf

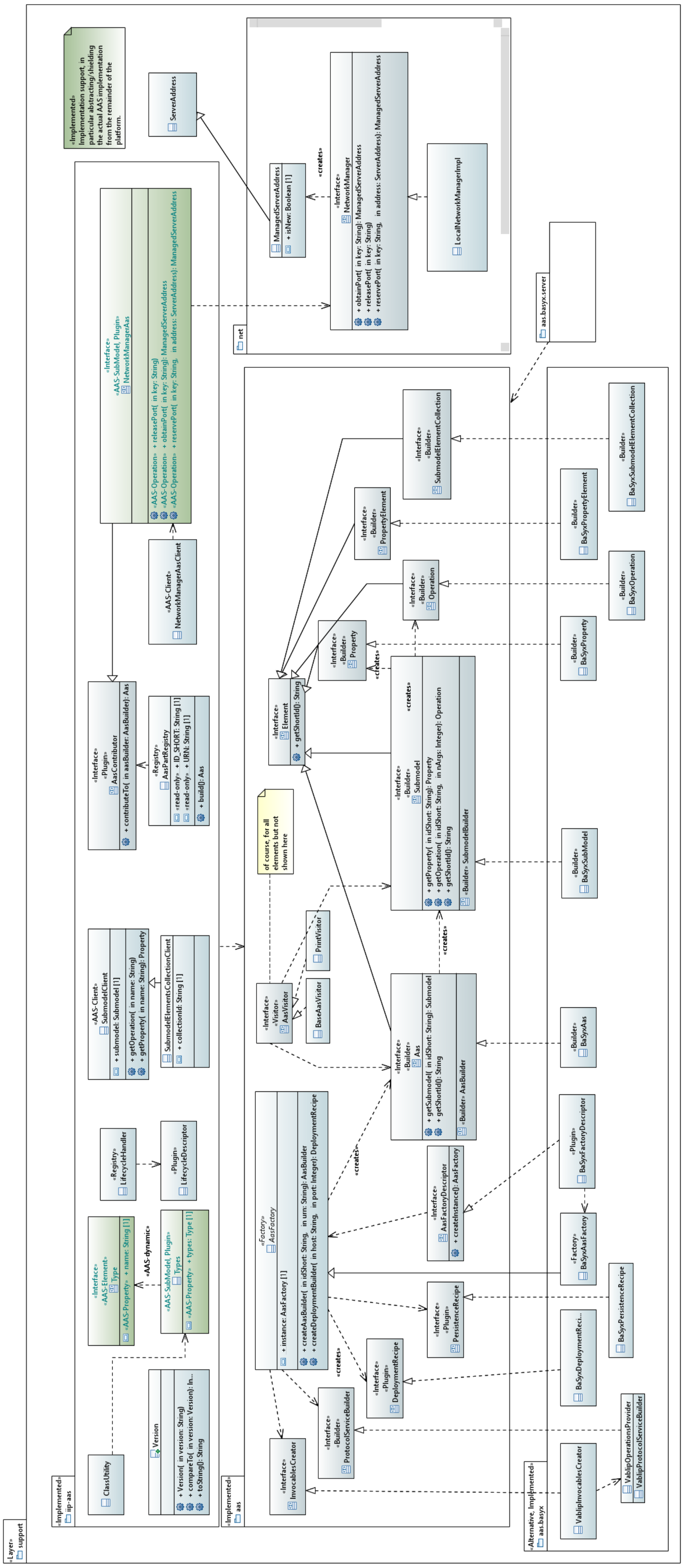


Figure 21: Support Layer overview (only selected classes/interfaces/operations are shown)

The `aas` component mainly consists of the instance factory as well as interfaces defining the functionality to be provided by an AAS implementation[41]. It is important to distinguish here between AAS interfaces (such as `Aas`, `SubModel`, `Property` and `Operation` following the AAS meta-model [25]) and the associated (nested) builder interfaces used to build concrete instances of these interfaces. The AAS interfaces provide access to the respective information and, to a certain degree, also allow for modifications, in particular if the interface represents a connected, deployed AAS element. In contrast, the builder interfaces are responsible for creating these instances, allowing for a concise coding style and additional consistency checks, e.g., preventing typical usage errors of the underlying AAS implementation.

Instances of the AAS interfaces can only be created through the factory and the builders, i.e., the top-most AAS-builder can be obtained from the `AasFactory` and all subsequent builders for nested AAS elements (sub-models, element collections, properties, operations) can transitively be obtained from the actual builder. Specific extensions to the typical AAS interfaces are the deployment support (`DeploymentBuilder`), the remote protocol support (`InvocablesCreator` and `ProtocolServiceBuilder`) as well as the `AasVisitor`. The `DeploymentBuilder` aims at realizing and encapsulating typical deployment recipes, such as local or remote AAS deployment. The protocol support encapsulates a specific remote communication protocol to implement the dynamic/active behavior of an AAS. This builder creates function objects delegating the respective operation to the protocol. The function objects can be attached through the `aas` interface to the underlying AAS implementation. This can happen in a straightforward manner if an AAS has direct access to an implementing instance such as a service. However, if AAS and implementation must be separated, e.g., due to remote deployment of the AAS or due to a programming language/process border (e.g., Python for AI services), a (remote) communication protocol must be utilized. As several options for such protocols do exist, e.g., REST, RMI, GRPC, etc., it is not possible just to provide a single protocol abstraction rather than to allow for openness. Therefore we offer a pair of interfaces, the `InvocablesCreator` being responsible for the function objects to be attached to an AAS (this is just a kind of factory rather than a builder) and a related `ProtocolServiceBuilder` being responsible for building up a server/service instance and registering the actual implementation functions for the function objects. Ultimately, the `AasFactory` is responsible for creating a matching pair of instances for a given protocol.

In addition, the abstraction encompasses an `AASVisitor`. As usual, a Visitor allows traversing a data structure in an extensible, polymorphic manner (based on inversion of control) without knowledge about the structure, need for explicit alternatives over types or type casting. Moreover, visitor instances can be applied to any element in the data structure and, thus, perform a partial traversal. Further, there is usually not a single Visitor implementation rather than many, each one for a specific purpose. Besides the interface, we provide the `PrintVisitor` which emits the structure of the AAS in textual form in particular for testing/debugging. Further, we provide, as usual, an empty basic implementation, the `BaseAasVisitor` to be used by visitor implementations to handle changes to the visitor interfaces in a default manner, i.e., further AAS elements will then not per se lead to a compile error.

A concrete implementation of the AAS abstraction provides an AAS factory. Except for the visitors, which are based on the abstraction rather than a concrete implementation and, thus, can directly be created on purpose by client code, instances of all other concepts can be obtained directly or indirectly

[41] We follow a pragmatic and agile approach here, i.e., we follow the meta-model in [25], but we do not aim to be complete from the very beginning. We add interfaces and operations only on usage demand. Ultimately, at latest at the end of the IIP-Ecosphere project, the abstraction shall be complete with respect to the most recent, implemented version of the AAS specification.

from the AASFactory. Concrete AAS factories are supposed to announce/register themselves via the AasFactoryDescriptor and the Java service loader mechanism[42], so that just the presence of an AAS implementation on the Java classpath enables the abstract AasFactory to create concrete instances.

The default implementation of the AAS abstraction is based on Eclipse BaSyx. The aas.basyx component implements the interfaces, typically in terms of adapter/wrapper[43] classes, i.e., classes that delegate the actual operations to the underlying BaSyx implementation. As remote communication protocol, the default implementation offers an extensible form of the BaSyx Virtual Automation Bus (VAB, in variants TCP, HTTP and HTTPS) through the VabIipInvocablesCreator and the VabIipOperationsProvider. Further, external protocols may be added using the ProtocolCreator (and the related JSL ProtocolDescriptor, both not shown in Figure 21).

In an installation setting, various kinds of AAS servers may be used, e.g., in-memory servers on edge devices or servers with persistent storage of the AAS on a central IT side. However, the different forms of servers imply different dependencies, in particular, database dependencies may not be feasible in resource limited environments such as edge devices as already mentioned above. Thus, implementations of the AAS abstraction are encouraged to reduce dependencies where ever possible to allow for execution in all environments. For IT side installations, all dependencies may have to be included to allow, e.g., for persistent database storage. For this purpose, we separate the AAS implementation into two parts, the (client-side) AAS for all environments and the server side. To announce the server side, we introduce the AasServerRecipeDescriptor (not shown in Figure 21), which, if present, hooks the server component with all its dependencies into the AASFactory and makes such servers transparently available.

The iip-aas component paves the way that AAS (sub-models) for the different IIP-Ecosphere platform layers can be collected and deployed as a single representation of a running resource depending on a given deployment mode. Therefore, the iip-aas component defines the AasContributor interface and the AasPartRegistry. The AasContributor is a plugin interface supposed to be implemented by upper platform layers to create the respective AAS (sub-model) and to register the implementing function objects with the protocol builders. An AasContributor can indicate whether prerequisites are met so that its AAS can be created. Instances of AasContributor are supposed to be announced/registered via the JSL mechanism. The AasPartRegistry provides access to those plugin instances and, e.g., triggers the creation and the deployment of an entire AAS for an installation. Thus, interfaces marked with the stereotype «AAS» (from the IIP-Ecosphere profile, see Section 3.4) are supposed to be implementations of the AasContributor interface and to announce themselves via JSL.

Moreover, iip-aas provides common classes to build up parts and pieces of platform AAS instances according to IIP-Ecosphere AAS conventions. Examples are the setup of AAS information via JAML or the ClassUtility which turns Java meta-classes into AAS elements and modify the information about available types reflected in the Types sub-model of the IIP-Ecosphere platform AAS. Akin, iip-aas implements basic forms of AAS-client, i.e., classes that conveniently wrap access to certain AAS parts such as operations or properties. Subclasses shall use or refine the basic functionality to implement concrete accessors, e.g., operation execution.

As far as possible, we aim for a transparent AAS integration. Therefore, platform code **must utilize the abstraction** for the aforementioned reasons. If AAS functionality is not available, new AAS concepts

[42] https://docs.oracle.com/javase/8/docs/api/java/util/ServiceLoader.html
[43] https://en.wikipedia.org/wiki/Adapter_pattern

become available or the underlying implementation changes significantly, a revised/extended AAS abstraction may be required, which, in turn, may require changes to existing platform code.

3.5.2 Network Support

In addition to the AAS abstraction, the `support` layer also provides basic network management functionality, in particular for TCP port negotiation. The network manager supports two modes, based on registered and dynamic/free ports. Both modes are relying on a self-selected key for the respective port, e.g., representing a service or a channel/topic identifier. Central services can register themselves with a platform-wide known key. Dynamic services are supported by assigning/reserving free (ephemeral) ports.

Network managers can be stacked, i.e., a parent network manager can contain (more) centrally registered addresses (e.g., for overarching communication brokers) while local managers focus on local (ephemeral) ports. The `NetworkManagerAas` realizes the active AAS frontend network manager instances, in particular for a central platform manager instance. The `NetworkManagerAasClient` implements an AAS-based access to the `NetworkManagerAas`, i.e., to allow implementing components to access a central network manager, also through stacking. It is important to note that not all components rather than installations may require a network manager. Further, not all network managers, in particular not local instances on (edge) resources must be exhibited through an AAS.

3.5.3 Lifecycle Support

A further basic capability is to start up components in a uniform but extensible manner. This is particularly important as individual components may rely on different technology imposing different technological requirements on the startup process. Moreover, it supports the transparent realization of optional and alternative platform components. Therefore, the Support Layer defines the `LifecycleDescriptor` (not shown in in Figure 21), allowing components to do the necessary startup/shutdown operations, declare a startup level (priority) and, if required, stop a component. A `LifecycleDescriptor` defines a priority (akin to startup levels in Linux) and may indicates, whether it desires to terminate the execution of the containing platform instance upon a certain event or condition. A `LifecycleDescriptor` announces itself through JSL and is taken up by the `LifecylceHandler`. The `LifecylceHandler` provides generic startup classes for all components, e.g., with or without the ability to terminate the platform instance, which trigger a respective processing of the lifecycle descriptors.

3.6 Transport and Connection Layer

The Transport and Connection Layer is responsible for connecting resources among each other, with a platform installation on a central IT or even with external cloud environments. We start off summarizing the requirements for the Transport Layer in Section 3.6.1. Then we will turn to the two interrelated components in this layer, the Transport Component (Section 3.6.2) and the Connectors Component (Section 3.6.3). Finally, in Section 3.6.4, we will discuss the realization of the requirements by the two components.

3.6.1 Requirements

In the requirements collection [8], the transport layer is particularly characterized by the requirements summarized in Table 5:

Table 5: Specific requirements from [8] for the Transport and Connection layer (not repeating the general requirements in Table 2 and Table 3).

Requirement	Summary
R13	Connectivity to other actors
R13a	Connectivity with I4.0 devices
R13b	Connectivity with I4.0 platforms
R13c	Connectivity with other IIP-Ecosphere platform instances
R14	Open and flexible connectors
R14a	At least OPC-UA and MQTT connectors
R14b	TCP-IP support
R14c	Bluetooth LE support
R14d	Integration of connectivity at runtime
R15	Connectors shall be as uniform as possible
R16	Integration of connectors shall be open and flexible
R17	Potential distribution of connectors to various devices
R17b	Management of connectors by platform
R17c	Connectors shall be parameterizable
R18	Securing connectors
R19	Use of a minimum set of internal data formats, examples mentioned in R19a, R19b
R19a	Example input formats (southbound)
R19b	Example input formats (northbound)
R19c	Restful APIs with JSON/XML
R19d	Example output formats (northbound)
R19e	Output data clocked in 5 second intervals
R19f	Data format conversion
R19g	Mechanisms to manipulate data
R20	Application-specific data paths (through the configuration model)
R21	Low impact on data throughput
R22	Platform data throughput of 500 Gbytes per year
R28	Machine pulse of 8 ms
R38	Appropriate authorization mechanism
R40	Role-based access control and TLS
R44	IDS-based connectors (including their security profile)
R49	Appropriate mechanisms to ensure data transfer and data sharing concerning principles of data protection such as legitimated purpose
R66	Pseudonymization and anonymization of data (transferred or shared)

#

Requirement R19c REST-APIs is not relevant for this layer as service and layer interfaces are expressed through AAS, which in the default implementation form an REST-API. However, JSON and XML mentioned in R19c may be potential wire formats for data transport. Further, as the Transport and Connection Layer shall support (complex) data types in generic manner, the examples mentioned in the explanations of R19a, R19b and R19d in [8] are covered generically. Regarding the general platform requirements in Table 2, in particular, R1, R2, R5, R8 and R10 apply to the Transport and Connection Layer. It is important to note that the realization of (sub-)requirements referring to the configuration model such as R17a or R20 will be discussed in Sections 3.11 and 4, but must be prepared already in the Transport and Connection layer.

With respect to the soft-realtime requirement R10, a stream-based data processing approach seems to be a feasible solution. This impression is supported by the successful application of such approaches in the Big Data domain [22] and the observation that several I4.0 platforms with (soft-)realtime promises typically rely on some form of stream-processing [27]. However, the streaming approach shall not impose unnecessary limitations to the data paths so that, e.g., (AI-)processors can operate with multiple in- and outputs or streams returning to the source (machine connector or underlying platform) can be realized.

3.6.2 Transport Component

At a glance, a Transport Component may appear to be superfluous if we build the platform on the capabilities of the AAS approach. We will outline our rationales first and then turn to the design of the Transport Component.

Initial experiments [29] with Docker containers and AAS (using BaSyx version as of July 2020[44]) on Raspberry Pi 3[45] as well as on Phoenix Contact PLCnext edge devices showed that the typical response time of operations without computational load is around 23 ms. In contrast, property value accesses can be executed at 4-10 ms. For comparison, plain Java Remote Method Invocations operate in this setup at 2-4 ms. Further investigations of this discrepancy indicated that utilized components from the BaSyx examples represent one operation as three internal REST calls, where two of them are required for managing internal functionality. Using lower level BaSyx components, e.g., through the `InvocablesCreator` and `ProtocolServiceBuilder` discussed in Section 3.5, operation calls can be executed at the same response time as property accesses, i.e., around 4-10 ms. This seems to be promising for R10 and, in particular, the 8 ms machine pulse mentioned in R28. However, the measured times are not suitable for (soft-)realtime processing or software-based stream-processing.

Consequently, we will use AAS primarily as control interface for the infrastructure layers and the platform services (R7). For cross-linking services in terms of data streams we will rely on a dedicated streaming approach, as to our very knowledge so far no concepts are provided for streamed data among multiple AAS[46]. In such a streaming setting, the integration of services (potentially implemented in different programming languages, R113a) could be done via AAS (not preferable as argued above) or through a specific streaming data interface. In the latter case, the service AAS is used to describe the streamed data types and the data connectors[31] but not for the actual streaming transport. While AAS may be preferable for uniformity (R7), specific streaming interfaces will allow for better performance (as we will detail in Section 3.6.2.3).

We structure this section into a brief review of related streaming approaches leading to some technology candidates, the design of the component and its (initial) technical validation.

3.6.2.1 Related Approaches

We discuss now approaches in the field of stream-processing, their relation to I4.0 and whether they could be useful for realizing the IIP-Ecosphere platform. As often certain protocols are required in I4.0 settings, we further discuss protocol realization candidates.

Regarding stream processing, we briefly review now related approaches, in particular stream processors (with own resource management approach) as well as stream processing libraries (focusing more on the stream-based transport). It is important to make this distinction, as the dynamic deployment and the adaptation capabilities foreseen for the IIP-Ecosphere platform shall integrate with rather than contradict automated management capabilities of the selected stream processing approach. From previous experience we know, that modifying a stream processor such as Apache Storm to introduce desired adaptation capabilities can be a tedious work that, if not accepted by the

[44] At least at that time, there were no version-based releases of BaSyx.

[45] Raspberry Pi is frequently used as IoT mockup device in literature and practice, e.g. for cost-effective showcases or even in experimental evaluations.

[46] AAS are designed to describe production assets along their lifetime rather than software components or even soft-realtime data services. Thus, this statement shall not be considered as critics rather than a potential limitation of AAS that has to be mitigated in IIP-Ecosphere by a different solution or additional technical means.

developing company, will not scale/evolve with future developments of the stream processor. Besides technical capabilities, important aspects are R13 (utilized transport protocols), R14 (openness and flexibility, in particular for connectors, transport protocols and formats) as well as availability and licenses (R5) of the individual approaches.

Table 6: Selected scientific streaming processing approaches (related to IoT)

Name	Mgt.	Based on	Protocols	Edge	Availability	License	Latest
EdgeWise [10]	Yes	Apache Storm		Pi3	Yes	APL2.0	2016
Frontier [23]	Yes		“WiFi”	Pi	-	?	?
[4]	Yes	Apache NiFi		Pi	-	?	?
Dart [3]	Yes		REST, JSON	Pi	-	?	?
PESP [14]	Yes	RabbitMQ			-	?	?
VISP [13]	Yes	Spring cloud stack, RabbitMQ	AMQP, MQTT		Yes	APL2.0	2018
Esc [28]	Yes				-	?	?

Table 6 compares related approaches that can be found in literature on stream processing for edge devices or IoT. Three of the approaches were evaluated on Raspberry Pi devices (mocking IoT devices, typically in rather different IoT application settings). For two of the approaches the implementation is available, and only [13] utilizes IoT protocols (as provided by the underlying technology). All of the approaches provide more or less (automated) management/deployment functionality. Although potentially interesting in an evaluation context, we see existing management functionality as a potential risk as they usually are designed for a certain setting and integrating, interfacing or in the extreme case replacing such functionality may lead to unforeseen complications.

Table 7 summarizes a set of recent stream processing frameworks and libraries that were identified through a web search (without specific focus on IoT/edge). Not included are commercial approaches like Grovestreams, Hazelcast, or Amazon Kinesis that were also part of the result set as R1, R5 and R6 in [8] express a clear direction towards Open Source.

Table 7 consists of two parts, the upper part for IoT related approaches and the lower part for generic stream processing approaches. In these approaches, the presence of management functionality correlates with the nature of a stream processor rather than a streaming library. Moreover, stream processors are usually realized as a complete stack, usually based on a cluster-based installation, e.g. with centralized servers such as Apache Zookeeper or a resource manager. Within the IT server installation of an IoT platform, such setups may appear to be feasible. However, we expect serious limitations for devices close to the production / OT[47] side. It is worth mentioning that some sources indicate that approaches like Apache Flink try to remove resource consuming central services like Apache Zookeeper and even consider the realization of specific versions for edge devices (e.g., a 2021 release of Flink shall include edge support). From a research perspective, e.g., in [10], such widely used stream processors are also criticized for their inflexible adherence to the One-Worker-Per-Operation-Architecture (OWPOA) as this design loses efficient processing scheduling opportunities by relying on the underlying operating system scheduler. A positive trend seems to be that more and more security features are built into the frameworks. Examples of such security features and a potential methodology to integrate them in a framework—especially from the early phases of framework

[47] As discussed in [8], **OT** refers to **O**peration **T**echnology, i.e., the control and monitoring of production machines, which typically operate under hard-realtime conditions. In contrast, **IT** (**I**nformation **T**echnology) such as the IIP-Ecosphere platform typically can only operate under soft-realtime constraints. Nowadays, edge devices may bridge OT and IT, e.g., in terms of separated, but integrated hard- and soft-realtime cores, potentially controlled by different operating systems/software.

design—is introduced in Section 3.9. This is in contrast to the initial tendency of, e.g. Apache Storm, to sacrifice security and encryption for throughput. Except for approaches dedicated to the IoT domain, full-stack frameworks typically realize own (usually fixed) transport protocols and, if at all, realize own internal interfacing concepts, e.g., based on REST-APIs. Thus, as far as we can see, (flexible, exchangeable) IoT-based transport protocols are rather uncommon and, typically, AAS is not considered for interfaces at all.

Table 7: Selected stream processing frameworks and libraries (=incubating, vispl = visual programming language)*

Name	Mgt.	Language	Protocols	Env	License	Latest	Comment
Apache Edgent	Yes	Java	MQTT, …	edge	APL2.0	2017*	Own language
Apache Streampipes	No	Java, Typescript	MQTT, OPCUA, ROS, …	IoT	APL2.0	2020*	ML support, Kafka, container, vispl
Benthos	No	Go	AMQP, MQTT		MIT	2020	Own language
Eclipse IoT Streamsheets	Yes	JavaScript	MQTT, OPCUA, …	IoT	EPL2.0	2020	Spreadsheets, Docker
Eclipse Kura	Yes	Java	MQTT, OPC-UA, …	IoT/edge	EPL1.0	2020	OSGi, Docker, vispl, stack
EdgeX Foundry	Yes	Go, C	MQTT, OPC-UA, …	IoT/edge	APL2.0	2020	REST, Stack
Flogo	Yes	Golang		IoT/edge, Cluster	BSD3-Clause	2019	TensorFlow, zero code
Sensorbee	No	Go		IoT	MIT	2017	ML integration, own language, documentation
Akka	Yes	Scala, Java	HTTP	Cluster	APL2.0	2020	Actors
Apache Apex	Yes	Java	Hadoop RPC	Cluster	APL2.0	2018	YARN
Apache Beam	Yes	Java, Python, Go		Flink, etc. cluster	APL2.0	2020	Requires processing cluster
Apache Flink	Yes	Java, Scala		Cluster	APL2.0	2020	Zookeeper, edge 2021?
Apache Flume	Yes	Java	Avro, protobuf	Cluster	APL2.0	2019	Zookeeper, big footprint
Apache Gearpump	Yes	Scala			APL2.0	2019*	Storm/Samoa compatible, YARN
Apache Kafka	Yes	Scala, Java		Cluster	APL2.0	2020	Zookeeper
Apache Kafka Streams	Yes	Java		Cluster	APL2.0	2020	Requires Kafka
Apache NiFi	Yes	Java		Cluster	APL2.0	2020	FlowFiles, REST, vispl

Name	Mgt.	Language	Protocols	Env	License	Latest	Comment
Apache Pulsar	Yes	Java		Cluster	APL2.0	2020	Brokers, Bookkeeper, Zookeeper
Apache Samza	Yes	Scala, Java		Cluster	APL2.0	2019	YARN, Kafka
Apache Spark	Yes	Scala	Spark RPC	Cluster	APL2.0	2020	shared memory
Apache Storm	Yes	Clojure, Java		Cluster	APL2.0	2020	Zookeeper
Hortonworks Streamline	No	Java, JavaScript	REST		APL2.0	2018	Documentation
Spring Cloud Stream	(No)	Java	open*		APL2.0	2019	Depends on Spring
StreamFlow	Yes	Java		Kafka, Storm cluster	APL2.0	2015	Vispl for Kafka, Storm
Streamtz	No	Python			Continuum	2020	Pandas, cuDF

Some of the approaches (Apache Streampipes, Sensorbee, Streamtz, Flogo) listed in Table 7 provide Machine Learning (ML) support/integration while other approaches already realize concepts for the integration of programs in multiple languages, e.g., Apache Storm and Apache Spark for Python. Approaches like Apache Streampipes, Eclipse Kura, Flogo, Apache NiFi, or Streamflow take up the trend towards visual programming, low code or even no code (as also identified for some I4.0 platforms in [27]). Although convenient, an integration of configuration modeling with existing visual programming languages for the orchestration of services may be a risky approach.

Summarizing these findings, only few approaches remain as candidates for the realization of streaming in the Transport Layer, which may involve edge devices, servers and even clouds. This (as well as license and normative considerations) requires a flexible selection of the transport protocol (R14) and the wire format. Moreover, the requirement of using AAS wherever possible (R7) or at least to allow for an exchange of the communication protocol excludes almost all stream processing frameworks (of course not from comparative evaluation settings). Needless to say that a candidate approach shall be able to handle synchronous (one input data item is related to one or no output item) and asynchronous (inputs and outputs can be decoupled) processing of data items as well as resources local (among local processes) and external network communication, e.g., for input and output.

As a result of these filter criteria, Apache Streampipes, Sensorbee, Hortonworks Streamline or Spring Cloud Stream appear to be feasible candidates. However, initial experiments indicated serious problems with Sensorbee and Hortonworks Streamline ranging from incomprehensible or non-existing documentation to problems when executing the respective examples. Apache Streampipes (APL2.0) ships with interesting features, several kinds of generic connectors, exchangeable transport protocol and wiring format although it is still considered to be in incubation state. As alternative we see Spring Cloud Stream[48] (APL 2.0), which allows exchanging the transport protocols for individual in/out-bound streams, supports user-defined payload wire formats, flexible exchange of communication protocols, network properties per data path among processors, implicit payload conversion (also through our serializer mechanism) and dynamic stream re-routing at runtime. Moreover, Spring Cloud Stream was successfully applied in [13].

[48] https://spring.io/projects/spring-cloud-stream

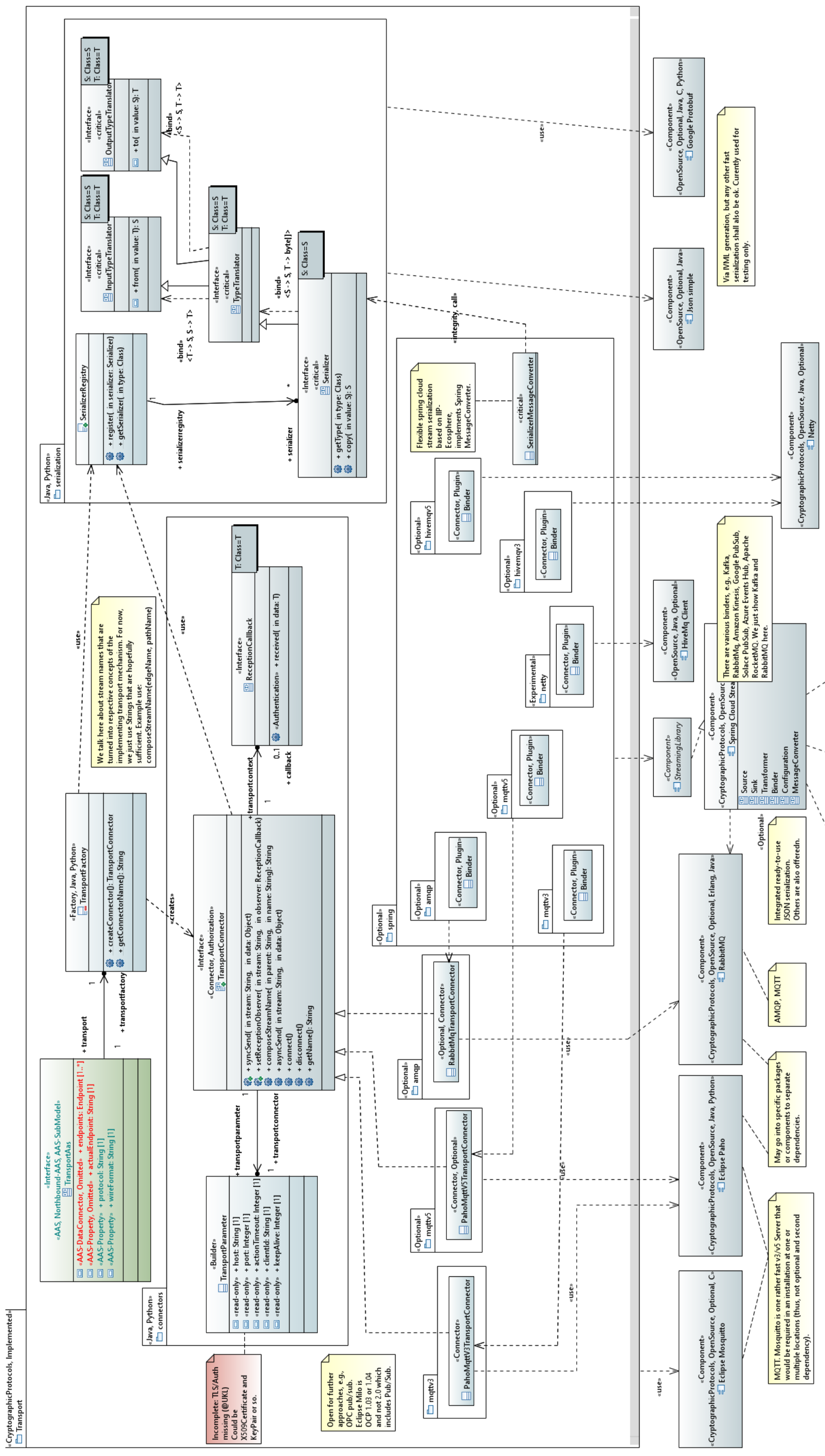

«CryptographicProtocols, Implemented» Transport
«Java, Python» serialization
SerializerRegistry
+ register(in serializer: Serializer)
+ getSerializer(in type: Class)
«Interface» «critical» InputTypeTranslator
+ from(in value: T): S
«Interface» «critical» OutputTypeTranslator
+ to(in value: S): T
«Interface» «critical» TypeTranslator
«Interface» «critical» Serializer
+ getType(in type: Class)
+ copy(in value: S): S
«bind» <T-> S, S-> T>
«bind» <S-> S, T-> byte[]>
«bind» <S-> S, T-> T>
+ serializerregistry
+ serializer
«Java, Python» connectors
«Interface» «AAS, Northbound-AAS, AAS-SubModel» TransportAas
«Factory, Java, Python» TransportFactory
+ createConnector(): TransportConnector
+ getConnectorName(): String
We talk here about stream names that are turned into respective concepts of the implementing transport mechanism. For now, we just use Strings that are hopefully sufficient. Example use: composeStreamName(edgeName, pathName)
«Interface» ReceptionCallback
+ received(in data: T)
«Interface» «Connector, Authorization» TransportConnector
+ syncSend(in stream: String, in data: Object)
+ setReceptionObserver(in stream: String, in observer: ReceptionCallback)
+ composeStreamName(in parent: String, in name: String): String
+ asyncSend(in stream: String, in data: Object)
+ connect()
+ disconnect()
+ getName(): String
«Builder» TransportParameter
+ host: String [1]
+ port: Integer [1]
+ actionTimeout: Integer [1]
+ clientId: String [1]
+ keepAlive: Integer [1]
Incomplete: TLS/Auth missing (@UKI) Could be X509Certificate and KeyPair or so.
Open for further approaches, e.g., OPC pub/sub. Eclipse Milo is OCP 1.03 or 1.04 and not 2.0 which includes Pub/Sub.
«Optional» spring
Flexible spring cloud stream serialization based on IIP-Ecosphere implements Spring MessageConverter.
«critical» SerializerMessageConverter
«Optional» hivemqv5
«Optional» hivemqv3
«Experimental» netty
«Optional» mqttv5
«Optional» amqp
mqttv3
«Connector, Plugin» Binder
«Optional, Connector» RabbitMqTransportConnector
«Connector, Optional» PahoMqttV5TransportConnector
«Connector» PahoMqttV3TransportConnector
«use»
«creates»
«integrity, call»
«Component» «OpenSource, Optional, Java, C, Python» Google Protobuf
«Component» «OpenSource, Optional, Java» Json simple
Via IVML generation, but any other fast serialization shall also be ok. Currently used for testing only.
«Component» «CryptographicProtocols, OpenSource, Java, Optional» Netty
«Component» «OpenSource, Java, Optional» HiveMq Client
«Component» StreamingLibrary
«Component» «CryptographicProtocols, OpenSource» Spring Cloud Stream
Source Sink Transformer Binder Configuration MessageConverter
There are various binders, e.g., Kafka, RabbitMq, Amazon Kinesis, Google PubSub, Solace PubSub, Azure Events Hub, Apache RocketMQ. We just show Kafka and RabbitMQ here.
«Component» «OpenSource, Optional» Apache Kafka
«Component» «OpenSource, Optional, Java» Jackson
«Component» «Java, OpenSource, Optional» Spring Framework
«Optional» Integrated ready-to-use JSON serialization. Others are also offeredn.
«Component» «CryptographicProtocols, OpenSource, Optional, Erlang, Java» RabbitMQ
AMQP, MQTT
«Component» «CryptographicProtocols, OpenSource, Java, Python» Eclipse Paho
May go into specific packages or components to separate dependencies.
«Component» «CryptographicProtocols, OpenSource, Optional, C» Eclipse Mosquitto
MQTT. Mosquitto is one rather fast v3/v5 Server that would be required in an installation at one or multiple locations (thus, not optional and second dependency).

Figure 22: Transport Component overview

However, it is important that we want to integrate the streaming approach with the connectors and the ML processors in a model-based manner similar to our work for Apache Storm in [6], here with an even stricter focus on isolating the utilized streaming approach. We believe that relying on glue code generation allows us to replace (within limits, e.g., always assuming a data flow graph with some kind of source, processor and sink) the stream approach. Thus, we see Spring Cloud Stream as a good and justified initial candidate for the reasons stated above as well as because it ships with several transport protocols including protocols for public clouds (although it also relies on a large dependency tree particularly induced by the Spring Framework[49]). In later stages of the project, we may take Apache Streampipes or an edge-enabled version of Apache Flink into account.

Regarding IoT protocols, several implementations are available, in particular from different projects of the Eclipse.IoT[50] ecosystem (provided under compatible licenses for IIP-Ecosphere). While some projects focus on specific protocols, e.g., Eclipse Paho[51] on MQTT, others already integrate various protocols such as Eclipse Hono[52]. Although such integrations may be an interesting foundation, they often rely on specific assumptions, e.g., Eclipse Hono collects binary payload from different protocols and forwards the payload to a fixed default protocol (MQTT). While such approaches may provide access to different low-level protocols or machine connectors (cf. Section 3.7), they may also introduce limitations due to their design choices or do not provide mechanisms for turning such generic implementations into application specific solutions, e.g., through application-specific data translators. Moreover, some transport protocols are currently not applicable, e.g., we currently do not consider OPC UA PubSub due to a lack of feasible implementations, where Eclipse Milo[53] currently does not support the required OPC UA version.

3.6.2.2 Design

Figure 22 depicts an overview of the packages and (top-level) classes in the Transport component. The Transport component is intended to be deployable as re-usable component rather than to act as a standalone communication container. The main concepts in this layer are:

- The `TransportConnector` allowing to bind transport protocols into the infrastructure. A transport connector allows sending/receiving of data on (virtual) channels. As receiving usually happens in asynchronous manner, implementations that rely on a `TransportConnector` are informed via the `ReceptionCallback` about received data.
- The actual wire format to be used for transport may differ from protocol to protocol. For example, low level transport protocols such as MQTT or AMQP support arbitrary binary payloads (might be with individual size restrictions) while higher level protocols such as OPC pub/sub define their own payload format. However, to be open and flexible with respect to the wire format and to utilize a minimum of data formats within the platform (R19), we foresee a mechanism for data transcoding. For performance reasons, data transcoding shall happen only when actually needed. Specifically for binary wire formats, the `Serializer` transcodes programming language objects into a binary representation and back. More generically, a `Serializer` is a `TypeTranslator` that can be applied also in other situations, e.g., data processing. In turn, `TypeTranslator` is a combination of `InputTypeTranslator` and

[49] Native executables are in experimental development and may help optimizing the deployment/performance: https://www.heise.de/news/Java-Framework-Native-Spring-Anwendungen-laufen-ohne-die-JVM-5078681.html
[50] https://iot.eclipse.org/
[51] https://projects.eclipse.org/projects/iot.paho
[52] https://projects.eclipse.org/projects/iot.hono
[53] https://projects.eclipse.org/projects/iot.milo

`OutputTypeTranslator` with cross-over template bindings[54]. Intentionally, we leave the actual technical approaches for transcoding open here (some candidates are JSON, OPC-JSON or protobuf[55]). The actual instances depend on the data types used in the application and are supposed to be generated from the configuration model. While instances of `TypeTranslator` are supposed to be attached where needed (and may be combined with `Serializer` instances), `Serializer` instances shall be usable dynamically on-demand, e.g., for a certain `TransportConnector` implementation. For this purpose, we provide a `SerializerRegistry`.

- The `TransportConnector` instances shall be available to other components of the platform where an internal data protocol is needed. To obtain `TransportConnector` instances, we define a `TransportFactory` and exhibit the actual protocol, the wire format and the broker data connector(s) from the platform configuration in the `Transport` AAS.
- Three default protocol plugins are shipped with the IIP-Ecosphere platform, namely MQTT v3 (based on Eclipse Paho), MQTT v5 (also Eclipse Paho) as well as AMQP (based on the RabbitMQ AMQP client). Each protocol plugin is an own alternative component, the installed ones determine the `TransportFactory` behavior through a JLS descriptor.
- The streaming approach is already relevant to the Transport Layer as transport protocols and wire formats must be provided accordingly. However, as discussed above, the streaming approach shall also remain exchangeable through glue code generation. Thus, the platform provides also transport plugins for the default streaming approach (Spring Cloud Stream), the so called `Binders`, which are realized in turn through the Transport Component. A basic `spring` component implements convenient mechanisms for applying Spring Cloud Stream in IIP-Ecosphere, e.g., to add serializers to the `SerializerFactory` through the component setup (in Spring `application.yml`, to be instantiated from the platform configuration) or to bind the `SeralizerFactory` to the data conversion mechanism of Spring Cloud Stream (`SerializerMessageConverter`). In addition, Spring Cloud Stream ships with generic serialization approaches, e.g., for JSON or XML that may be used out-of-the-box. By default, the IIP-Ecosphere platform ships with five alternative Spring Cloud Stream protocol binders for MQTT v3 (based on Eclipse Paho and HiveMQ-client), MQTT v5 (based on Eclipse Paho and HiveMQ-client) and AMQP (based on the RabbitMQ AMQP client). Alternative binders for the same protocol are mainly provided for performance comparison.

It is important to mention that further protocol binders are available for Spring Cloud Stream, e.g., for RabbitMq, Amazon Kinesis, Google PubSub, Solace PubSub, Azure Events Hub, Apache RocketMQ. These binders may be helpful for realizing Cloud integrations, e.g., in the Security and Data Protection Layer. However, for uniform usage of protocols within the platform, a respective `TransportConnector` shall be provided (the AMQP connector may already be used for RabbitMq). Furthermore, it is important to mention that we do not prescribe the amount or deployment strategy for communication servers (Brokers for the mentioned concrete protocols) within a platform installation. The platform configuration shall provide opportunities to define multiple brokers (to be reflected in the `Transport` AAS) while the broker(s) to be used shall be instantiated through the platform configuration or the network managers into the respective deployment units. Moreover, based on the provided mechanisms of the protocol implementations and the streaming library,

[54] At a glance, `TypeTranslator` shall be sufficient, but in some situations it is convenient that only the required direction must be implemented rather than both. This is in particular true for the machine/platform connectors, which require either direction for different types but usually not both directions. As `TypeTranslator` inherits from the input/output type translators, it is also possible to use a fully-fledged `TypeTranslator` in these situations.
[55] https://developers.google.com/protocol-buffers

different levels of resilience or recovery can be realized, while failover to alternative broker servers may require additional implementation work.

3.6.2.3 Validation and Evaluation

We discuss now briefly the validation of the design and the implementation of the Transport Component as it has a major impact on the performance of the entire platform. We start off with a discussion of the regression testing approach and turn then to an initial performance evaluation.

The implementation of the Transport Component is subject to **regression testing** and continuous integration. Testing protocol integrations requires some form of server or broker instance. Therefore, further Open Source components are utilized so that the tests are self-contained, e.g., embeddable protocol brokers to simulate the platform side in the respective tests. The required dependencies are only active in testing, i.e., they are not part of a platform installation and, thus, here relaxed license or Java version rules may apply if needed. In the regression tests, we use protobuf and a simple JSON implementation for serialization as well as Apache HiveMq or Moquette as MQTT broker and the Apache Qpid broker as AMQP broker.

For the Spring Cloud Stream binders we realized a simple setup validating the envisioned streaming capabilities mentioned in Section 3.6.2.1. This is reflected in the communication setup shown in Figure 23. Ingested by a `Source` (the regression test), a mocked stream component (`Transformer`) modifies the data (synchronously) and passes the data to the broker (representing the IIP-Ecosphere platform/server). The communication between these instances is handled by the `Protocol Binder` under test as well as the `Serializer` selected by the test. The `Protocol Binder` is based on the respective protocol implementation and in the test bound against a corresponding embedded test server/broker. To test also the flow back, a shortcut client based on a corresponding `TransportConnector` receives the data and ingests modified data asynchronously, which now flows through the `Broker`, the `Serializer` and the `Protocol Binder` back to the `Source` acting also as `Receiver`. Combining `Source` and `Receiver` is a relevant setup, as a machine/platform connector (to be discussed in Section 3.6.3) also ingests data and may receive information, e.g., to reconfigure an edge device or a machine. The regression test has access to the sent/received information and, thus, can validate the entire flow.

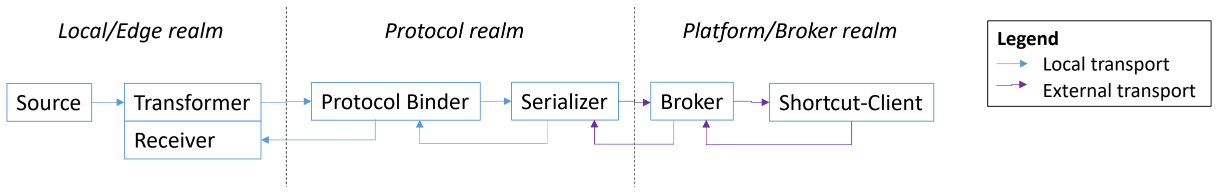


Figure 23: Regression testing data flow for the Transport Component.

In addition, it is also important to understand the (early) fulfillment of **quality requirements**. We determine the respective properties in terms of a **performance experiment**. Figure 24 details the setup of this experiment, which in fact is a variant of the regression test setup. Here, the `Source` produces a stream of data items at a certain ingestion frequency. Each data item consists of at least 50 values with repeatable characteristics (R19a). We concentrate on the payload and scope out meta-information (R79) for now. A simple `Anonymizer` takes a produced data item and turns one property (a name String) into simple pseudonyms. An "`AI-Service`" inspects the data and sends for 5 received data items one "command" back to the `Source`. Again, on the forward flow, the processors operate synchronously, while the backward "command" flow is ingested asynchronously. The number of received data items is recorded in all processors by simple monitoring probes and written in parallel once per second to a log file. An additional stream is used to asynchronously send experiment control commands to all involved processors, e.g., to terminate the experiment and to close the monitoring log. Items on the experiment control stream are not recorded by the probes.

The processors in Figure 24 can be executed locally (in one process, in multiple processes) or distributed on separated hosts as indicated in Figure 24. For the distributed execution, two brokers are used, one in the local realm and a remote broker in the platform realm. In the local realm, we currently use the same transport protocol/mechanism as in the platform realm, i.e., we focus at the moment on an Inter-Process Communication (IPC) setup rather than an edge setup where at least one stream goes to a different resource or the platform. Replacing the transport protocol, using different brokers or exchanging the wire format for serialization may be subject to future experiments. In this experiment we focus on the basic transport characteristics of the utilized approach.

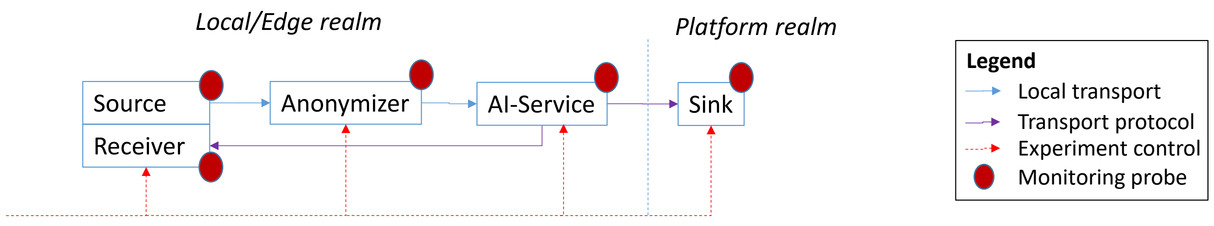


Figure 24: Performance testing data flow for the Transport Component.

For executing the experiment, we use a selection of the binders available in the platform (HiveMq v3, v5 with QoS AT_LEAST_ONCE, AMQP) with the setup as shown in Figure 24 and a respective (local, embedded) broker (Apache HiveMQ 2020.4, Apache Qpid 8.0.2). As baseline, we realized a plain network communication binder/distributed broker based on Netty[56], an asynchronous networking library, and the network port management of the platform. For the source, we use a message ingestion rate[57] per experiment and vary it from slow pace (R28) up to congestion. As wire format, we use a simple JSON serialization (leading to 650 Bytes of payload). We run the experiment for 1 minute and exclude by default the first three seconds as well as the last second where fluctuations due to network, just-in-time compilation and broker startup activities may occur. Further, some time may elapse until the average throughput is established, which we consider in this experiment as part of the stable measurements although it may significantly cause variations.

The measurements for this initial experiment have been taken on an Intel Core 7-8750U @ 1.90GHz with 32 Gbyte running Windows 10 and OpenJDK 13+33. As we aim at the moment for initial measures, we do not pay specific attention to a clean setup, e.g., getting rid of potentially other process influences such as a virus scanner or system updates.

[56] https://netty.io/

[57] The ingestion is based on the Spring Default Poller, which is controlled by a fixed delay between message ingestion time slots (translates to a minimum ingestion rate) and a maximum number of messages ingested within a slot (determines a maximum ingestion rate). The effective ingestion rate is within the minimum and maximum ingestion, but subject to an internal congestion control of Spring Cloud Stream.

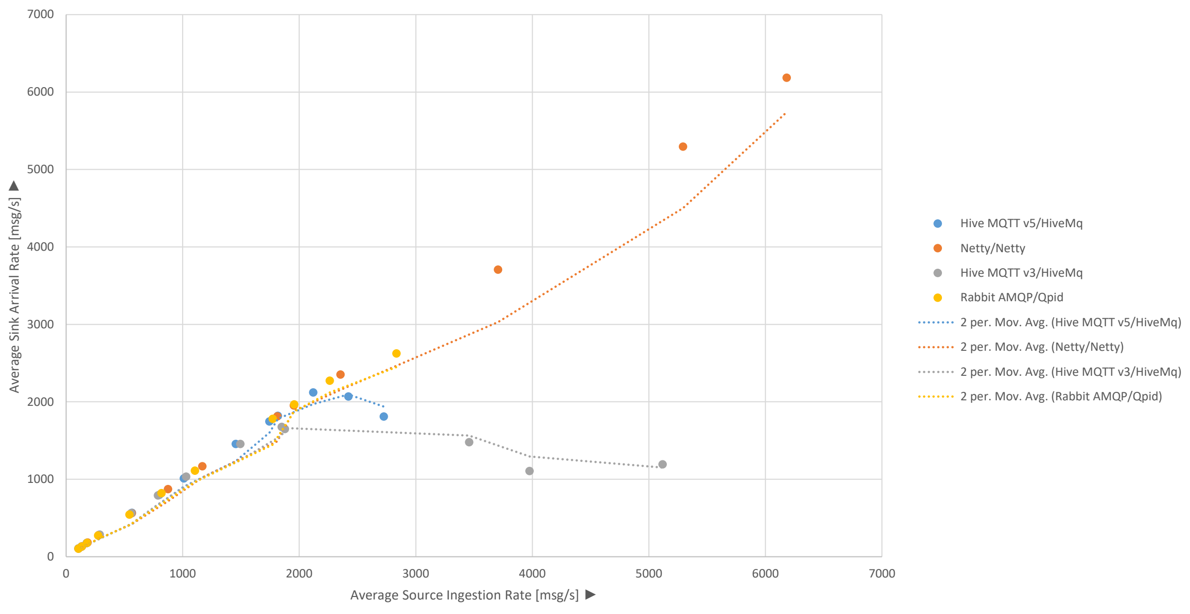


Figure 25: Average stream throughput measures for the four utilized alternative binders with trendlines.

Figure 25 illustrates the average ingestion rate at the source on the horizontal axis and the average arrival rate at the sink on the vertical axis. Until an ingestion rate of around 1000 messages per second, all binders scale similarly. Over 1000 messages per second, the behavior of the four binders differ significantly. The arrival rate of the MQTT v3 binder starts dissociating from the ingestion rate at around 1500 messages per second. For MQTT v5 this happens at around 2100 messages per second and for AMQP at a rate of roughly 2300 messages per second. While the MQTT v3 binder tries to cope with the ingestion rate until 6500 messages per second (dropping at the sink to 1400 messages per second), the MQTT v3 and the AMQP binders stop operating around 2700 messages per second. In contrast, the experimental `Netty` binder scales well until 7200 messages per second. Then the sink rate starts dissociating from the ingestion rate and above 9300 messages per second the simple experimental broker implementation stops operating as indicated by the trendline in Figure 25. Moreover, there are noticeable differences in settling time for the average throughput (not shown in Figure 25): All binders require more than 10 seconds to reach the respective average throughput, while `Netty` requires higher settling times for lower ingestion rates and AMQP leads faster to a stable throughput than both MQTT versions.

As Figure 25 relates source and sink throughput rates, it does not reflect the total number of translated messages. Due to the streaming setup, the messages among source, processors and sink and also messages on the "command" channel (one item per five input messages) are communicated. Thus, the absolute number of transmitted messages per second is higher (least around factor 3.2). Table 8 details these numbers for the measured protocol-client-server combinations. In particular, our HiveMq readings amount to similar ranges as reported in [20], where two server machines with up to 16 CPU cores but no stream processing approach were used.

Table 8: Total number of translated messages per second in best source/sink transmission situation.

Total number of translated messages per second	
MQTT v3: HiveMq, HiveMq embedded server	6172
MQTT v5: HiveMq, HiveMq embedded server	8908
AMQP: Rabbit MQ client, Qpid embedded server	9531
NETTY	30298

In summary, the required rate of 125 messages at 8 ms machine pace (R28) is supported by all brokers and works in combination with the Spring Cloud streaming approach. At around 50 values per message (650 Bytes of payload in a JSON serialization), a stable ingestion of 1000 messages per second leads to (calculated) 2.1 GByte of data transmission per hour. Moreover, the `Netty` binder can cope with (calculated) 15.6 GByte of data, which even qualifies for R91[58]. It is important to emphasize that we focus here on pure IPC transport characteristics without significant data processing load. Moreover, we use a single stream, i.e., multiple (moderate) input streams from different edge devices may easily aggregate to even higher frequencies and volumes. In a realistic setting, we expect a multi-server setup as platform installation and potentially also a redundant cluster-based message handling for individual tasks, e.g., in the data integration, so that the envisioned approach qualifies for the given data (transport) quality requirements, in particular frequency and volume.

Further experiments indicate that the discussed behavior is similar when running the data processing within a single JVM, i.e., as threads, or in separate processes. Measurements on real edge devices with inter-device (cross-realm) network communications are subject to future work. As soon as further parts of the platform are available that potentially impact the data size or the performance (meta-information, security, etc.), further experiments shall be performed.

3.6.3 Connectors Component

The Connectors Component is responsible for the communication with already installed platforms (the virtual platform aspect) or machines (potentially connected via some form of edge devices). The aim here is to allow for a bi-directional, typed communication represented in terms of connector instances. Relying on the design of the Transport Component, it is desirable that the machine/platform connectors utilize type translators or serializers for the inbound communication, i.e., to translate received information (if feasible already filtered in application-specific manner) into application-specific datatypes that can further be processed in the IIP-Ecosphere platform. For the outbound direction, (AI-)services or humans may make decisions about changes in the connected machines/platforms. These decisions are represented as information, e.g., commands, and are translated/sent through the connector to the machine or platform. Here, type translators shall turn the application-specific data types received from the platform side into information suitable for the external side. As stated in Section 3.6.2, application-specific type translators shall be realized by code generation to ease the development of applications.

The connectors discussed in this section may be utilized within the realm of the same factory, i.e., they may run at reduced or even no security measures. The connectors may also link to external realms, e.g., via the internet. In this case, adequate encryption mechanisms may apply or even the machine/platform connectors may have to be extended through IDS functionality.

3.6.3.1 Related approaches

Regarding IoT protocols, several implementations are available, in particular from various projects of the Eclipse.IoT ecosystem. Some projects focus on specific protocols, e.g., among others Eclipse Paho on MQTT, Eclipse Milo on OPC UA, Eclipse Californium[59] on CoAP, Eclipse Leshan[60] on LWM2M, or Eclipse Tahu[61] on legacy SCADA/DCS/ICS protocols. Other projects already integrate various protocols such as Eclipse Hono, Eclipse Agail[62], Eclipse Kapua[63] with a cloud focus based on MQTT transport or

[58] Based on the transferred messages in Table 8, this leads to 13.5 GBytes up to 66 GBytes per hour.
[59] https://projects.eclipse.org/projects/iot.californium
[60] https://projects.eclipse.org/projects/iot.leshan
[61] https://projects.eclipse.org/projects/iot.tahu
[62] https://projects.eclipse.org/projects/iot.agail
[63] https://www.eclipse.org/kapua/

Eclipse Ponte[64] for mapping IoT protocols into REST. Although such integrations may be an interesting basis for our work, they already realize concepts and ideas that do not fully comply with the requirements of the IIP-Ecosphere platform. For example, Eclipse Hono collects binary payload from different protocols and forwards the payload to a default protocol (MQTT) without options for filtering/translating the payload or for supporting alternative protocols. Similarly, Eclipse Ponte focuses on REST for internal communication, which from our point of view is just one potential alternative. Further, Eclipse Agail emphasizes the cloud aspect but neglects local resources or edge devices. While these and similar approaches may ease the access to different low-level protocols or machine connectors (cf. Section 3.7), they usually do not provide mechanisms to enable core IIP-Ecosphere functionalities such as data filtering or translation. Moreover, if they implement a stack or multiple integrated layers, they usually do not offer AAS functionality (R7).

While it makes sense to review these approaches to find a feasible abstraction as well as to consider existing abstractions and protocol implementations as those mentioned above, it is not productive just to implement connectors to achieve a high number of protocols (this is one of the strengths of existing I4.0 platforms as shown in [27]). It is more important to develop and evaluate concepts to enable openness and extensibility for inbound/outbound directions. From a resource perspective, it is important to realize connector types for the actual needs of the stakeholders (R14a states MQTT and OPC UA). Moreover, one goal is to demonstrate how model-based generation can turn such generic connectors into efficient and application-specific mechanisms already at the bottommost layer of a platform.

3.6.3.2 Design

For the design of this component, it is important to recall that in contrast to the Transport Component, the Connectors Component already deals with processing and translating application-specific data. For example, it is not performant to just ingest, e.g., an entire OPC UA namespace upon each data modification or, if polling/sampling shall be applied, in each poll cycle. It is more important to select the required data in an application-specific manner and to focus on the information that is required by an application running on the platform. We call the step of translating an outbound protocol into an internal protocol (and back) "protocol adaptation", i.e., a (generated) plug-in `ProtocolAdapter` will be responsible for this task. One form of implementing the protocol adaptation is in terms of existing `TypeTranslator` and `Serializer` instances from the Transport Component, either as the realizations are part of the platform and can be re-used or because they are defined as part of the application and can be generated or are provided as hand-crafted components. However, also other forms of type translation may occur. This applies to connectors that handle generic payload (where the payload format must be translated to application-specific instances and can optionally be filtered/translated). Further, it applies to connectors that are based on a specific information model, such as OPC UA or AAS. In the latter case, we aim for specific `TypeTranslator` instances that are linked to a generic model interface abstracting over the underlying information model. However, not all approaches support the same range of concepts and types, e.g., OPC UA allows different kinds of custom datatypes while AAS does not. Thus, connectors will differ in the offered functionality of such an interface and it may be helpful to provide meta-information stating the connector capabilities in order to dynamically guideline the code generation for a certain connector.

[64] https://projects.eclipse.org/projects/iot.ponte

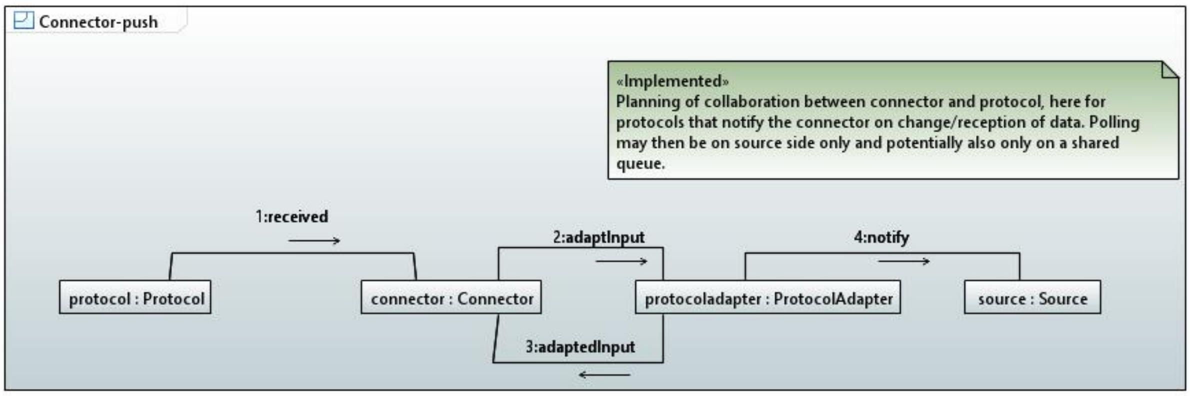

Connector-push
«Implemented»
Planning of collaboration between connector and protocol, here for protocols that notify the connector on change/reception of data. Polling may then be on source side only and potentially also only on a shared queue.
1:received
2:adaptInput
4:notify
3:adaptedInput
protocol : Protocol
connector : Connector
protocoladapter : ProtocolAdapter
source : Source

Figure 26: Event-based connector and push-based protocol-adaptation.

Moreover, connectors may differ in their data provisioning style. For performance reasons it is desirable to utilize **event-based ingestion**, i.e., the underlying protocol or information model informs the connector about new or changed data. Message passing approaches like MQTT or information-model based approaches like OPC UA provide such events. In this case, as illustrated in Figure 26, the "`Protocol`" notifies the `Connector` about new data. In turn, the `Connector` consults the `ProtocolAdapter` to translate the external data into an application-specific type, which, dependent on the "`Protocol`" capabilities, can be done in terms of payload translation or by querying the abstracted model of the "`Protocol`" (not shown in Figure 26). When the data is translated, the respective instance is passed on to a registered streaming `Source` in asynchronous manner. For the outbound direction (not shown in Figure 26), the `Source` ultimately receives the data as a stream and calls the `Connector` upon a received data item, which then consults the `ProtocolAdapter` in the backward direction ultimately leading to a send/write command on `Protocol`.

Protocol implementations not offering such notifications are subject to **polling**. One example here is the current BaSyx implementation of AAS. In the version that we currently use, no events are provided (BaSyx plans for events earliest end of 2020[65]). As illustrated in Figure 27, the IIP-Ecosphere `Connector` then actively (based on connector settings) polls information from the "`Protocol`". As before, the `Connector` consults the `ProtocolAdapter` and notifies the registered `Source` about the data to be ingested. The outbound direction works as discussed for event-based ingestion.

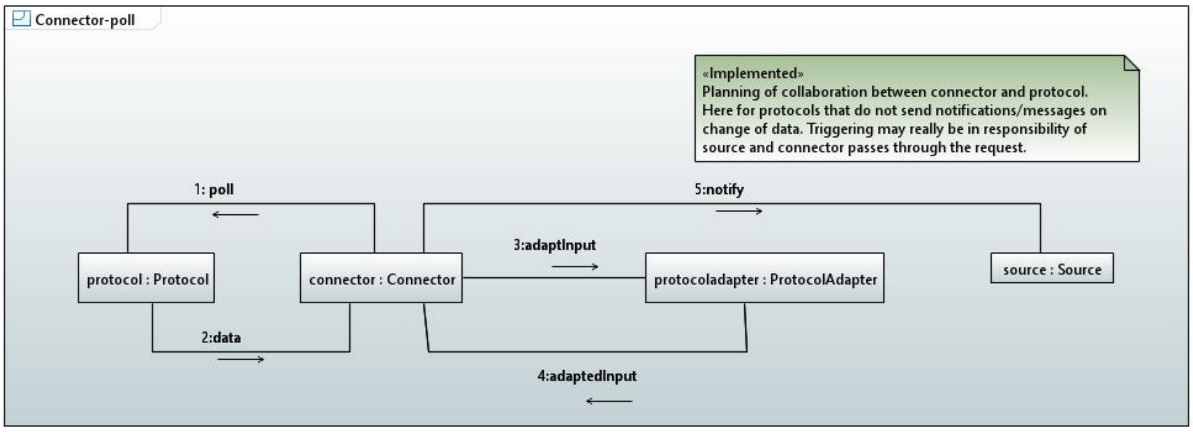

Connector-poll
«Implemented»
Planning of collaboration between connector and protocol. Here for protocols that do not send notifications/messages on change of data. Triggering may really be in responsibility of source and connector passes through the request.
1: poll
5:notify
3:adaptInput
2:data
4:adaptedInput
protocol : Protocol
connector : Connector
protocoladapter : ProtocolAdapter
source : Source

Figure 27: Poll-based connector and subsequent protocol adaptation.

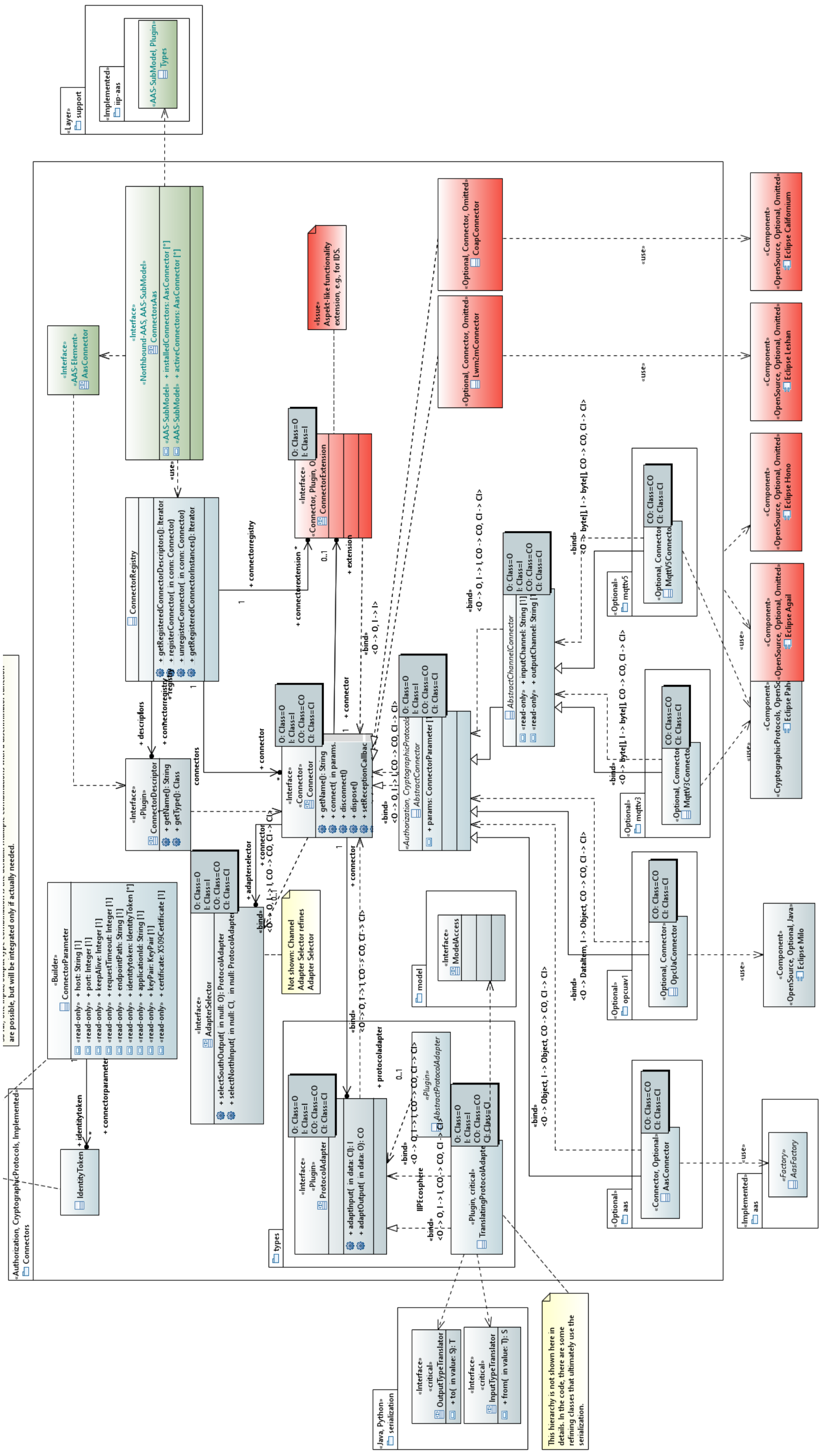
[65] If the required notifications are available, the AAS machine/platform connector can be extended to support event-based ingestion.

Realizing the polling cycle in the Connector rather than the Source allows for connector-specific polling strategies as well as for a uniform interface towards the stream-based data processing in the IIP-Ecosphere platform.

While event-based injection and polling may appear to be an alternative choice, a Connector may, if feasible, implement both alternatives and let the user (via the setup/platform configuration) decide about the desired approach. In particular, connectors for protocols based on information models may support both forms (such as OPC UA).

Figure 28 presents an overview of the main classes in the Connectors Component of the IIP-Ecosphere platform. The component consists of:

- The Connector interface in the middle of the figure representing a platform/machine connector. Connectors based on an information model shall exhibit a ModelAccess instance to interact with the information model. The Connector interface defines four template parameters, consisting of the data types accessible from the platform, i.e., CI for input into the connector and CO for output produced by the connector, and the data types for the handshake with the underlying protocol implementation, i.e., I for input into the protocol and O for output issued by the protocol. A Connector can be connected as specified in the ConnectorParameters and security settings like IdentityToken or certificates. When connected, received data of type O is passed through a ProtocolAdapter and an interested party is informed through a ReceptionCallback (from the Transport Component) in terms of a data object of type CO. Via the write method, data of type CI can be passed in, is translated by the ProtocolAdapter and handed as an instance of I to the underlying protocol. Finally, a Connector can be disconnected or, ultimately, disposed. So far, we plan for a single distinct pair of input/output types. If heterogeneous types shall be covered, we see two alternatives: 1) Mapping the alternative types as alternatives into an umbrella type. 2) Using a discriminator in terms of the AdapterSelector.



«Layer» support
«Implemented» iip-aas
«AAS-SubModel, Plugin» Types
«Authorization, CryptographicProtocols, Implemented» Connectors
«Interface» «AAS-Element» AasConnector
«Interface» «Northbound-AAS, AAS-SubModel» ConnectorsAas
ConnectorRegistry
«Interface» «Plugin» ConnectorDescriptor
«Interface» «Connector» Connector
«Interface» AdapterSelector
«Builder» ConnectorParameter
IdentityToken
«Issue» Aspect-like functionality extension, e.g., for IDS.
«Interface» «Connector, Plugin, Optional» ConnectorExtension
«Authorization, CryptographicProtocols» AbstractConnector
AbstractChannelConnector
«Optional, Connector, Omitted» CoapConnector
«Optional, Connector, Omitted» Lwm2mConnector
«Optional» mqttv5
«Optional, Connector» MqttV5Connector
«Optional» mqttv3
«Optional, Connector» MqttV3Connector
«Optional» opcuav1
«Optional, Connector» OpcUaConnector
«Optional» aas
«Connector, Optional» AasConnector
model
«Interface» ModelAccess
types
«Interface» «Plugin» ProtocolAdapter
«Plugin» AbstractProtocolAdapter
«Plugin, critical» TranslatingProtocolAdapter
IIPEcosphere
«Java, Python» serialization
«Interface» «critical» OutputTypeTranslator
«Interface» «critical» InputTypeTranslator
Not shown: Channel Adapter Selector refines Adapter Selector
This hierarchy is not shown here in details. In the code, there are some refining classes that ultimately use the serialization.
are possible, but will be integrated only if actually needed.
«Component» «OpenSource, Optional, Omitted» Eclipse Californium
«Component» «OpenSource, Optional, Omitted» Eclipse Leshan
«Component» «OpenSource, Optional, Omitted» Eclipse Hono
«Component» «OpenSource, Optional, Omitted» Eclipse Agail
«Component» «CryptographicProtocols, OpenSource» Eclipse Paho
«Component» «OpenSource, Optional, Java» Eclipse Milo
«Implemented» aas
«Factory» AasFactory
«use»
«bind»

Figure 28: Overview of the Connectors Component.

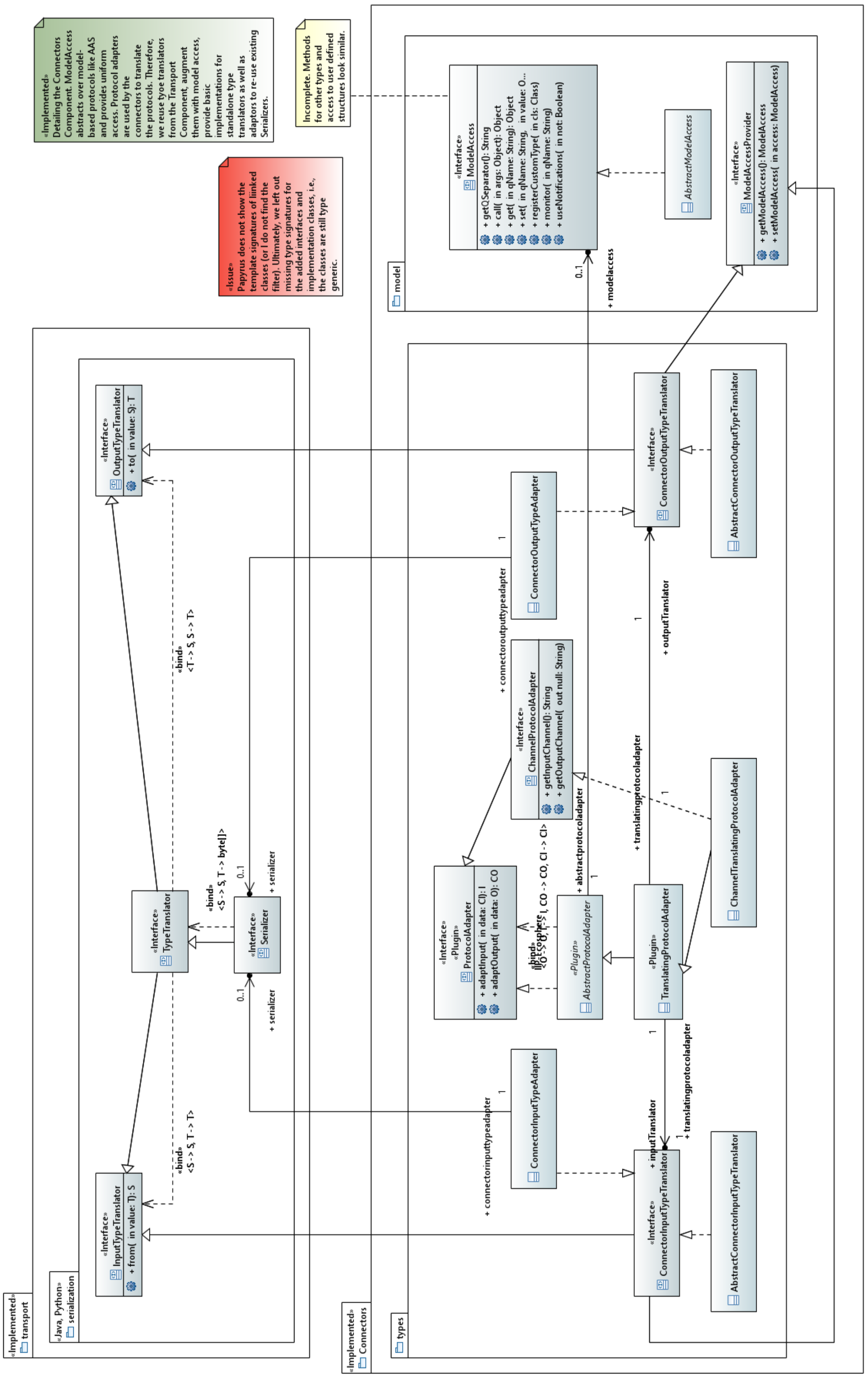
- The `TranslatingProtocolAdapter` is a default implementation of the `ProtocolAdapter` and relies on type translators, i.e. `InputTypeTranslator` and `OutputTypeTranslator` defined by the Transport Component. The `ProtocolAdapter` and its related classes will be detailed below. In particular protocol adapters to information models have a relation to a `ModelAccess` instance, which allows the type translation to interact with the model.
- The `AbstractConnector` provides a basic implementation, e.g., for handling the `ReceptionCallback`, for utilizing the `ProtocolAdapter`, etc. leaving just methods open that are protocol specific. The `AbstractChannelConnector` specializes the `AbstractConnector` for channel-based protocols such as MQTT and, in turn, requires a specialized protocol adapter (as we will detail below).
- The `ConnectorExtension` may add additional capabilities to a connector, e.g., IDS support. The IDS reference architecture model introduces the concept of Trusted Connector. Such a connector extends the security features of Base Connector. An IDS connector generally focuses on security and delivers a trusted platform, incorporating several mechanisms such as identity and trust management for authentication, trustworthy communication based on encrypted connections. Instances of trusted connector allow the remote integrity verification to ensure the integrity of the deployed software before granting corresponding access to data. Such connectors guarantee a controlled execution environment for data services [16].
- The `ConnectorRegistry` collects information about installed and used connectors. Installed connectors are registered through an instance of `ConnectorDescriptor` upon infrastructure startup (in Java through JSL) with the `ConnectorRegistry`.
- The information provided by the `ConnectorRegistry` is also the basic information to be presented in the AAS of the Connectors Component. Further, selected capabilities of the connectors are made available through the `installedConnectors` sub-model of the platform AAS. Created connector instances register themselves upon connect/disconnect with the `ConnectorRegistry`, which in turn leads to an update of the `activeConnectors` sub-model, i.e., connected connectors appear as sub-model elements and disconnected connectors are flagged as inactive[66]. Further, connector instances provide access to their input/output data types by referencing to the respected sub-model elements in the `Types` sub-model (see Section 3.5). Ultimately, connector instances link to their descriptors in the `installedConnectors` sub-model to indicate their origin and capabilities.

Currently, four specific (optional) connectors are realized in terms of individual components extending the Connectors Component. These are the generic `AasConnector` for integrating external AAS into the platform (based on the `AasFactory` from the Support Layer[67]), the `OpcUaConnector` for OPC UA 1.04 (based on Eclipse Milo) as well as two payload-based MQTT connectors, one for MQTT v3 and one for MQTT v5, also based on Eclipse Paho akin to the Transport Component. Each of these protocols bind the known template parameters of `Connector` as needed, all leaving `CI` and `CO` unbound as these are application-specific types to be defined when instantiating the respective connector (and providing a matching `ProtocolAdapter`). These approaches/protocols have been selected due to the required mandatory support for BaSyx (R7) as well as R14a explicitly mentioning OPC UA and MQTT. All connectors are based on TCP/IP networking (R14b). However, R14b does not clarify the protocol and the wire format, but some TPC/IP based protocol can be realized using the structures

[66] So far it seems that no elements can be removed at runtime from an AAS, potentially to not render references among them illegal.

[67] While BaSyx is the default implementation for IIP-Ecosphere, this connector provides the possibility to define the individual instance to be used, i.e., individual instances for specific connections may use other factory instances than the default one.

defined in the Transport and the Connectors Component. Similarly, Bluetooth LE was mentioned in R14c without further details. As the Connectors Component is designed to be open, further connectors can be added easily.


«Implemented» transport
«Java, Python» serialization
«Interface» InputTypeTranslator
+ from(in value: T): S
«Interface» TypeTranslator
«Interface» OutputTypeTranslator
+ to(in value: S): T
«bind» <S -> S, T -> T>
«bind» <T -> S, S -> T>
«bind» <S -> S, T -> byte[]>
«Interface» Serializer
0..1
+ serializer
«Implemented» Connectors
types
model
«Interface» «Plugin» ProtocolAdapter
+ adaptInput(in data: CI): I
+ adaptOutput(in data: O): CO
«Interface» ChannelProtocolAdapter
+ getInputChannel(): String
+ getOutputChannel(out null: String)
ConnectorInputTypeAdapter
ConnectorOutputTypeAdapter
+ connectorinputtypeadapter
+ connectoroutputtypeadapter
«bind» <O -> O, I -> I, CO -> CO, CI -> CI>
«Plugin» AbstractProtocolAdapter
+ abstractprotocoladapter
«Plugin» TranslatingProtocolAdapter
+ translatingprotocoladapter
«Interface» ConnectorInputTypeTranslator
+ inputTranslator
«Interface» ConnectorOutputTypeTranslator
+ outputTranslator
AbstractConnectorInputTypeTranslator
ChannelTranslatingProtocolAdapter
AbstractConnectorOutputTypeTranslator
«Interface» ModelAccess
+ getQSeparator(): String
+ call(in args: Object): Object
+ get(in qName: String): Object
+ set(in qName: String, in value: O...
+ registerCustomType(in cls: Class)
+ monitor(in qName: String)
+ useNotifications(in not: Boolean)
0..1
+ modelaccess
AbstractModelAccess
«Interface» ModelAccessProvider
+ getModelAccess(): ModelAccess
+ setModelAccess(in access: ModelAccess)
1

«Implemented» Detailing the Connectors Component. ModelAccess abstracts over model-based protocols like AAS and provides uniform access. Protocol adapters are used by the connectors to translate the protocols. Therefore, we reuse tyoe translators from the Transport Component, augment them with model access, provide basic implementations for standalone type translators as well as adaptors to re-use existing Serializers.

«Issue» Papyrus does not show the template signatures of liinked classes (or I do not find the filter). Ultimately, we left out missing type signatures for the added interfaces and implementation classes, i.e., the classes are still type generic.

Incomplete. Methods for other types and access to user defined structures look similar.

Figure 29: Model Access and Protocol Adapter in the Connectors Component.

We will now detail the ModelAccess and the ProtocolAdapter approach illustrated in Figure 29. Some approaches like OPC UA or Asset Administration Shells (AAS) are based on an information model, OPC UA even with user-defined custom types. Accessing this model in a uniform manner is a key requirement for simplifying the generation of application-specific code for the connectors. This is specified by the ModelAccess interface, which allows to read/write properties (based on a hierarchical naming scheme to be interpreted in the context of the underlying protocol), to call operations, and to register (the implementation counterpart of) custom types. The specific ModelAccess instance of a Connector can perform instance translations between value instances of the model and the actually used internal type in the platform/application datatypes. ModelAccess provides also opportunities to establish monitors on the underlying information model of the protocol, i.e., to be notified on specific changes, as well as to register programming language counterparts of custom types defined in the model.

While all methods can be implemented for OPC UA, not all methods are currently meaningful for AAS or at least the version of BaSyx that we are using, i.e., some capabilities may not be supported which can be indicated in the meta-data of a Connector. The use of the abstracted model access is supported by AbstractModelAccess providing a common basic implementation. For payload-based protocols such as MQTT, implementing the ModelAccess interface is not required.

As illustrated on the left side of Figure 29, several further interfaces and classes are defined to support the type translation. These classes shall provide flexible support for applying the type translation mechanism and even to utilize existing serializers if applicable. If required, a Connector may be also implemented from scratch, i.e., without a ProtocolAdapter or using a ProtocolAdapter that is not based on TypeTranslator from the Transport Component. The AbstractProtocolAdapter is a default implementation providing access to the ModelAccess instance of the connector. The TranslatingProtocolAdapter is a refined default implementation of ProtocolAdapter using two type translators, one for the translation of information provided by the protocol, i.e., types O to CO, and one for the translation in counter direction, i.e., types CI to I.

As discussed above, the role of the ProtocolAdapter and the involved type translators changes when the underlying protocol is based on an information model rather than on payload transport. For payload transport, the target communication channels are needed, which are introduced by the ChannelProtocolAdapter and its default implementation ChannelTranslatingProtocolAdapter, an extension of the TranslatingProtocolAdapter. In contrast, for an information model-based protocol, also the ModelAccess instance must be made available to the type translators as well as further initialization work such as defining the polling mode must be performed. This is introduced by the two refining type translator interfaces, namely ConnectorInputTypeTranslator and ConnectorOutputTypeTranslator, both with a corresponding basic implementation.

One last aspect is covered by the types package on the left side of Figure 29. The direction of type translation of Serializer in the Transport Component is opposite to the direction for a ProtocolAdapter in the Connectors Component, i.e., Serializer instances do not fit directly into these. In some cases it would be convenient to use already defined Serializer instances. To facilitate this reuse, we introduce the ConnectorInputTypeAdapter and the ConnectorOutputTypeAdapter, which both take a Serializer as input and make it usable in the context of a Connector.

3.6.3.3 Validation

The functional validation of the Connectors Component and the specific connectors realized as extensions happens through regression tests. Therefore, we follow the same basic idea as explained for the Transport Component in Section 3.6.2.3, i.e., we set up a corresponding protocol server/broker and cause an information shortcut between server side and test code. The test code produces protocol output data (of type O) either by modifying the underlying information model (event-based ingestion, polling) or by sending respective payload. The connector under test translates the data and issues an instance of type `CO` to a `ReceptionCallback` in the test code, which turns the information into an instance of `CI` and writes it back into the connector. The respective information must occur on the protocol side and can be analyzed and asserted by the test code. Also these regression tests are subject to Continuous Integration.

So far, no performance analysis with the connectors has been performed. This will happen after exercising some domain use cases that connect edge devices, brokers or other components. These combined conceptual and implementation tests are planned as next steps after this release, i.e., for summer/fall 2021.

3.6.4 Requirements Discussion

Finally, we review in this section the realization of the most relevant requirements for the Transport and Connection Layer. The results are summarized in Table 9.

Table 9: Review of realized[68] requirements for the Transport Layer (based on Table 2, Table 3 and Table 5).

Requirement	Summary
R1	Support for protocol/streaming extensions of different vendors based on different technologies.
R2	Used standards: MQTT, AMQP, OPC-UA, AAS
R3	Virtual platform: Container integration possible, communication with underlying platforms possible but not responsibility of this layer
R4	Design based on components and services
R5	Eclipse Paho, Rabbit MQ, Eclipse Milo, Eclipse BaSyx, Spring Cloud Stream, more for testing
R6	Open for optional/commercial components (transport connectors, serializers, machine/platform connectors, protocol adapters, etc.)
R7	Basic information on transport as well as available/active machine/platform connectors is provided. More information regarding supported protocols or broker may follow.
R8	On component/extension level: Alternative `TransportConnector` and Spring Cloud Stream binder components without cross-dependencies among the protocols (except for re-use in testing). Optional platform/machine connectors without cross-dependencies among the protocols (except for re-use in testing).
R9	*Level 1:* By auto-reconnect mechanisms of the protocol implementations and of the streaming library *Level 2:* By monitoring the service execution and restarting services if needed (see ECS runtime in Section 3.8.1) *Level 3:* By explicit fallback, i.e., hot-standby replication of services, multiple connected broker installations and dynamic stream rerouting. Level 3 is supported by the selected streaming library, *but not realized in this release*.
R10	Soft real-time processing (<100 ms) for production-critical functions feasible (see Sections 3.6.2.3 and 3.6.3.3, excluding services on this level)
R11	Documentation (also in terms of this section), extensive code documentation with JavaDoc, generation subject to Continuous Integration and Maven deployment

[68] In the requirements review tables, „realized" refers to implemented in terms of functionality, committed into the IIP-Ecosphere Github repository, tested and integrated with the platform functionality. Text in italics refers to missing functionality, i.e., entries that are partially formatted in italicts typically indicate partial realization. Work in progress or incomplete/non-integrated realization may be excluded from platform releases.

Requirement	Summary
R12	*Not implemented in this release.*
R13	Connectivity to other actors via standardized and open protocol integration
R13a	I4.0 devices: Via standardized/open protocol integration and flexible wire formats.
R13b	I4.0 platforms: Via standardized/open protocol integration and flexible wire formats.
R13c	Other IIP-Ecosphere platform instances: Supported via implemented AAS connector, optionally using IDS functionality. Supported by standardized transport protocols and flexible wire formats.
R14	Open and flexible connector integration and default connectors for platform internal messaging and stream transport.
R14a	MQTT is supported. OPC UA PubSub is currently not feasible on this layer due to missing implementations. Services and Connectors Layer (cf. Section 3.7) will take OPC UA into account.
R14b	TCP/IP support by all implemented connectors. Further, plain TCP/IP protocols with flexible wire formats possible through extensibility of discussed components.
R14c	*Future: Bluetooth LE may be supported through additional connectors if required.*
R14d	Connectivity at runtime possible through connector selection/instantiation at runtime.
R15	Uniform connectors through two main interfaces as well as type transformation / serialization interfaces.
R16	Open and flexible connector integration shown for five protocols. Extension by further protocols from partners and externals possible, supported through Open Source development Github.
R17	Distribution of connectors to devices by considering the Transport Layer as re-usable, deployable component only with dependencies to selected protocol implementations.
R17b	Management of connectors by platform through design management classes, exhibited by respective AAS (R7)
R17c	Parameterizable connectors through parameter objects and connector plugins such as the `ProtocolAdapter`.
R18	*Connectors with specific security mechanisms are not part of this release.*
R19	Minimal number of internal wire formats through common type-safe data serialization while enabling application-specific data types. Feasible wire formats can be selected through serialization implementation/generation.
R19a	Example input formats (southbound), covered by application-specific types and generic serialization. 50 values per data item feasible see Sections 3.6.2.3 and 3.6.3.3.
R19b	Example input formats (northbound), covered by application-specific types and generic serialization.
R19c	Restful APIs with JSON/XML through AAS implementation and wire format as argued in this section.
R19d	Example output formats (northbound), covered by generic serialization
R19e	Output data clocked in 5 s intervals possible (see Sections 3.6.2.3 and 3.6.3.3, excluding services on this level)
R19f	Via the `TypeTranslator`, configuration and code generation
R19g	Not part of this layer, supported at least through `TypeTranslator`, configuration and code generation
R20	Supported by streaming library, to be realized by glue code generation (through the configuration model).

Requirement	Summary
R21	Low impact on data throughput (see Sections 3.6.2.3 and 3.6.3.3, for existing connectors, excluding services on this level)
R22	Platform data throughput of 500 GBytes per year (see Sections 3.6.2.3 and 3.6.3.3)
R28	Machine pulse of 8 ms feasible (see Sections 3.6.2.3 and 3.6.3.3)
R35	OT sampling frequency of 2 ms does not apply to the IT side.
R38-R68	*Several security and privacy mechanisms that are introduced in Figure 17 are used to ensure introduced security and privacy mechanism. These mechanisms can be added to different layers, phases, and abstraction levels of design. Security mechanisms are indicated in the architecture but not part of the implementation of this platform release.*
R91	7 GByte per hour not validated on this level (see Sections 3.6.2.3 and 3.6.3.3, excluding services on this level)

We conclude, that most of the basic requirements for this layer are already implemented. Advanced functionality as well as security and data protection mechanisms (although prepared through respective abstractions) are subject to one of the next releases.

3.7 Services Layer

The Services Layer introduces the basis for deployable services, i.e., their interfaces, data flows, monitoring support, management and AAS representation. We separate this layer into two major components, one component to control/manage service instances and a second providing a unified execution environment for services. We start with a discussion of the terminology and background in Section 3.7.1 and detail then the requirements for this layer (Section 3.7.2). In two further sub-sections, we turn then to the two major components of this layer.

The service management component is generic and can be realized in the same way for all services. Due to the overall vision of IIP-Ecosphere to support easy-to use AI methods in intelligent production, AI functionalities shall be realized in terms of services ("AI services", named as AI toolkit in [8]). Nowadays, AI is typically realized using various programming languages, in particular Python (R113 names Python and Java). We support this in terms of a language-specific execution environment supporting a unified integration and easing the development of services for the IIP-Ecosphere platform. In Section 3.7.3, we discuss the Service Execution Environment for Java and Python.

The Control and Management component (Section 3.7.4) is closely related to ECS runtime and acts as control interface for the platform to take command over services running on certain devices. Control operations are, e.g., starting, stopping, reconfiguring or updating services. These operations are offered through an AAS, which also provides access to runtime monitoring information for individual services. Specific operations involve multiple services, such as switching among equivalent services or migrating services among resources, where the control and management component is responsible for the orchestration of such operations.

While we briefly discuss the validation of the individual components at the end of the respective section, we review the requirements for this layer in Section 3.7.5.

3.7.1 Terminology and Background

In this section, we briefly introduce our notion of the term service and discuss the bigger picture, where service implementations are supposed to originate from.

Several notions for services are used, ranging from web services to microservices. In the IIP-Ecosphere platform, a **service** is (a thread in) a process implemented in any programming language. A service receives data and produces data. Input and out data types are defined and their correct composition shall be controlled through the configuration model and the subsequent code generation (cf. Section 6). Data handling can happen synchronously, i.e., an input item is turned directly into zero or multiple output items, or asynchronously, i.e., the service receives data and produces data at any time later if at all. A service indicates its state (R4c), meta-information (R4b), name, identification, version, kind/category as well as the typed input- and output connectors (R4a). Moreover, it allows for certain runtime operations such as passivation, migration, runtime switch to an equivalent service, reconfiguration or (re-)activation. Services are typically connected to other services of different kind, ranging from source over transformation to sink services. A specific service kind is the probe service, with inspects data for monitoring, but passes the data through (typically) without modification. Instantiated connections between services are called data paths/relations [30] and shall be defined as part of the applications in the platform configuration.

Further, it is important to answer the question "**Where do services come from?**". Details of the mechanisms are introduced later, in particular in Section 3.11 and Section 6. Services are specified in the platform configuration, in particular through their meta-information and the input/output datatypes. Also the relations among the services in terms of application-specific service meshes are defined in the platform configuration. The code generation turns this information into service interfaces, data classes and data serializers. Further, it binds the service interfaces into service/glue code for the selected streaming engine. Along with that, also stub implementations of the service interfaces are generated that "implement" a non-Java service on the Java side and transparently send data through IPC/a network protocol to the non-Java service implementation and ingest returned data back into the Java streaming process. Dependent on the service configuration, data may be handled synchronously or asynchronously. As part of the generation process, also the service descriptors required by the Service Control and Management component or the Spring application setup including the service wiring are created.

3.7.2 Requirements

In Table 10, we summarize the specific requirements for the Services Layer discussed in [8]. The notion of a service is cross-cutting, i.e., it occurs in many topic areas in [8] and, thus, a summary of all relevant requirements is important for the design and realization. Besides these functional requirements, we must also take into account the decisions made so far, i.e., that services may offer a two-folded communication: 1) communication at lower pace for commands, status and quality properties via AAS and 2) soft-realtime communication via streams whereby the stream-integration shall be generated and flexible in order to allow for an exchange of the streaming approach. This is in particular important for monitoring (R4b, R4c, R4e, R4f, R133) of runtime properties and the runtime stream management, in particular to start, stop, connect (R20), update (R135), configure (R32), adapt (R69 and R31c, see also dynamic service selection in [8]) or dispose (R134c) services on demand. To be integrated in a flexible manner, monitoring and service management must be realized based on explicit interfaces, so that an exchange of the implementations becomes possible. If feasible, existing interfaces shall be utilized.

In the default stream processing approach in IIP-Ecosphere, i.e., Spring Cloud Stream (see Section 3.6.2.1), the micrometer[69] interface is used to exhibit monitored information in HTTP/REST style. Moreover, Spring provides specific management capabilities for Spring Cloud based service applications, e.g., to start services in individual processes. As micrometer is supported by several (commercial) monitoring tools, it appears to be a valid choice as monitoring interface, which, however, must be integrated with AAS (R7). For the stream management, it makes sense to reuse existing functionality from the Spring Cloud ecosystem, e.g., the Spring Cloud Deployer. It is important to make

[69] https://micrometer.io/

this functionality optional and to enable it when Spring Cloud Stream is selected as stream processing engine of a platform instance. However, most of the Spring management providers target rather specific (non-edge cloud) environments, so we will rely on the so called Local Deployer and integrate it as optional extension of the Service Control and Management component.

Moreover, the Service Layer must set the scenes for the management of heterogeneous service implementations (R113), including platform services that are more likely to be realized in Java or as Java interfaces to service implementations of underlying frameworks or platforms.

Table 10: Requirements for the Services Layer (excluding configuration, storage services, not repeating Table 2 or Table 3)

Requirement	Summary
R4a	Components/services must be described with their interface (input, output)
R4b, R131b	Components/services must be equipped with meta information (version, categorization)
R4c	Components/Services must have a queryable state
R4d	The execution of the services must be supervised
R4e	Service monitoring shall be parameterizable
R4f	Service monitoring shall be realized by application-specific services
R20	Application-specific data paths
R20b	Data paths can have properties/parameters
R20c	Data paths shall be managed by the platform
R29c, R70, R122f	Services shall describe their own quality properties and functions as AAS
R31	Container shall contain only the required components/services
R31b	Containers can contain optional components
R31c, R69	Alternative services for one task, even dynamic exchange of (alternative) services at runtime
R32	Configuration of services via parameters
R39	Personal data processing only for authorized users
R41	The security mechanisms shall be integrated with common directory services
R42	Further safety mechanisms must be configured uniformly
R46	Collection of personal data must be for specified, clear and legitimate
R47	Avoiding the processing of personal data as much as possible
R48	The platforms should not store data for longer than necessary
R49	Process personal data adequate and relevant to the legitimate purposes
R52	Store personal data in a structured, common and machine-readable format
R67	Capture and classify generated cookies or similar identifiers stored on end devices
R73a-f, R79	Supported datatypes: structured, heterogeneous time series, unstructured data, labeled data, meta data/data schema
R113	Support for different programming languages, e.g., Python
R132	Platform-supplied and application-specific services shall be supported
R133	Runtime support for applications (and the services an application consists of)
R133a	Status of services
R133c	Support for changing the status of services
R133d	Detection of failure states and functions to mitigate failures
R134c	Removal/disposal of services
R135	Update of applications (and the services an application consists of)

3.7.3 Service Environments

In the IIP-Ecosphere platform, the service environments provide implementation and execution support for services realized in different programming languages. Java services and non-Java services are integrated differently into (a Java-based stream-based) service execution engine. While Java services can be directly called, non-Java services are executed as individual processes and receive their control commands and data via inter-process communication/network, in particular an AAS command server. It is desirable to use a single server here, e.g., the “AAS command server”.

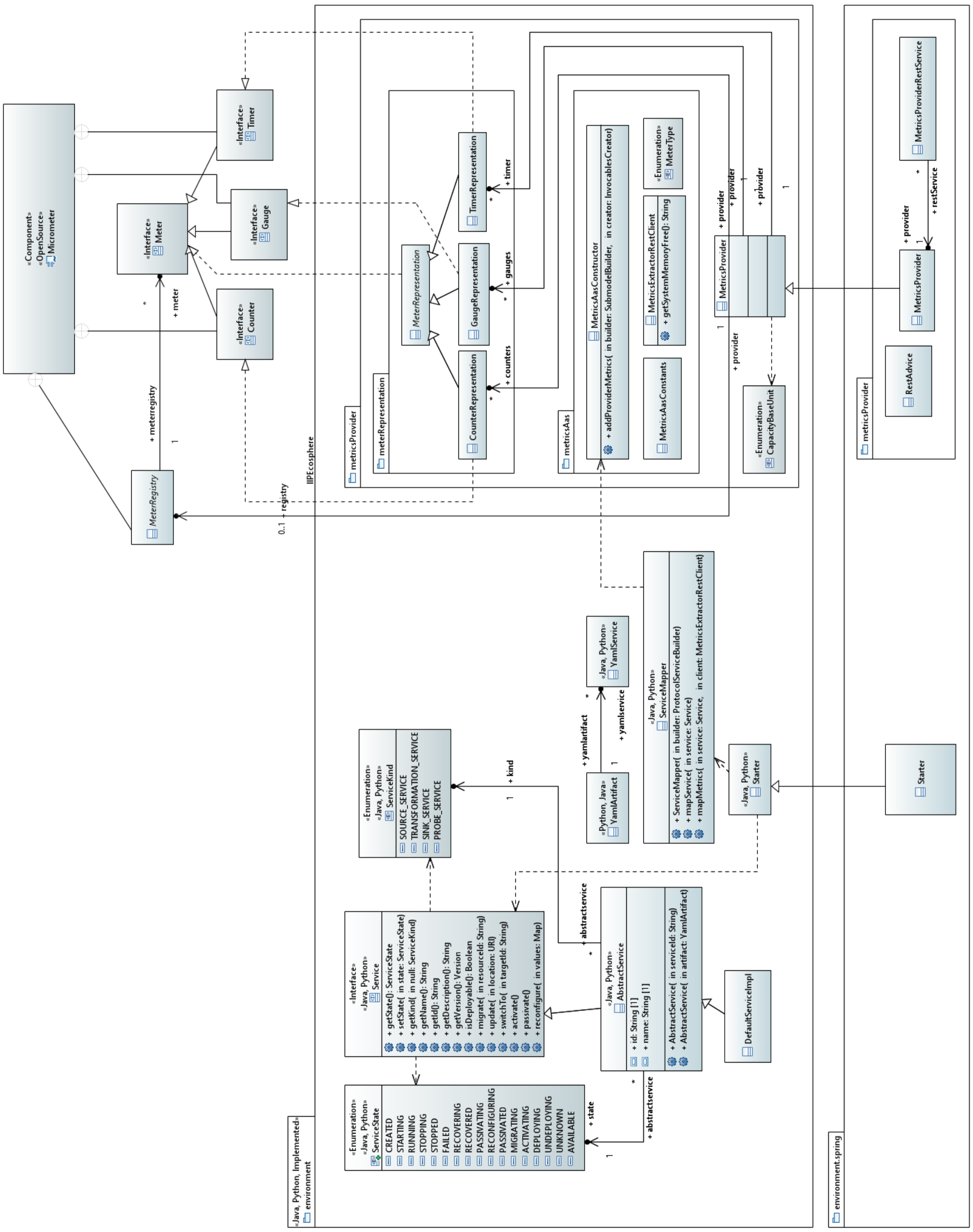

«Component» «OpenSource» Micrometer
«Interface» Timer
«Interface» Gauge
«Interface» Meter
«Interface» Counter
MeterRegistry
+ meterregistry
+ meter
IIPEcosphere
metricsProvider
meterRepresentation
MeterRepresentation
TimerRepresentation
GaugeRepresentation
CounterRepresentation
+ timer
+ gauges
+ counters
metricsAas
MetricsAasConstructor
+ addProviderMetrics(in builder: SubmodelBuilder, in creator: InvocablesCreator)
«Enumeration» MeterType
MetricsExtractorRestClient
+ getSystemMemoryFree(): String
MetricsAasConstants
MetricsProvider
+ provider
«Enumeration» CapacityBaseUnit
0..1 + registry
environment.spring
metricsProvider
MetricsProviderRestService
+ restService
MetricsProvider
RestAdvice
Starter
«Java, Python, Implemented» environment
«Enumeration» «Java, Python» ServiceKind
SOURCE_SERVICE
TRANSFORMATION_SERVICE
SINK_SERVICE
PROBE_SERVICE
+ kind
«Python, Java» YamlArtifact
«Java, Python» YamlService
+ yamlartifact
+ yamlservice
«Java, Python» ServiceMapper
+ ServiceMapper(in builder: ProtocolServiceBuilder)
+ mapService(in service: Service)
+ mapMetrics(in service: Service, in client: MetricsExtractorRestClient)
«Java, Python» Starter
«Interface» «Java, Python» Service
+ getState(): ServiceState
+ setState(in state: ServiceState)
+ getKind(in null: ServiceKind)
+ getName(): String
+ getId(): String
+ getDescription(): String
+ getVersion(): Version
+ isDeployable(): Boolean
+ migrate(in resourceId: String)
+ update(in location: URI)
+ switchTo(in targetId: String)
+ activate()
+ passivate()
+ reconfigure(in values: Map)
«Java, Python» AbstractService
+ id: String [1]
+ name: String [1]
+ AbstractService(in serviceId: String)
+ AbstractService(in artifact: YamlArtifact)
+ abstractservice
DefaultServiceImpl
«Enumeration» «Java, Python» ServiceState
CREATED
STARTING
RUNNING
STOPPING
STOPPED
FAILED
RECOVERING
RECOVERED
PASSIVATING
RECONFIGURING
PASSIVATED
MIGRATING
ACTIVATING
DEPLOYING
UNDEPLOYING
UNKNOWN
AVAILABLE
+ state

Figure 30: Design of the Service environments.

As the configuration modeling approach does not target the modeling of behavioral code rather than configuration options and their interdependencies, the actual implementation of services cannot be produced in this step. Such an approach would require a high effort in modelling, in particular (a combination with) modeling approaches that are better suited to represent algorithm behavior, call sequences etc. We do not target such complex models as the realization of functional services shall ultimately be a human activity, e.g., by a data analyst, AI expert, etc. and shall be integrated with their work. Thus, the interfaces produced by the code generation are input to a manual implementation process and interrelate in particular with later phases of the data analysis lifecycle. For this purpose, the code generation produces interfaces, data classes and serializers for all supported languages, i.e., currently Java and Python. The resulting code shall lead to versioned code artifacts (Maven artifacts, Python code with "python" as classifier) and be deployed into a repository. For the aforementioned deployment descriptors and the automated creation of containers (not part of this release) for the service execution, also non-Java dependencies, e.g., to AI frameworks are relevant. The respective artifact information is specified for the individual services in the configuration model and the packaging of the application artifacts at the end of the code generation process integrates the service code with service descriptor and application setup. Ultimately, the application artifacts are made available through an own repository mechanism (called the Service Store in [30]) so that either the automated generation of containers or the Service Management and Control component on the target resources can obtain the service code and install/execute it.

Figure 30 illustrates the concepts and relations of the service environments. The central package (`environment`) represents both, the basic service environment for Java and Python. This package (on the left side of Figure 30) defines the `Service` interface with all operations discussed for an IIP-Ecosphere Service in Section 3.7.3. The states from Figure 31 are represented in terms of the `ServiceState` enumeration, the four main kinds of services in terms of the `ServiceKind` enumeration.

A service realization is free to fill the service meta-information as desired, e.g., through code generation or by reading the information from a file. As the Service Management and Control component relies on service deployment descriptors, one obvious approach is read out the relevant information from that descriptor. As these descriptors are given in YAML format, the two classes `YamlArtifact` and `YamlService` are part of the service environment to read and represent that information. It is important to recall that we need here only a part of the information in the deployment descriptor, e.g., the technical information on how to transfer network ports or how to start a Python process are not required. Thus, the two classes represent only the relevant information and the YAML parser is commanded to ignore all further information. In turn, both classes can be used as a basis to realize the parsing of the deployment descriptor of the service management and control operations in Section 3.7.3. For this purpose, parts of the (Java) service environment are imported into the Service Management and Control component and used there.

The Java service environment also provides a `ServiceMapper`, a helper class that binds a service against a given AAS command server. Moreover, the `ServiceMapper` registers also the available metrics (see below) in the AAS command server. Ultimately, the `Starter` realizes a basic (optional) process to register all services given in a YAML service descriptor and to start the AAS command server on a given port. Services may also take care of a self-registration as it is the usual approach for Spring-based service implementations.

The right side of Figure 30 illustrates the extensible resource and service metrics framework based on the work of Miguel Gómez Casado [2]. All information to be monitored is represented in terms of gauges, counters or timers as defined in the micrometer monitoring interface[70]. We opted for micrometer, because a number of well-known monitoring tools such as Dynatrace support this set of concepts and because Spring Cloud Stream already exposes several default metrics via this interface.

[70] https://micrometer.io/docs/concepts

In more details, a *gauge* is a handle to get the current value of a monitored property, e.g., the number of threads in a running state. A *counter* is something that can be incremented or decremented by a fixed amount, while a *timer* is intended for measuring short-duration latencies, and the frequency of such events.

Micrometer provides interfaces and basic implementations for these concepts for the provider/service side, a JSON format to transport the information and through Spring a server to expose this information in terms of REST. However, the use of the micrometer ends at the service side, as typically accessing the monitored information is not part of the interface. Usually, the information is requested through some form of REST client. This would contradict our basic requirement to try to realize all (distributed) communication via AAS (R7) or Industry 4.0 protocols (R14). Therefore, we use the REST information only locally and map this information into the AAS of the respective services. Moreover, it would be convenient to access the data of the meters in uniform manner also in other platform components, e.g., in the platform monitoring component. The first step towards this goal is to realize a request-side implementation of the micrometer representing (distributed) meters (package `environment.metricsProvider.meterRepresentation`). The second step is to encapsulate the communication, i.e., the micrometer REST communication as well as the AAS representation of the metrics. This is done in terms of `MetricsExtractorRestClient` and `MetricsAasConstructor` in `environment.metricsProvider.metricsAas`. Finally, the `MetricsProvider` in `environment.metricsProvider` defines the unified access to predefined micrometer elements such as the system memory, but also custom meters, e.g., to measure the stream throughput. We consider monitored values as properties of the respective AAS submodel elements. To obtain the values, we attach functors to these properties to read out the monitored values. These functors may either rely on polling individual values via VAB [2] (pure R7) or on pushing the entire metrics provider via the Transport Layer (R7 and R14) into a local data instance attached to the AAS functors.

The classes discussed so far are intended for generic stream processors. Spring Cloud Stream and Spring Boot require specific code for services, the integration of the metrics and for their startup process. While the built-in metrics can be activated through a setting in the Spring application setup and by adding a respective dependency, the additional metric mechanisms defined in `environment.metricsProvider` are not automatically integrated. As discussed in more detail in [2], this is handled in the VAB poll approach by the extended `MetricsProvider` for Spring, the `RestAdvice` and the `MetricsProviderRestService`. Moreover, the startup code in `Starter` hooks into the Spring startup process, i.e., it obtains the Spring Rest server port, it attaches the port to the `MetricsExtractorRestClient` used by the upcoming services and starts the AAS command server of the parent class at a point in time when this is permissible for Spring. The `Starter` class is then integrated by the code generation in the actual service start code, which finally consists of just a few methods refining or delegating work to the `Starter` classes defined in the service environment.

So far, we exclusively discussed the Java side of the service environment. Except for the monitoring and the Spring-specific implementation, the Python service environment is to a large degree a mirror of the Java service environment. Differences are:

- The Python environment is accessed through the Java representation of services in the streaming engine, i.e., the Python environment realizes an AAS command server (currently available for the TCP-based VAB protocol, the HTTP-based protocol is under development) as well as the soft-realtime data transport (not available in this release). If possible, the VAB server shall also serve for the data transport, even if some modifications to BaSyx classes are required.
- A second difference is that we do not plan to monitor the non-Java environments unless explicitly required, because stream measures can be taken on the Java side. Resource

measures such as memory consumption can be combined with the related Java process, i.e., the monitoring there requires an extension so that the resources consumed by the Python process can be taken into account. This form of Python resource monitoring is not part of this release of the platform.

The service environment is subject to automated regression and integration testing. In particular the monitoring classes are tested extensively [2]. Also the remaining classes of the Java/Python service environments are executed in regression tests, i.e., the Java based build environment also executes Python unit tests. However, many methods are intended to be used by a stream-based application. As done with the components before, a manual implementation of a test application and execution in particular of the Spring service environment might be helpful here, but may fall short for the plain Java environment (test metrics are currently accounted per Java project rather than across projects). While currently the test coverage of the service environment could be increased, the classes defined there are tested in terms of integration tests, e.g., through test artifact for the Spring Service Management and Control implementation.

So far, no performance evaluations of the generated code and the underlying service environment have been conducted. Therefore, the manually implemented service chains from the experiments discussed in Section 3.6.3.3 could be used as baseline.

Besides service-level tests, performance experiments for the VAB poll approach have been performed in [2]. Retrieving a meter via an AAS on a current Lenovo Z50-70 laptop requires 4-5 ms after a settling time of 200 repetitions, whereby most of the time is attributed to the AAS communication. In contrast, initial requests are comparatively slow (8-10 ms), probably an effect of JVM settling periods. Moreover, some meters can schedule own update operations, which doubles the round-trip time. In the current implementation, the `MetricsProvider` performs such updates only on request, thus, saving roughly factor 2 response time in average. The internal operation of the meters, in particular parsing the JSON information requires at maximum 70 µs, i.e., most of the response time can be attributed to communication and AAS operations. A modification of BaSyx classes as mentioned above for a unified data transport could also speed up these operations.

While the readings for the monitoring work fine, they are also just taken for individual measures rather than for full AAS. Polling all monitoring information from an AAS may induce a significant response time. Moreover, as detected during tests with the full platform components, a remotely deployed AAS reads out all values of all properties during its serialization for client use. This implies significant overhead and, more dramatically, in some cases even hang up the component. Thus, as indicated above, we realized a second approach based on turning the metrics provider into a micrometer-based JSON format and push this information into the (serialized remote) AAS. To avoid the problems mentioned before, the information is not written into the functors or the AAS rather it is implemented as a data instance shared by all functors of the service (represented by a submodel element). This shared instance can quickly react during AAS serialization (significantly faster than the 4-5 ms mentioned above) and initializes the transport connector lazily upon the first request. Moreover, this approach decouples the startup of the AAS implementation server as only the platform transport broker/server is used, which was already started along with the platform. Alternatively, for a plain AAS realization (R7), we could have realized a similar push approach through an operation on server side, but refrained from this idea as such a collector method shall not be part of the visible interfaces of the platform.

However, the push approach via the Transport Layer could leave the impression that the work on individual meters in an AAS is superfluous, in particular as the mechanism could equally be used to realize the central platform monitoring (cf. Section 3.8). This is not the case as discussions with other

AAS users show: Nowadays, typical AAS tend to expose a huge amount of static and dynamic data for the described assets, i.e., the expected/actual resource consumption is often mandatory in some form.

3.7.4 Service Control and Management

The Service Control and Management component defines the service-interface of a (compute) resource towards the platform. It must provide means to load a service implementation onto the resource (in terms of binary artifacts, e.g., from a central platform server), to identify the descriptive information about services (id, name, description, version, service kind) and to provide access to runtime capabilities, e.g., the state of the service, reconfiguration capabilities, or runtime monitoring values. As the execution of the services happens within their (programing-language) specific environment, the control and management component can be realized in generic manner.

Individual services must comply with a lifecycle that can be queried and influenced by the platform. The underlying lifecycle state machine is depicted in Figure 31. Services can be downloaded from the service store and become `Available` on the hosting resource. When triggered through the platform and the `ServiceManager`, a service is deployed (`Deploying`) and gradually turns into the `Running` state. If nothing bad happens at runtime, a service is stopped through the `ServiceManager` (turns to Stopping and Stopped) and if requested, may be removed from the resource (`Undeploying`, afterwards Unknown, not shown in Figure 31). At runtime, a service may be reconfigured, adapted or migrated (which may need passivation and activation). Further, a service may fail, which can lead to a recovery procedure (in the lower sub state machine in Figure 31). If the service becomes operational again, it jumps back into the upper sub state machine and there into the last “normal operation” state (via the UML H* deep history state) and goes on from there.

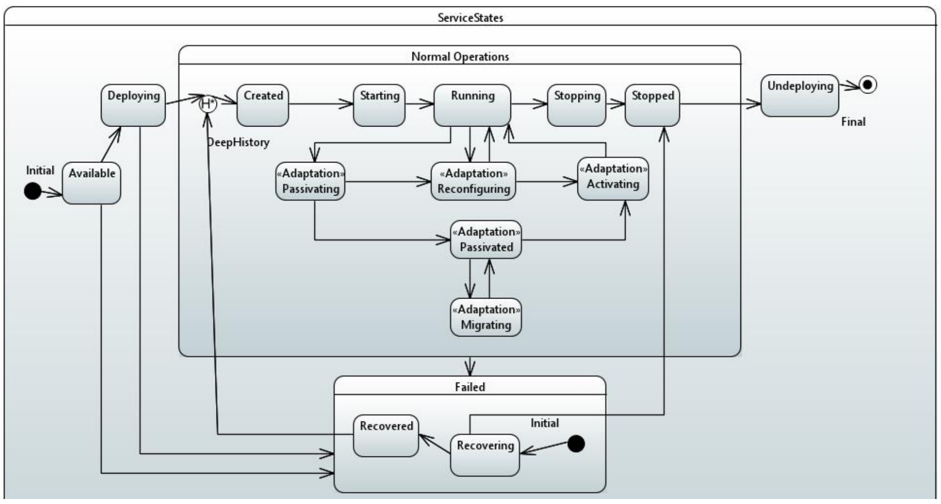
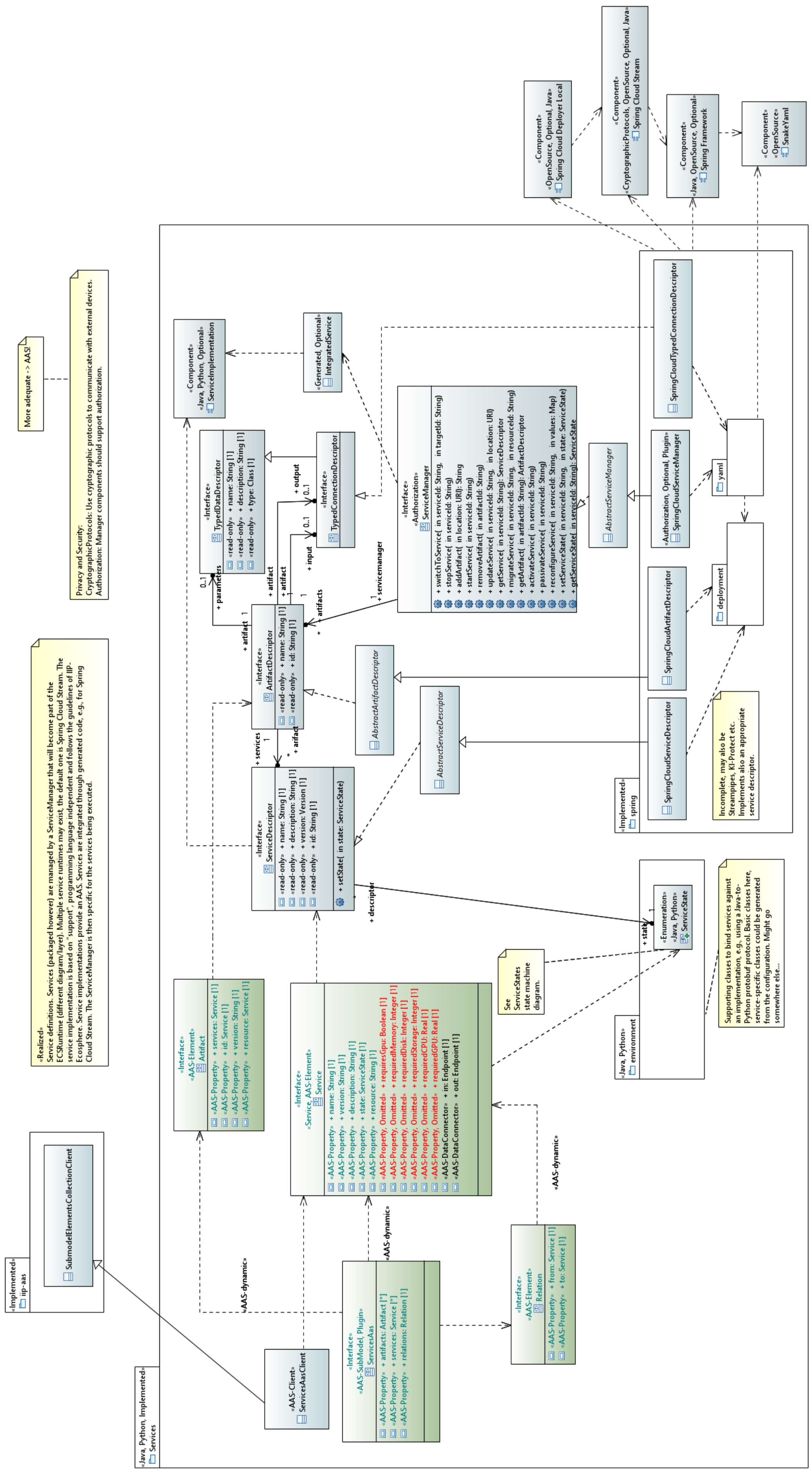


Figure 31: Service states (comments cropped)


«Implemented» iip-aas
SubmodelElementsCollectionClient
«Java, Python, Implemented» Services
«AAS-Client» ServicesAasClient
«Interface» «AAS-SubModel, Plugin» ServicesAas
«AAS-Property» + artifacts: Artifact [*]
«AAS-Property» + services: Service [*]
«AAS-Property» + relations: Relation [1]
«AAS-dynamic»
«Interface» «AAS-Element» Artifact
«AAS-Property» + services: Service [1]
«AAS-Property» + id: Service [1]
«AAS-Property» + version: String [1]
«AAS-Property» + resource: Service [1]
«Interface» «Service, AAS-Element» Service
«AAS-Property» + name: String [1]
«AAS-Property» + version: String [1]
«AAS-Property» + description: String [1]
«AAS-Property» + state: ServiceState [1]
«AAS-Property» + resource: String [1]
«AAS-Property, Omitted» + requiresGpu: Boolean [1]
«AAS-Property, Omitted» + requiredMemory: Integer [1]
«AAS-Property, Omitted» + requiredDisk: Integer [1]
«AAS-Property, Omitted» + requiredStorage: Integer [1]
«AAS-Property, Omitted» + requiredCPU: Real [1]
«AAS-Property, Omitted» + requiredGPU: Real [1]
«AAS-DataConnector» + in: Endpoint [1]
«AAS-DataConnector» + out: Endpoint [1]
«Interface» «AAS-Element» Relation
«AAS-Property» + from: Service [1]
«AAS-Property» + to: Service [1]
«Interface» ServiceDescriptor
«read-only» + name: String [1]
«read-only» + description: String [1]
«read-only» + version: Version [1]
«read-only» + id: String [1]
+ setState(in state: ServiceState)
+ services 1
+ arifact *
+ descriptor
«Interface» ArtifactDescriptor
«read-only» + name: String [1]
«read-only» + id: String [1]
+ artifact 1
+ parameters 0..1
+ artifact
+ artifact 1
+ input 0..1
+ output 0..1
* + artifacts
«Interface» TypedDataDescriptor
«read-only» + name: String [1]
«read-only» + description: String [1]
«read-only» + type: Class [1]
«Interface» TypedConnectionDescriptor
«Component» «Java, Python, Optional» ServiceImplementation
«Generated, Optional» IntegratedService
AbstractArtifactDescriptor
AbstractServiceDescriptor
1 + servicemanager
«Interface» «Authorization» ServiceManager
+ switchToService(in serviceId: String, in targetId: String)
+ stopService(in serviceId: String)
+ addArtifact(in location: URI): String
+ startService(in serviceId: String)
+ removeArtifact(in artifactId: String)
+ updateService(in serviceId: String, in location: URI)
+ getService(in serviceId: String): ServiceDescriptor
+ migrateService(in serviceId: String, in resourceId: String)
+ getArtifact(in artifactId: String): ArtifactDescriptor
+ activateService(in serviceId: String)
+ passivateService(in serviceId: String)
+ reconfigureService(in serviceId: String, in values: Map)
+ setServiceState(in serviceId: String, in state: ServiceState)
+ getServiceState(in serviceId: String): ServiceState
AbstractServiceManager
«Java, Python» environment
+ state 1
«Enumeration» «Java, Python» ServiceState
«Implemented» spring
SpringCloudServiceDescriptor
SpringCloudArtifactDescriptor
«Authorization, Optional, Plugin» SpringCloudServiceManager
SpringCloudTypedConnectionDescriptor
deployment
yaml
«Component» «OpenSource, Optional, Java» Spring Cloud Deployer Local
«Component» «CryptographicProtocols, OpenSource, Optional, Java» Spring Cloud Stream
«Component» «Java, OpenSource, Optional» Spring Framework
«Component» «OpenSource» SnakeYaml

«Realized»
Service definitions. Services (packaged however) are managed by a ServiceManager that will become part of the ECSRuntime (different diagram/layer). Multiple service runtimes may exist, the default one is Spring Cloud Stream. The service implementation is based on "support", programming language independent and follows the guidelines of IIP-Ecosphere. Service implementations provide an AAS. Services are integrated through generated code, e.g., for Spring Cloud Stream. The ServiceManager is then specific for the services being executed.

More adequate -> AAS!

Privacy and Security:
CryptographicProtocols: Use cryptographic protocols to communicate with external devices.
Authorization: Manager components should support authorization.

See ServiceStates state machine diagram.

Supporting classes to bind services against an implementation, e.g., using a Java-to-Python protobuf protocol. Basic classes here, service-specific classes could be generated from the configuration. Might go somewhere else...

Incomplete, may also be Streampipes, KI-Protect etc. Implements also an appropriate service descriptor.

Figure 32: Service interfaces and management

Figure 32 illustrates the design of the Service Control and Management component. At the core of this layer is the `ServiceManager`, which performs operations such as starting and stopping individual services. Services are packaged and transported in terms of artifacts, i.e., an artifact may contain multiple services realized in different programming languages. Instead of the actual instances that may be located in a different container, the `ServiceManager` primarily operates on descriptors, such as the `ArtifactDescriptor` detailing structural information on contained services. Access to artifacts and services happens through identifiers[71], whereby several operations and information accesses are delegated by the service manager to the descriptors or through the respective AAS to the AAS implementation server directly approaching the respective service instance.

The `ServicesAasClient` provides access to the properties and operations of the AAS of the service layer. To avoid adding even more visual complexity to Figure 32, we did not indicate the relation between `ServiceManager` and `ServicesAasClient`. Actually, both implement the same interface called `ServiceOperations`, which contains the basic operations of `ServiceManager` not requiring the (repeated, potentially inconsistent instantiation of) service descriptors. The `ServicesAasClient` can be used by upstream layers to conveniently access the services AAS.

Different technologies can be used to realize and execute service chains, i.e., to efficiently pass data along pre-defined data paths between the services, to transform data where needed etc. As part of such a service chain, data is turned into some form that can be transported by the utilized protocols. This serialization as well as the transformation of data to fit the input/output requirements of a service is part of the mechanisms of the Transport Layer. As discussed in Section 3.6.2, we rely on Spring Cloud Stream as default (stream-based) service execution engine. An integration of the transport level protocols and serialization mechanisms for Spring Cloud Stream was introduced in Section 3.6.2. As also stated there, we foresee that the IIP-Ecosphere platform shall support also other service engines in a flexible manner. Thus, the design of the Service Control and Management Component must allow for the execution of the management binding against alternative service execution technologies. For this purpose, the `ServiceManager` as well as the related descriptors are defined in Figure 32 as interfaces (in the package `services`), while the actual implementation is realized in a separate, alternative package (`services.spring` for the default engine), which hooks itself as implementation into the basic service management interfaces via JSL.

The default implementation of the `ServiceManager` in `services.spring` relies on Spring Cloud Stream, the Spring deployer mechanism and, in turn, on the Spring Boot framework. For Spring-based services, the packaging happens in terms of specifically packed JAR files (a form of “fat”, standalone JAR files also containing the required Spring base classes). Besides the Spring application configuration defining the actual wiring of the services, such artifact JAR files contain a deployment specification detailing the services, their communication, service dependencies and, if required, also non-Java service implementations and their integration. Following the conventions of the platform, these deployment specifications are stated in terms of YAML files. Both, the Java object representation of the YAML contents as well as the JAR artifacts are linked against the Spring-specific Artifact and Service Descriptors, which contain additional information required to manage services using Spring. The Spring-based `ServiceManager` utilizes the Spring deployer mechanism, i.e., the local Spring deployer. The deployment specification also allows defining external service implementation processes, e.g., for Python, so that the data communication is managed by Spring-services while the actual implementation of the service operates in an own process. By default, services are executed in their own processes so that services can be restarted in case of failures (R9) without accidentally shutting down healthy services. However, such a single-process deployment may not be desired in

[71] Identifiers are just services to comply with AAS types. Identifiers may be global to an infrastructure or local to a resource. Here, TT Plattformen will come up a naming schema after we made first experiences.

some cases so that the deployment descriptor allows for specifying “ensemble” services, i.e., Spring-services that must be executed within the same process.

The AAS for the Service Layer consists of a services sub-model indicating as sub-model element collections the (locally) installed artifacts and the contained services, the installed services and their properties as well as the data paths/relations among services. When a service is started, its state changes and for each data path a relation instance is created, i.e., a relation represents the instantiated data path between two service instances and points to the actual start and end service. Start and end service occur in the AAS as soon as the respective service is created. In turn, this information is used by the service manager to determine available services, e.g., during startup of dependent services in service chains. Most operations provided by the `ServiceManager` (also via AAS) are parameterized by an artifact or service identifier. However, internally the operations are bound to the resource the respective artifact/service is installed on, so these operations do not occur at the services in the services collection rather than for the resource in the resource collection. We will detail the resources in Section 3.8.1 as part of the design of the ECS runtime. As all those operations may fail, the implementation must not only return a result but also carry information about thrown exceptions when calling an AAS operations.

The service manager AAS is primarily intended as service-level control and monitoring interface. Services are supposed to register themselves with the respective local AAS command server (see also Figure 3 in Section 3.1) to react on command requests. Similarly, when monitoring information is requested, the (central or locally deployed) AAS communicates with the respective AAS command server. In case of services not implemented in Java, the respective service environment must provide an AAS command server and pass the information on to the service instances.

As discussed above, soft-realtime data streams shall not be transmitted through AAS rather than through the streaming engine (for our default engine using one of the protocols of the transport layer). If the service implementation is done in Java, the streaming engine will directly communicate with the service (potentially involving glue code generated from the platform configuration). If non-Java service implementations are used, the service representation in the streaming engine must route the data to the respective service environment, which shields the services from the actual communication and passes the data in adequate form to the respective service instances.

The requirements in [8] do not explicitly define the properties that shall be monitored for services. R29a, R70, R122f just indicate that services may have quality properties, e.g., to support adaptive service selection. Monitoring probes may be generic or bound to the services and, thus, are realized in the service environment (in particular the default one for Java, cf. Section 3.7.3). Similarly, the creation of related parts of the AAS are realized there. Further, probe services may be inserted to perform application-specific monitoring. However, probe services are currently not realized.

The `ServiceManager` and its AAS are validated in terms of regression tests. As the `ServiceManager` and the descriptors are interfaces/abstract classes, the validation must set up a pseudo implementation for basic testing. The Spring Cloud Specific functionality is tested through a handcrafted service artifact with simple contained services and multiple deployment descriptor targeting different artifacts, e.g., with or without process ensembles. This artifact is based on the Java service environment (cf. Section 3.7.3). In these tests, the setup of the `ServiceManager` provides a broker, dynamic network settings are handled by a local `NetworkManager` and the service manager is utilized for starting and stopping services. The running services are validated in terms of their data throughput and the actual metric values that the services provide, i.e., that the metrics defined in the service environment (cf. Section 3.7.3) become part of the AAS of the service management. Moreover, also the dynamic aspects of the AAS are validated, in particular during startup in order to figure out whether a service is already running.

Currently, as prescribed by the development streams in Section 3.2, design and realization focuses on a basic implementation to apply the IIP-Ecosphere approach as early as possible to existing resources. Advanced capabilities such as switching among alternative services or migrating services are subject to a later release.

3.7.5 Requirements Discussion

In this section, we review the already realized requirements for this layer. As mentioned in the sections before, we aimed for a basic implementation in this release, which is reflected accordingly in Table 11.

Table 11: Review of realized[68] requirements for the Service Layer (based on Table 2, Table 3 and Table 10)

Requirement	Summary
R4a	Services with input/output in `ServiceDescriptor` and AAS (`services` sub-model)
R4b, R131b	Metadata in `ServiceDescriptor` and AAS (`services` sub-model)
R4c	State in `ServiceDescriptor` and AAS (`services` sub-model)
R4d	The generic execution state of Spring Cloud Stream is supervised, also the resource consumption and stream processing or applications can define custom metrics. The state is reflected into the service descriptor, state and monitoring information (as far as enabled) show up in the AAS (services sub-model). *So far, services are not re-started if they fail as this is not provided by the Spring Local Deployer.*
R4e	*Not yet implemented, but possible e.g., through service descriptor.*
R4f	*Possible via probe services, but no basic implementation provided so far.*
R20	Application-specific data paths are supported through the streaming library (by Spring Cloud Stream, even at runtime). Data paths are dynamically indicated in the AAS (sub-model `relations`).
R20b	Data paths can have properties/parameters. Basic properties like the protocol or the encoding are supported by Spring Cloud Stream. Additional properties can be specified in the service deployment descriptor (Yaml file in artifact).
R20c	Data paths shall be managed by the platform through the configuration (Section 6), code generation, Spring application setup/service deployment descriptor and during service startup, e.g., taking dependent services into account.
R29a, R70, R122f	Services describe their functionality and their runtime properties (as provided, selected, implemented) through the Service Management and Control AAS, in particular supported by service monitoring.
R31	A container shall contain only the required components/services. This is supported through the service artifacts, that will be composed from the configuration model for a certain target deployment, i.e., with the (minimal) required resources. *Automatic creation of multi-resource service artifacts is subject of the next release.*
R31b	Artifacts may contain optional components, which are then not executed. *Optional services and their wiring is not subject of this release.*
R31c, R69	Dynamic exchange of service implementations is prepared by separating service implementation and (generated) binding against the stream processing library. The service interfaces allow for dynamic exchange and service migration and the operations are available through the AAS of the service management and control component. *The realization of the operations was not part of this development stream/release.*
R32	Services can declare and describe typed parameters. The `ServiceManager` supports changing these configuration parameters.
R38-R68	*A variety of security and privacy mechanisms are introduced (e.g. in Figure 17) which ensure relevant security and privacy requirements. Security mechanisms are indicated in the architecture but not part of the platform implementation of this release.*

Requirement	Summary
R73a-f, R79	As argued for the Transport/Connector components, we do not limit or prescribe types.
R113	The `ServiceManagement` and the stream processing approach are both realized in Java. Thus, services realized in Java can be directly integrated with the stream processing and be executed in the same threads/processes. Support for different programming languages, e.g., Python via IPC is foreseen in terms of the service environments. *A basic version of the service environment for Python is available, although HTTP-VAB and streaming data communication are not part of this release.*
R132	Application-specific services are supported through service interfaces as well as integration of artifacts in the code generation and packaging process.
R133	Runtime support for applications (status of services) is provided via the `ServiceManager` and the AAS.
R133a	The status of services is provided via the `ServiceManager` and the AAS
R133c	Within the limits of the service state machine, the `ServiceManager` and the AAS provide means for adjusting the state of a service. In particular, functions for activating, passivating and migrating services are provided and generically implemented.
R133d	*No specific functionality to resolve error conditions is provided in this release.*
R134c	The `ServiceManager` supports stopping as well as removal/disposal of services and service artifacts.
R135	An operation for updating services is provided and available through the AAS, *but the operation itself is not implemented in this release*.

We conclude, that most of the basic requirements for this layer are already implemented. Advanced functionality such as dynamic service operations or monitoring as well as security and data protection mechanisms are subject to one of the next releases.

3.8 Resources and Monitoring Layer

The Resources and Monitoring layer enables the deployment of services to (edge, server, cloud) devices, allows for overarching management of the devices and provides aggregated monitoring information about running resources and services. Moreover, the first platform components for the overall management of resources, namely the device management and the platform monitoring are located in this layer. We will discuss the ECS runtime in Section 3.8.1, the device management in Section 3.8.2 and the monitoring in Section 3.8.3.

3.8.1 ECS runtime

Flexible and heterogeneous deployment to edge, server and cloud devices is a central capability of the IIP-Ecosphere platform. [8] defines several requirements for the envisioned deployment approach. Table 12 summarizes the requirements to be taken into account. R25c and R25d target the (central) management of resources and, thus, are addressed by the device management in Section 3.8.2.

Table 12: Specific requirements for the heterogeneous deployment (excluding configuration)

Requirement	Summary
R23	Support for dynamic deployment
R24	Deployment to different types of resources/hardware
R25	Resource properties or functionalities described as AAS
R25a	AAS of a resource shall be realized by an ECS runtime
R25b	AAS of available resources must be announced to the platform
R25e	Resource AAS must describe static properties

Requirement	Summary
R25f, R103a	Resource AAS must describe dynamic properties, e.g., the resource utilization, the memory usage or the utilization of CPU/GPU/TPU cores
R25g	Resource AAS must contain functions for the deployment
R25h	Resource AAS shall contain functions for exchanging deployment units at runtime
R26	Deployment to on-premise resources
R27	Optional deployment to connected IIP-Ecosphere instances
R28	Optional deployment to cloud resources, e.g., Google Cloud or Gaia-X
R29	Deployment unit must provide an explicit interface in terms of an AAS
R29a	(Quality) properties and functional interfaces of deployment unit via AAS
R29b	Deployment unit AAS shall be linked to resource AAS
R29c	Contained services/containers shall be made available via the deployment unit AAS
R30	Deployment unit must be encapsulated as container
R30a	Deployment units on IT level must be technologically uniform
R30b	Deployment units on OT level can be technologically different
R30c	Platform can support the integration of external container repositories
R31	Container shall contain only the required components/services
R31b	Containers can contain optional components
R31c	Components/services in a container may be exchanged dynamically
R32	Container can contain data/models, to be configured via parameters
R33	Container can contain local data stores
R35	Sampling rate of 2 ms through container
R36	Optional configuration of resources
R36a	Writing of resource configuration
R36b	Reading of resource configuration
R37	Optional remote maintenance of resources
R38-R44	Security requirements
R45-R68	Data protection requirements

As described in [30], each device shall execute a basic runtime component (`ECSRuntime`) providing the AAS of the device and managing the containers in which individual services are executed. Figure 33 illustrates the design. The fundamental parts are the `ResourceUnit` representing the AAS of the resource on which the runtime component is executed as well as the `DeploymentUnit` containing the services executed on the resource. The Service Management and Control component from Section 3.7.3 contributes information to the `DeploymentUnit`, e.g., the running services and their instantiated relations. Through the ECS runtime, the device can receive and execute commands from the platform, such as downloading or starting a container. Moreover, different container technologies must be considered and addressed in a uniform manner through the ECS runtime.

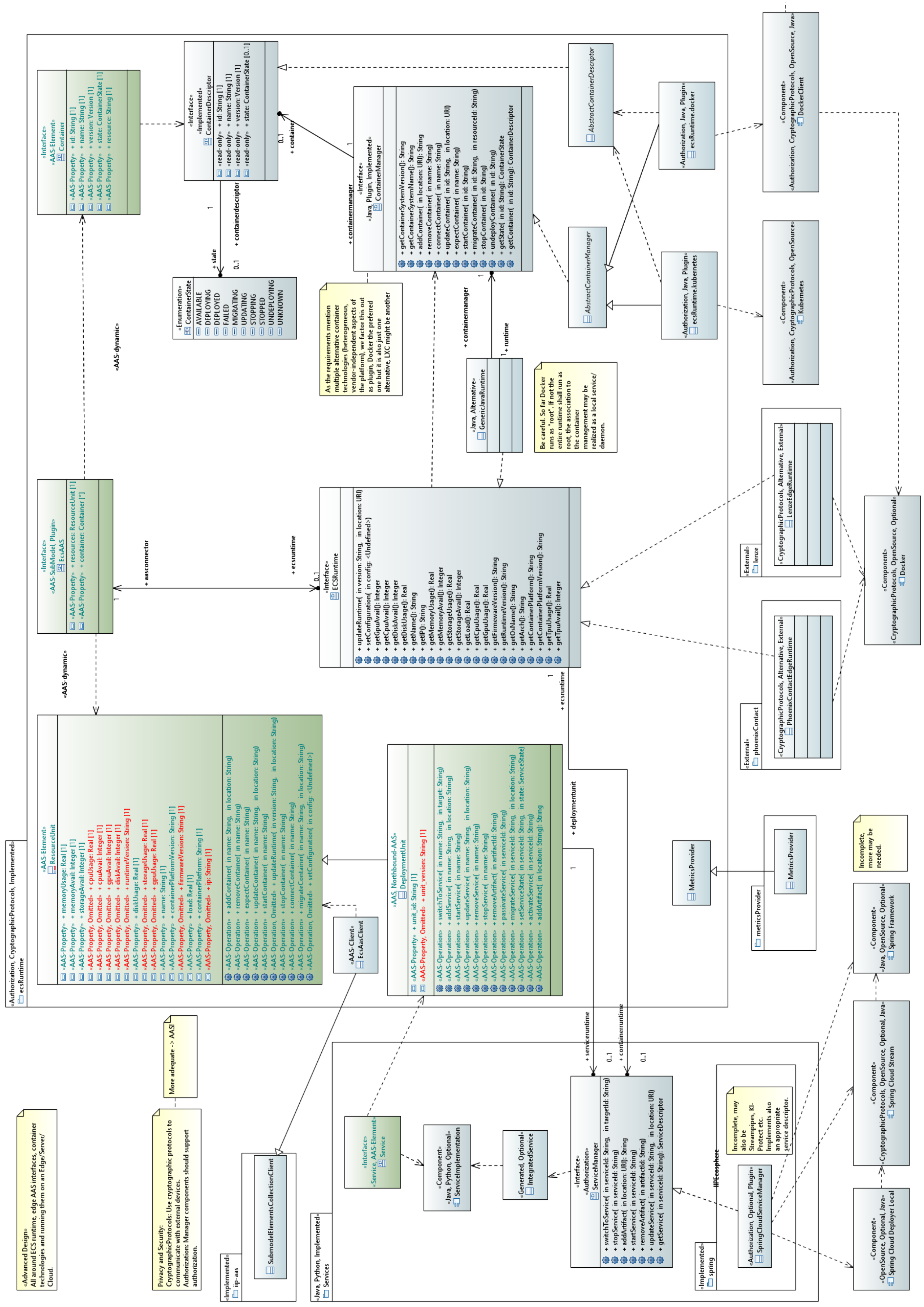


Figure 33: ECS runtime for Service Deployment (comments partially cropped)

Different ways to install such an ECS runtime are possible depending on the capabilities of the underlying device. The default approach is to provide an automatically created container with the instantiated ECS runtime as well as one or multiple (dynamic) containers for the services. Depending on the capabilities of the device, e.g., whether a suitable version of Java is available, the ECS runtime could also directly be installed on a device. If in the future a resource description such as the AAS of the ECS runtime is standardized (and IIP-Ecosphere platform is compliant with that standard), one could also imagine that the device already ships with an ECS runtime (possibly realized in some other language than Java) or it can be installed from the store of the device vendor. Measures to install, manage and update such installations are subject to the device management (Section 3.8.2).

As already emphasized in [8, 30], one fundamental basic work for the resource abstraction runtime is the LNI 4.0 edge configuration usage view [21]. [30] subsumes and extends [21] and [8] integrates relevant requirements from [30]. As the need for managing resources and containers on resources, in particular edge devices, is known in Industry 4.0, platforms [27] and also other work address this topic in various ways. In addition to the 21 platforms analyzed in [27], also approaches like OpenHorizon[72] or the IBM Edge Application Manager[73] are available. In recent time, also container orchestrators such as Kubernetes[74] became popular. Although there are significant overlaps, there are also important differences between these approaches and the ECS runtime in IIP-Ecosphere. One main difference is the general requirement R7 that all interfaces in the IIP-Ecosphere platform shall be based on AAS aiming at an interoperable integration of heterogeneous devices (based on an agreed structure, at least within IIP-Ecosphere). Moreover, it is important to point out that IIP-Ecosphere aims at a flexible integration of components leaving the final decision to the installing company through the configuration model, i.e., we do not make decisions such as statically relying on Kubernetes. In contrast to existing edge management approaches, as already pointed out in [30], IIP-Ecosphere aims at supporting more sophisticated management operations on the edge, in particular for data paths and relations as discussed in Section 3.7.

Figure 33 illustrates the design of the ECS runtime component. Figure 2 in Chapter 3 already discussed the context/stack for this component, i.e., on top of the AAS support, network management, transport and connectors and (optionally) service layer, the ECS runtime is supposed to provide a resource abstraction to manage the containers containing services to be executed on a resource. At the heart of the component is the `ECSRuntime` which acts as internal façade[32] for the ECS runtime AAS. Behind that façade, the actual operations are realized to be able to customize the ECS runtime for the resource at hands. Two example devices (produced by Phoenix Contact or Lenze) are indicated in Figure 33, but also a `GenericJavaRuntime`, which relies on an abstract `ContainerManager` (along with a `ContainerDescriptor`, akin to the service descriptors in Figure 32). IIP-Ecosphere provides a plain Docker[75] container manager (`DockerContainerManagement`). As for the service descriptors, the `ContainerDescriptor` is manifested in terms of a Yaml file, which is supposed to form the main entry point for adding a container at runtime, i.e., the platform specifies a URI pointing to the Yaml file, which indicates the name of the packaged container at the same location. We refrained from adding the descriptor to the packaged container as this may not be permissible for some container formats. A container manager for Kubernetes via Industry 4.0 protocols (R7, R14a) is in development but not part of this release.

The `EcsAasClient` provides access to the properties and operations of the AAS of the resources layer. To avoid adding even more visual complexity to Figure 33, we did not indicate the relation

[72] https://www.lfedge.org/projects/openhorizon/
[73] https://www.ibm.com/docs/en/edge-computing/4.1
[74] https://kubernetes.io/de/
[75] https://www.docker.com/

between ContainerManager and EcsAasClient. Both classes implement the same interface called ContainerOperations, which contains the basic operations of ContainerManager not requiring the (repeated, potentially inconsistent instantiation of) container descriptors. The EcsAasClient can be used by upstream layers to conveniently access the ECS runtime AAS.

At a glance, Figure 33 does not indicate much monitoring support for the ECS runtime except for some AAS properties in ResourceUnit. As the Java service environment (cf. Section 3.7.3) provides a generic and extensible monitoring approach, we re-use it here although the ECS runtime is not a "service". Thus, the ECS runtime defines a MonitoringProvider as well as a regular monitoring update operation that is started as part of the JSL lifecycle descriptor of the ECS runtime (not detailed in Figure 33). The operations to create the AAS refer to the MetricsAasConstructor of the Java service runtime mirroring a default set of meters of the monitoring provider into the AAS of the ECS runtime (therefore, currently some runtime properties in ResourceUnit are realized while others appear as omitted). Depending on future decisions, a specific set of resource meters can be defined and applied to both components in uniform manner.

Figure 34 illustrates two potential deployments to aforementioned example devices (including AAS server components, deployment interactions, a broker server and stereotypes from the UMLsec/security profile).

The AAS of this component is represented by EcsAAS, actually the resources sub-model already mentioned in Section 3.7. This sub-model consists of the ResourceUnit instances (corresponding to single ECS runtimes) representing the resources and the installed/running Container instances. The ResourceUnit offers the operations to manage containers on the respective resource. Moreover, ResourceUnit is extended by service operations if the resource offers a ServiceManager (either installed in the same or in a different container on the same resource) as discussed in Section 3.7.

As for the generic IIP-Ecosphere components, regression tests validate the basic operations of the ECS runtime, i.e., an artificial test container manager and its AAS. For the Docker-based container management, the regression tests utilize a small Open Source container image and exercise the implemented operations. Akin to services, currently advanced container operations such as update and migration are not implemented.

Initial experiments with containers and AAS indicated that properties and operations work as expected. Simple operations can be executed in at maximum 5 ms runtime (or significantly less for monitoring properties as discussed in Section 3.7.3). Complex operations, e.g., starting a container depend on the time that is required by underlying operation of the container implementation, e.g., Docker. Direct execution on an operating systems was not necessarily better in this regard. However, this experience strongly depends on the AAS and protocol implementation and, thus, is not representative. Further experiments with the Service Management and Control component from Section 3.7 will provide more insights.

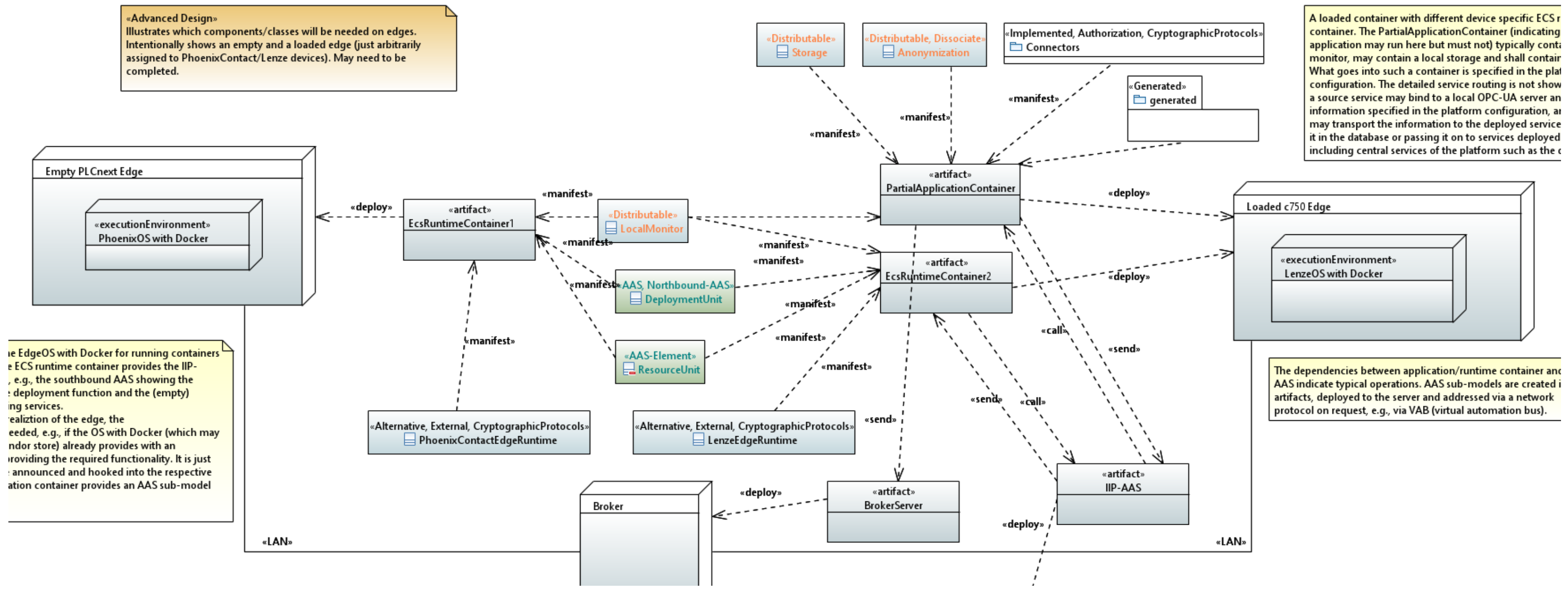

«Advanced Design»
Illustrates which components/classes will be needed on edges. Intentionally shows an empty and a loaded edge (just arbitrarily assigned to PhoenixContact/Lenze devices). May need to be completed.
«Distributable» Storage
«Distributable, Dissociate» Anonymization
«Implemented, Authorization, CryptographicProtocols» Connectors
«Generated» generated
A loaded container with different device specific ECS r
container. The PartialApplicationContainer (indicating
application may run here but must not) typically conta
monitor, may contain a local storage and shall contair
What goes into such a container is specified in the plat
configuration. The detailed service routing is not show
a source service may bind to a local OPC-UA server an
information specified in the platform configuration, ar
may transport the information to the deployed service
it in the database or passing it on to services deployed
including central services of the platform such as the c
«manifest»
«artifact» PartialApplicationContainer
«deploy»
Empty PLCnext Edge
«executionEnvironment» PhoenixOS with Docker
«artifact» EcsRuntimeContainer1
«Distributable» LocalMonitor
«AAS, Northbound-AAS» DeploymentUnit
«AAS-Element» ResourceUnit
«artifact» EcsRuntimeContainer2
Loaded c750 Edge
«executionEnvironment» LenzeOS with Docker
«call»
«send»
The dependencies between application/runtime container anc
AAS indicate typical operations. AAS sub-models are created i
artifacts, deployed to the server and addressed via a network
protocol on request, e.g., via VAB (virtual automation bus).
ne EdgeOS with Docker for running containers
e ECS runtime container provides the IIP-
, e.g., the southbound AAS showing the
: deployment function and the (empty)
ing services.
ealiztion of the edge, the
eeded, e.g., if the OS with Docker (which may
ndor store) already provides with an
roviding the required functionality. It is just
announced and hooked into the respective
ation container provides an AAS sub-model
«Alternative, External, CryptographicProtocols» PhoenixContactEdgeRuntime
«Alternative, External, CryptographicProtocols» LenzeEdgeRuntime
«artifact» BrokerServer
«artifact» IIP-AAS
Broker
«LAN»

Figure 34: Example deployments to empty (left) and loaded (right) edge device (comments and further deployment nodes representing the platform partially cropped)

Table 13: Review of realized[68] requirements for the ECS runtime component.

Requirement	Summary
R23	The ECS runtime provides the basis for dynamic deployment units. The actual deployment units are packaged during code generation for the configuration model. *However, automatic creation of containers is not part of this release/development stream.*
R24	Generic and specific implementations of the ECS runtime aim at supporting the deployment to different types of resources/hardware. So far, we provide a generic ECS runtime in Java with a default container manager for Docker.
R25	Common functionalities of the resources (as well as service operations) are defined in the AAS of this component. The AAS also reflects the monitored resource meters of an ECS runtime instance.
R25a	The AAS of a resource is realized by the IIP-Ecosphere ECS runtime.
R25b	The AAS can and shall be deployed to a central platform AAS, in particular to integrate with the service management operations. *This may have to be complemented with a registration function in the resource/device management (cf. Section 3.8.2).*
R25e	Basic static properties are provided such as resources hosting a container.
R25f, R103a	The AAS also reflects the monitored resource meters of an ECS runtime instance. *More specific meters such as the utilization of GPU/TPU cores may be added in one of the next releases.*
R25g	Resource AAS (`ResourceUnit`) defines functions for the deployment.
R25h	*Resource AAS (`ResourceUnit`) does contain functions for exchanging deployment units at runtime, but the functionality is currently not implemented.*
R26	Deployment to on-premise resources is supported by the ECS runtime.
R27	*Optional deployment to connected IIP-Ecosphere instances is intended for the third development stream and, thus, currently not realized.*
R28	*Optional deployment to cloud resources, e.g., Google Cloud or Gaia-X is considered in the third development stream and, thus, this requirement is currently not realized.*
R29	The deployment unit provides an explicit interface in terms of an AAS.
R29a	Functional interfaces as well as quality properties of a deployment unit are provided via AAS.
R29b	The deployment unit AAS is linked to the resource sub-model and the services sub-model.
R29c	Contained services/containers are available through the deployment unit AAS.
R30	A deployment unit is encapsulated as container, in particular the services are encapsulated in artifacts to be deployed individually into containers.
R30a	Deployment units on IT level shall be technologically uniform, through the general use of Docker containers. The `ContainerManager` supports the exchange of the respective implementation/integration.
R30b	Deployment units on OT level can be technologically different, but we aim for Docker as the default technology. The `ContainerManager` supports the exchange of the respective implementation/integration.
R30c	The platform can support the integration of external container repositories. The ECS runtime allows to obtain containers via an URL, which may point to an external container repository. *Container repositories are not part of this release.*
R31	A container shall contain only the required components/services as discussed in Section 3.7. *This depends on the packaging, for which an automated approach is not part of this release.*

Requirement	Summary
R31b	Artifacts may contain optional components, which are then not executed as discussed in Section 3.7. *This depends on the packaging, for which an automated approach is not part of this release.*
R31c	Components/services in a container may be exchanged dynamically as supported by the service management in Section 3.7. *However, this functionality is not part of this release.*
R32	Container can contain data/models. These are artifacts in the containers and respective (typed) parameters are offered by the services cf. Section 3.7. *So far, the packaging of further resources is not realized.*
R33	A container can contain local data stores (in terms of services cf. Section 3.7). *Data stores are not part of this release.*
R35	AAS operation/access speed in containers was around 5 ms. *A sampling rate of 2 ms through a container is plausible, but must be shown in future experiments.*
R36	*Configuration of resources is part of device management, currently not realized and intended for a later development stream.*
R36a	*Configuration of resources is part of device management, currently not realized and intended for a later development stream.*
R36b	*Configuration of resources is part of device management, currently not realized and intended for a later development stream.*
R37	*Remote maintenance of resources is part of device management, currently not realized and intended for a later development stream. This can potentially be integrated with container orchestrator operations.*
R38-R44	*Security mechanisms are indicated in the architecture but not part of the platform implementation of this release.*
R45-R68	*Data protection mechanisms are indicated in the architecture but not part of the platform implementation of this release.*

We conclude, that basic requirements for this layer are implemented. However, in comparison to the service management, connectors or transport component, in this component more advanced functionality is dependent on the automatic creation of containers or the device management. These components are scheduled for future releases.

3.8.2 Device/Resource Management

The device management shall support and ease the administration of devices, i.e., compute resources. As stated above, e.g. along with the ECS runtime in Section 3.8.1, the notion of devices in IIP-Ecosphere is rather broad as it involves edge, cloud and (on-premise) server devices. From a practical point of view, the scope includes all devices that potentially can run an ECS runtime (including the IT infrastructure from [30]). Also different forms of installation for an ECS runtime as discussed in Section 3.8.1 are subject to the device management. It is important to recall that following the scope prescribed in [30], Industry 4.0 field devices such as machines are out of scope for the IIP-Ecosphere platform.

From [8] we know that the main requirements for the device management refer in particular to the "Device Description Store", the "Device Configuration Tool" and the "ECS runtime" introduced in [30], i.e., also to the abstraction of vendor dependencies (R25.a), on/offboarding (R25a) or device management (R25b). Common management functions which are neither listed in [8] nor [30], e.g., mechanisms for human interactions (acknowledgements), management techniques such as device templates or import functions for "asset data providers" [30] are desirable, but also well covered by existing platforms [27]. Thus, in [8, 30] it is intentionally left open, whether the IIP-Ecosphere just focuses on the essential capabilities mentioned in [8, 30] or provides also additional useful capabilities.

Besides this freedom, there are requirements that also prescribe the design of the device management. One important requirement is R7 which requires the use of AAS for the interfaces of all layers/components in the IIP-Ecosphere platform. On the one side, the device management must take the information in the platform AAS on available resources into account and use the operations provided there to manage resources, i.e., this component can require its own operations in the resources sub-model elements collections described in Section 3.8.1. On the other side, the device management shall provide relevant own additional operations (such as onboarding, selection of device templates) to upper layers such as the user interface of the platform. Where adequate, these operations shall be parameterized with the resource identifier from the resources sub-model (cf. Section 3.8.1). The functionality of the device management is influenced by given information (through AAS events[76] and polling, R11), but may also directly influence the resource sub-model elements collection, e.g., adding/removing devices (potentially requiring subsequent operations, e.g., shutdown/migration of containers or services).

Moreover, the device management must take the virtual character of the IIP-Ecosphere platform into account (cf. Section 3.1). Therefore, it is mandatory that the device management is able to operate on multiple AAS of the structure described in this document rather than on “just” a singleton AAS of the IIP-Ecosphere platform. This allows taking other IIP-Ecosphere platform instances as well as underlying mapped-in platform instances into account. However, it is important to understand that access to these further AAS may be restricted, e.g., management operations are not allowed to be executed. This may be represented in terms of missing operations or AAS access limitations[77].

Primarily, for the device management Java 1.8 compatible libraries shall be used, although this constraint may be relaxed for this component as it will be utilized in the central IT installation. Regarding security (R38-R44) or data privacy (R45-R68), this layer may provide supporting administrative functionality.

Currently, ongoing work details the design of the Device/Resource management. Neither the design nor the implementation are part of this release.

3.8.3 Monitoring

Service execution shall be monitored, in terms of resources but also in terms of functionality, e.g., through application specific probes and alerts. Therefore, the IIP-Ecosphere platform foresees a set of generic built-in monitoring probes (cf. Section 3.7) as well as application-specific probe extensions that communicate their information via topic streams to one or multiple monitoring information aggregators. In turn, aggregators provide their state to upper level layers. Also (application-specific) alarming via specific streams shall be supported. In addition to the service monitoring, the IIP-Ecosphere platform shall also monitor resources via the installed ECS runtimes and also the execution of the ECS runtime.

While the probing of the individual services or ECS runtimes/resources happens on the devices (and thus belongs to Section 3.7 or Section 3.8.1, respectively), the main task of this component is to aggregate the information on IT infrastructure level (see also [30]). The aggregation of the received values shall follow existing guidelines, approaches, relevant standards or norms in I4.0. As the IIP-Ecosphere platform shall operate across a plethora of resources (and connected or underlying

[76] If possible, the component may rely on change events of the AAS implementation. However, in BaSyx events are currently in realization and, thus, not yet reflected in the AAS abstraction introduced in Section 3.5. Thus, the component design shall foresee event-based change notifications as well as (potentially less efficient) polling/scanning of the respective AAS structures.

[77] Currently, security and access restriction mechanisms are not (fully) in place in BaSyx and, thus, not reflected in the AAS abstraction introduced in Section 3.5.

platforms and their resources, if available), the monitoring component shall foresee (optional) hierarchical aggregation to distribute the input load and to increase the efficiency.

As described for the device management, the monitoring component must take the virtual character of the IIP-Ecosphere platform into account (cf. Section 3.1). Therefore, it is mandatory that the monitoring is able to operate on multiple AAS of the structure described in this document rather than on “just” the singleton AAS of the IIP-Ecosphere platform. This allows taking other IIP-Ecosphere platform instances as well as underlying mapped-in platform instances into account. However, it is important to understand that access to these further AAS may be restricted, e.g., access to information is limited. This may be represented in terms of missing properties or AAS access limitations[77].

Primarily, for the monitoring component Java 1.8 compatible libraries shall be used, although this constraint may be relaxed for this component as it will be utilized in the central IT installation.

Currently, ongoing work details the design of the Monitoring component. Neither the design nor the implementation are part of this release.

3.9 Security and Data Protection Layer

As discussed in Section 3.1, the purpose of this layer is not to realize typical cross-cutting security mechanisms, which will be subject to the security discussion in Section 7.

Currently, ongoing work details the design of this layer. Neither the design of the specific components nor the realization are part of this release.

3.10 Reusable Intelligent Services Layer

On top of the layers discussed before, the Reusable Intelligent Services Layer provides AI mechanisms in reusable and configurable manner and integrates received/monitored data with additional information such as product order information or floor plans to provide further valuable input to the AI. A particular aim is an AI toolkit facilitating the definition and composition of relevant, configurable AI-services for the IIP-Ecosphere platform.

Besides the platform overview in [27], ongoing work and discussion with project partners and other projects aim at collecting the state of the art/practice as fundamental input to the design of this layer. Similarly, work on the data integration is ongoing, but not included in this release.

3.11 Configuration Layer

It is important to recall that all relevant static and runtime information shall be reflected in terms of IVML structures, relations and constraints, while the IVML validation reasoner validates the platform configuration before and at runtime by identifying problems and deviations from validation rules and expected information. The Configuration Layer provides functionality to define applications in terms of the platform IVML configuration on top of the (reusable) services, to dynamically and adaptively optimize the deployment of services and containers and to adapt the use of services at runtime.

Table 14 summarizes all requirements from [8] regarding the configuration. The use of the configuration for resource optimization or adaptation is not listed in Table 14. In this release, we focus on the Configuration component (responsible for the configuration modelling and the instantiation) of the Configuration layer. Optimized container deployment and adaptive operations are deferred to future releases.

Table 14: Specific requirements for the configuration (in addition to the general requirements in Table 2, Table 3)

Requirement	Summary
R8	SPL approaches shall be used for variant management.
R8a	The platform must contain an integrated configuration model for applications, services and platform properties.
R8b	Automated validation of the configuration model
R8c	Automated derivation of platform instances
R12a	The platform can automatically derive the documentation of data processing methods from the configuration model.
R17a	Connectors shall be described in the configuration model.
R19f	The platform shall provide mechanisms for format adaptation or format conversion described in the configuration model.
R19g	The platform shall provide mechanisms for customization or manipulation of metadata as specified in the configuration model.
R20a	Data paths/relations must be defined in the configuration model.
R20b	Data paths/relations can have properties/parameters.
R25c	The platform must manage the available resources.
R31c	The required components to be installed into a container must be specified in the configuration model.
R31b	Containers can contain optional components.
R34	The creation of containers by the platform shall be automated, based on the settings in the configuration model.
R34a	A model validation can be performed before creation or execution to ensure executability.
R34b	The platform can support externally provided containers (e.g., for digital twins).
R36	The platform shall enable configuration settings for resources (read/write).
R40a	RBAC roles can be specified in the configuration model.
R40b	TLS certificates can be specified in the configuration model.
R41a	Directory services must be configured in the configuration model.
R42	Further safety mechanisms must be configured uniformly via the configuration model.
R43	Performance targets shall be considered in the configuration model.
R44	The configuration model shall offer IDS-based connectors as optionally configurable.
R64a	The specification of the data fields for anonymization shall be done via the configuration model.
R65a	The specification of the data fields for anonymization of personal data shall be done via the configuration model.
R73e	The data schema for storage services of structured data shall be defined in the configuration model.
R77a	If the platform supports cloud services, the configuration model must offer the use of cloud-based storage services as an option.
R80	Data (including meta-data) shall be described in the configuration model, including data protection classes.
R86	The functionality of the data integration shall be defined by the configuration model.
R89	The platform must allow the data integration write access to data. The data stores shall be defined in the configuration model.
R93	The platform must be systematically configurable in the form of a configuration model.

Requirement	Summary
R94	The platform must support the automatic validation of the configuration model for inconsistencies and errors.
R94b	Validating a configuration model with 50 resources and 5 applications shall be completed in less than 1 second.
R95	The platform must support automatic platform instantiation for a configuration.
R95a	The instantiation of a configuration model with 50 resources and 5 applications shall be completed in less than 15 minutes.
R96	The configuration model must represent optional and alternative platform components/services.
R96a	The configuration model must describe properties of the platform components/services.
R97	The configuration model must include the applications running on the platform.
R97a	An application configuration must contain the configured services for the application.
R97b	An application configuration must contain the configured connectors for the application.
R97c	An application configuration must contain the data paths of the application.
R97d	An application configuration shall contain alternative services.
R97e	The configuration model can allow for application templates.
R98	The configuration model shall support customizations at different times in the software lifecycle.
R99	Information from the configuration model can be made available to other components via internal connectors.
R100	The configuration model can be a decentralized model.
R101	Information provided in the AAS of components/services shall be mapped automatically into the configuration model.
R101a	The transfer of information for a configuration model with 50 resources and 5 applications shall be completed in less than 1 second.
R112a	Parameters of the AI services shall be described in the configuration model.
R112b	Properties of the distribution of AI services shall be described in the configuration model.
R112c	Distribution shall be subject to restrictions for individual AI services.
R113a	Technical dependencies to AI frameworks shall be specified in the configuration model.
R119b	The release of the trained model shall be determined via settings in the configuration model.
R119c	The release of the trained model can be automatic (if specified in the configuration model).
R119e	Changes initiated by a release of an AI model training shall be reversible, e.g., due to configurable criteria.
R120	The configuration model must describe alternative AI components for an AI method.
R122c	The adaptation must store its decisions in the configuration model.
R131a	The configuration model must support the specification of applications, their required services, connectors, involved data paths and the needed resources.
R131b	The configuration model must allow for the versioning of applications and services.
R131c	The configuration model can enable the parameterization of applications.
R131d	The configuration model shall support application templates for simplified configuration of requirements.
R131e	The configuration model must describe dependent applications or services.
R131f	The configuration of applications and data paths can be done in a graphical way.

Requirement	Summary
R132a	The configuration model must support application-specific services.
R133a	The platform must know the status of the services.
R133b	The platform must know the status of the running applications.
R134b	The platform can support the removal of applications from the configuration model.
R135	The platform shall support the update of applications.

Figure 35 illustrates the design of the configuration component. While the diagram (and the implementation) may appear rather trivial, most of the complexity is in the configuration model, the instantiation process and the underlying framework EASy-Producer.

As already discussed for Figure 1, the configuration model follows the layered architecture of the platform, i.e., each platform layer is represented by a configuration module. Figure 35 just indicates the topmost module, named `IIPEcosphere`, representing the configuration meta-model, i.e., the configuration options, their structures as well as constraints permitting certain configurations or propagating values among configuration options. We will discuss the model in more details in Section 6. For each platform to be installed, a dedicated platform configuration is created which specifies the AAS settings, the platform data types, the platform services etc. Moreover, for each application a separated (imported) configuration module shall be created, which contains the application-specific data types, the application-specific services as well as the service meshes (directed data flow graphs relating connectors and services) constituting the application. This combined platform configuration is one dedicated instance of `IIPEcosphere`, in Figure 35 an application configuration taken as input from the Application Layer is illustrated.

The platform instantiation process is defined based on `IIPEcosphere` meta-model, i.e., an instance of `IIPEcosphere` can be used as input that defines how the platform shall be instantiated. The platform instantiation process turns the configured information into source code artifacts, setup information, deployment descriptors and executable build scripts. This process also significantly contributes to the invisible complexity of this component. We will discuss also the instantiation process in more details in Section 6.

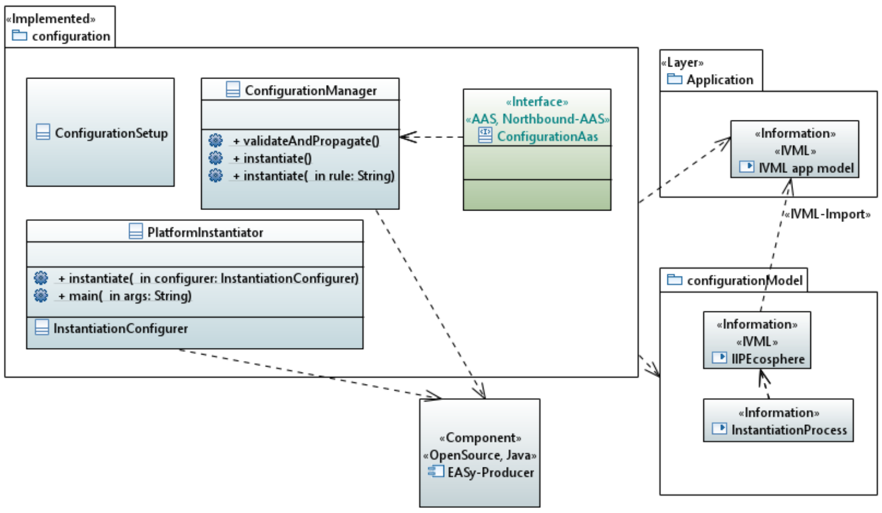


Figure 35: Configuration and instantiation of Definition of applications and orchestration of services (comments cropped)

On top of the models and the instantiation process, the Configuration component just orchestrates the relevant processes. The `ConfigurationSetup` (read from a Yaml setup file) defines the file system paths where the meta-model, its instance and the instantiation process are defined (meta-model and instantiation process are part of the respective release). The `ConfigurationManager` ensures the consistency of the operations, currently of loading, validating and instantiating the model. In future releases, also modifications to the actual instance of `IIPEcosphere` will be provided. These operations shall be reflected into the `ConfigurationAas` so that other components on this layer but in particular also a potential user interface can define applications or services and instantiate components. For now, the `PlatformInstantiator` realizes a command line tool to perform the basic operations of the `ConfigurationManager`, i.e., to allow a user to instantiate the platform and the defined applications.

Following the structure of the previous section, we discuss now the implementation of the configuration requirements. However, so far we did not detail the structure of the IVML model and the capabilities of the instantiation. This information is provided in Section 6, because we focus here on the architectural side. To avoid two separate discussions of the realized requirements, Table 15 includes forward pointers to Section 6 and summarizes already the requirements state explained there.

Table 15: Review of realized[68] requirements for the configuration (based on Table 2, Table 3 and Table 14)

Requirement	Summary
R8	SPL approaches are used for variant management.
R8a	The platform contains an integrated configuration model for applications, services and platform properties.
R8b	Automated validation of the configuration model is supported in terms of the constraints in the variability model and the EASy-producer IVML reasoner (cf. Section 6).
R8c	Automated derivation of platform instances is supported through the instantiation process (cf. Section 6).
R12a	*Derivation of the documentation of data processing is currently not supported.*
R17a	Connectors are part of the configuration model (cf. Section 6).
R19f	*Currently no mechanisms for format adaptation or format conversion are described in the configuration model.* Data serializers are implicitly derived during instantiation (cf. Section 6).
R19g	*Currently the configuration model does not support mechanisms for customization or manipulation of metadata.*
R20a	Data paths/relations are defined in the configuration model in terms of service meshes (cf. Section 6).
R20b	Data paths/relations can have properties/parameters although currently only the name is specified (cf. Section 6).
R25c	*The platform must manage the available resources. Resources are foreseen through the conceptual foundation of the meta-model but not used (cf. Section 6).*
R31c	*The required components to be installed into a container are currently not linked to resources.*
R31b	*Containers can contain optional components. Currently, services and resources are not linked.*
R34	*The automated creation of containers is not part of this release.*
R34a	A model validation is performed before instantiation and through the `ConfigurationManager` can be performed before further platform operations.
R34b	*Currently no externally provided containers are supported.*
R36	*The platform shall enable configuration settings for resources (read/write). These resource configuration settings may be reflected by the device management into the configuration.*
R40a	*RBAC roles are currently not specified in the configuration model.*
R40b	*TLS certificates are currently not specified in the configuration model.*

Requirement	Summary
R41a	*Directory services are currently not part of the configuration model.*
R42	*Further safety mechanisms are currently not part of the configuration model.*
R43	*Performance targets are currently not part of the configuration model.*
R44	*The configuration currently does not offer IDS-based connectors as optionally configurable.*
R64a	*The specification of the data fields for anonymization is currently not supported by the configuration model.*
R65a	*The specification of the data fields for anonymization of personal data is currently not supported by configuration model.*
R73e	*The data schema for storage services of structured data is currently not linked to data storages.*
R77a	*The platform currently does not support cloud services.*
R80	Data (including meta-data) shall be described in the configuration model, including data protection classes. Data types are supported*, meta data or protection classes are currently not part of the configuration model.*
R86	*The functionality of the data integration is currently not part of the configuration model.*
R89	*Currently no data stores are defined in the configuration model.*
R93	The platform is systematically configurable through a configuration model.
R94	The platform does support the automatic validation of the configuration model.
R94a	Validating a configuration model with 50 resources and 5 applications shall be completed in less than 1 second. *The models currently do not reach this size,* but on example configurations validation is currently not a bottleneck.
R95	The configuration model does support automatic platform instantiation.
R95a	The instantiation of a configuration model with 50 resources and 5 applications shall be completed in less than 15 minutes. *So far, test configurations* up to 3 applications including all platform components require less than 3 minutes on a computer in the same network as the (snapshot) code repository.
R96	The configuration model includes optional and alternative platform components/services (cf. Section 6).
R96a	The configuration describes properties of the platform components/services (cf. Section 6).
R97	The configuration model defines applications running on the platform (cf. Section 6).
R97a	An application configuration contains the configured services for the applications (cf. Section 6).
R97b	An application configuration contains the configured connectors for an application (cf. Section 6).
R97c	An application configuration must contains the data paths/relations of an application (cf. Section 6).
R97d	An application configuration does allow for alternative services (via families, cf. Section 6).
R97e	*The configuration model currently does not allow for application templates.*
R98	The configuration model does allow for customizations at different times in the software lifecycle *although not all relevant ones are defined* (cf. Section 6).
R99	*Information from the configuration model is currently not made available to other components via internal connectors.*
R100	*The configuration model is currently a centralized model.*
R101	*Information provided in the AAS of components/services is currently not mapped automatically into the configuration model.*

Requirement	Summary
R101a	*The transfer time for a configuration model with 50 resources and 5 applications is currently unknown as R101 is not realized.*
R112a	*Parameters of (AI) services are currently not described in the configuration model.*
R112b	Properties of the distribution of AI services shall be described in the configuration model. *Currently, the configuration contains only the information whether a services is distributable.*
R112c	*Currently no distribution apply.*
R113a	*Technical dependencies to AI frameworks are currently not part of the configuration model.*
R119b	*The release of a trained model is currently not considered in the configuration model.*
R119c	*The release of the trained model is currently not considered in the configuration model.*
R119e	*Changes initiated by AI model training are currently not subject to configurable release or quality criteria.*
R120	The configuration model must describe alternative AI components for an AI method. This is realized in conjunction with R97d.
R122c	*The adaptation is not part of this release.*
R131a	The configuration model supports the specification of applications, their required services, connectors and involved data paths. *However, currently needed resources are not linked to an application/service although the allocation to resources is prepared.*
R131b	The configuration model allows for the versioning of applications and services.
R131c	*The configuration model currently does not enable the parameterization of applications/services.*
R131d	*The configuration model currently does not support application templates for simplified configuration of requirements.*
R131e	*The configuration model currently does not describe dependent applications*, but service chains in service meshes.
R131f	*The configuration of applications and data paths is currently not done in a graphical way as no UI is provided.*
R132a	The configuration model does support application-specific services.
R133a	The platform must know the status of the services. *Currently no runtime data is reflected in the configuration.*
R133b	The platform must know the status of the running applications. *Currently no runtime data is reflected in the configuration.*
R134b	The platform can support the removal of applications from the configuration model. *The configuration layer currently does not provide detailed configuration manipulation operations.*
R135	The platform shall support the update of applications. *The configuration layer currently does not provide detailed configuration manipulation operations.*

We conclude, that basic requirements for this layer are implemented, in particular also for services and applications ([8] only states “the application”). However, there are many (cross-cutting) requirements for the configuration in [8] and in several cases the underlying platform components are not realized so that configuration modeling for those requirements is useless at the moment. These components/requirements are scheduled for future releases.

3.12 Application Layer

Ultimately, the Application Layer represents individual applications, i.e., it is the actual home of the application configurations to be installed, the generated artifacts and additional application-specific (handcrafted) components and services. The overall picture is depicted in Figure 36.

Currently, this layer does not really exist as platform instance/application configurations are defined as part of the tests of the Configuration Layer or on the command line of the respective tooling. Thus, generated and packaged artifacts are currently belonging to the Configuration Component (temporary, generated artifacts folder). The setup and the application layer will change in the next releases.

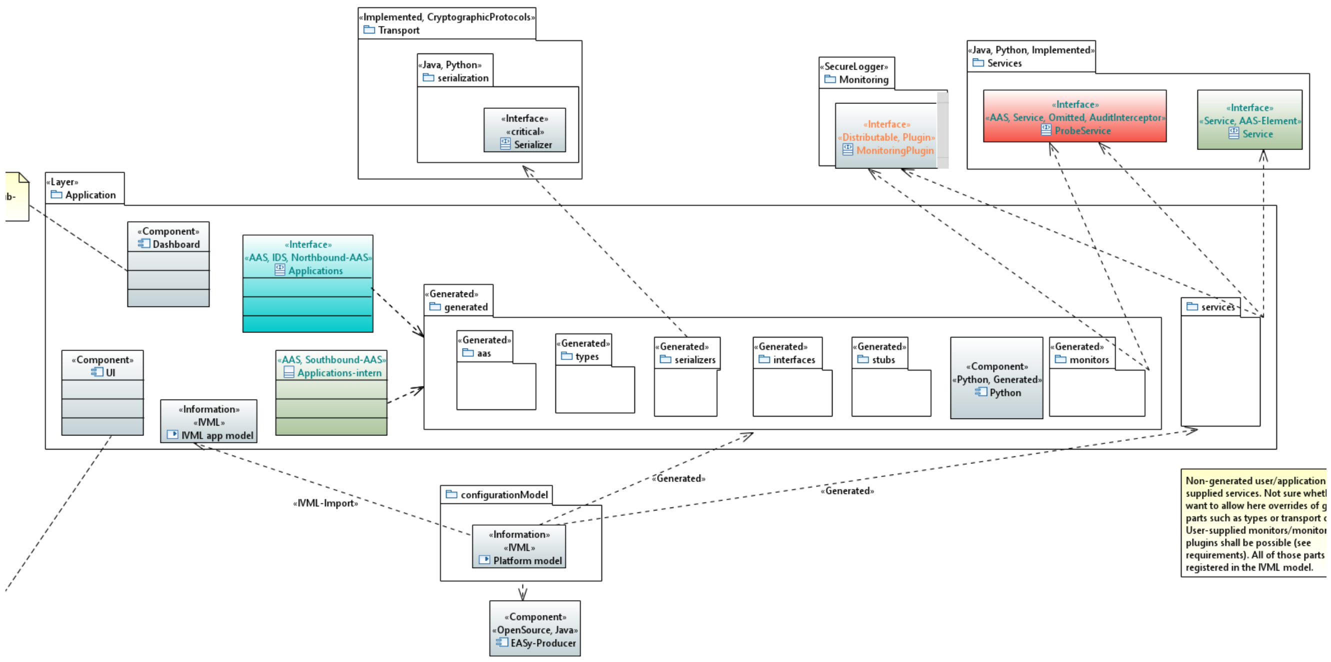

«Implemented, CryptographicProtocols» Transport
«Java, Python» serialization
«Interface» «critical» Serializer
«SecureLogger» Monitoring
«Interface» «Distributable, Plugin» MonitoringPlugin
«Java, Python, Implemented» Services
«Interface» «AAS, Service, Omitted, AuditInterceptor» ProbeService
«Interface» «Service, AAS-Element» Service
«Layer» Application
«Component» Dashboard
«Interface» «AAS, IDS, Northbound-AAS» Applications
«Generated» generated
«Component» UI
«AAS, Southbound-AAS» Applications-intern
«Information» «IVML» IVML app model
«Generated» aas
«Generated» types
«Generated» serializers
«Generated» interfaces
«Generated» stubs
«Component» «Python, Generated» Python
«Generated» monitors
services
«IVML-Import»
«Generated»
«Generated»
configurationModel
«Information» «IVML» Platform model
«Component» «OpenSource, Java» EASy-Producer
Non-generated user/application supplied services. Not sure whet want to allow here overrides of g parts such as types or transport c User-supplied monitors/monitor plugins shall be possible (see requirements). All of those parts registered in the IVML model.

Figure 36: Application Layer (comments cropped)

3.13 Platform Server(s)

As discussed above, the IIP-Ecosphere platform consists of several layers and many components. However, so far there also is a component that provides the setup and lifecycle mechanisms for the central IT-side of the platform, e.g., powering up the platform AAS service. At a glance, this component does not provide new functionality or concepts and may not be worth mentioning. In fact, it is a vital part for later platform instantiation, as it defines how central services can be configured, instantiate and how these services are started. Moreover, it provides an initial simple command line interface to operate with the IIP-Ecosphere platform, e.g., to start containers or services.

Figure 37 depicts the structural design of the `platform` component through using the server implementations and server-related parts defined in all layers and components discussed before. As stated above in this chapter, this component serves for two purposes:

1. Powering up the servers to run the IIP-Ecosphere platform. Therefor, the component defines a lifecycle descriptor (`PlatformLifecycleDescriptor`), which reads information from the `PlatformConfiguration`[78] representing the YAML setup file. The lifecycle descriptor is loaded via JSL into the `LifecycleHandler`, which, in turn, is called by the `platform`

[78] Due to terminological alignment with Spring there are currently two kinds of “configuration” in the platform. Some classes representing the setup as `PlatformConfiguration` as well as all components and files related to the IVML platform configuration (cf. Section 3.11). We did not resolve these overlapping names in this release, but may rename classes like `PlatformConfiguration` to names like `PlatformSetup`.

component during its main program. During this startup process, all "installed" lifecycle descriptors (e.g., the descriptor for the network manager; the platform instantiation is responsible for this) are also started up. As part of the startup also the platform AAS is constructed, which at least contains the platform "nameplate" (`Platform` sub-model).

2. Providing a simple command line interface (`Cli`) to experience the operations of the IIP-Ecosphere platform. The command line interface does not rely on the lifecycle mechanism, but on the `PlatformConfiguration` and, in particular, on the AAS clients of the service and the resources layer to ease executing the operations defined there. Figure 38 illustrates an example interaction with the interactive mode of the command line interface, here turning into the resources `commands`, showing the commands for resources (`help`), listing the available resources and, finally, ending the client. For the single resource shown in Figure 38, in particular the integrated container manager (for Docker) and various initial runtime measurements for disk and memory allocation are shown. It is important to emphasize that the command line performs its operations via the platform AAS and the respective AAS clients for services and the ECS runtime.

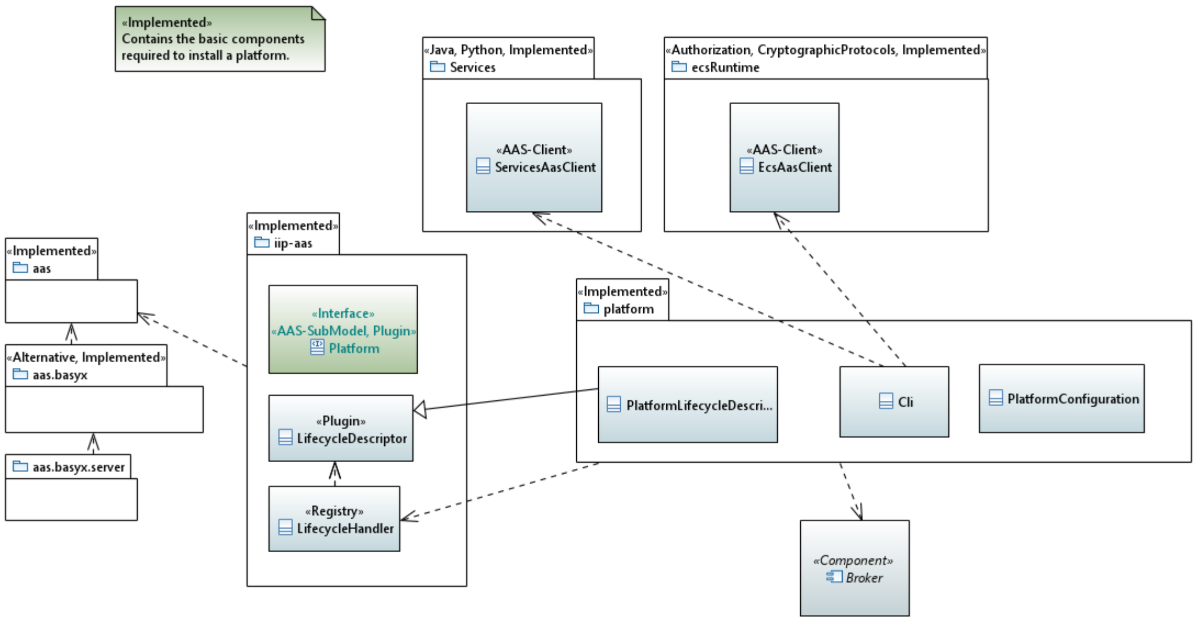

«Implemented»
Contains the basic components required to install a platform.
«Java, Python, Implemented» Services
«AAS-Client» ServicesAasClient
«Authorization, CryptographicProtocols, Implemented» ecsRuntime
«AAS-Client» EcsAasClient
«Implemented» aas
«Alternative, Implemented» aas.basyx
aas.basyx.server
«Implemented» iip-aas
«Interface» «AAS-SubModel, Plugin» Platform
«Plugin» LifecycleDescriptor
«Registry» LifecycleHandler
«Implemented» platform
PlatformLifecycleDescri...
Cli
PlatformConfiguration
«Component» Broker

Figure 37: Platform server(s) component

```
IIP-Ecosphere, interactive platform command line
AAS server: http://127.0.0.1:9001
AAS registry: http://127.0.0.1:9002/registry
Type "help" for help.
> resources
resources> help
  list
  help
  back
resources> list
- Resource a005056C00008
  systemdisktotal: 1023887356
  systemmemorytotal: 2147483647
  simplemeterlist: ["system.cpu.count","system.cpu.usage",
    "system.disk.free", "system.memory.free"…]
  containerSystemName: Docker
  systemmemoryfree: 2147483647
  systemdiskfree: 464061712
  systemmemoryused: 2147483647
  systemdiskusable: 464061712
  systemmemoryusage: 0.5555296172875698
  systemdiskused: 559825644
resources> back
> exit
```

Figure 38: Interaction with the preliminary interactive platform command line interface.

Using the platform command line interface, we validated the interaction among the components. Therefore, we started platform, ECS runtime and service manager component as individual programs. Through the command line interface, we validated the resource represented by the ECS runtime and started a simple generated application (cf. Section 6). We identified here the following issues:

- Unfortunately, BaSyx issues exceptions when checking whether an AAS exists through accessing it.
- Long running commands such as starting services are currently rather quiet on the command line interface, i.e., they do not show intermediary steps while the logs on the respective device indicate the actual state. AAS do not support return streams, so either polling from the caller or transmitting the results via the Transport Layer could be options for improvement.

We also validated the execution of services in a service manager container, starting and stopping of containers via the platform and the ECS runtime execution in terms of a (Docker-out-of-Docker) container. Please refer to Section 8.4 on how to install, instantiate and containerize the IIP-Ecosphere platform, i.e., to perform the steps that we also executed for validating the command line interface and the instantiated platform components.

4 Architectural Constraints

Besides structure and communication sequences, often an architecture explicitly or implicitly defines constraints that must be obeyed by an implementation. We summarize and explain the constraints for the IIP-Ecosphere platform here:

C1. Higher layers and contained components are allowed to have dependencies only to downstream layers and components, if possible only to the directly adjacent lower layer. This constraint is induced by the basic layered architecture style of the IIP-Ecosphere platform.

C2. As an exception from C1, the ECS runtime shall not depend on the Services Layer so that the services layer can be installed separately (as explained in Section 3). Both, Services Layer and ECS runtime may depend on certain classes of the services environment.

C3. Wrapped singleton components or libraries shall not be called by other components than the wrapper itself. Basically, this applies to transport and connector protocols, the AAS implementation (BaSyx), but also for container management libraries such as Docker. This constraint intentionally focuses on singleton components/libraries, as some libraries may occur in multiple component dependencies, e.g., the stream processing framework due to the need for different protocol/binder implementations. In turn, this also applies to some transport/connector protocol client implementations. Another exception is `support.aas.basyx.server`, which is allowed to access (as the only component) `support.aas.basyx` as it represents the server component with full dependencies.

C4. Support components for C3 shall be realized as optional components, e.g., the Spring service environment refining the generic Java environment. There shall be no references into such components except for refining components. In particular, generic components shall not reference their specialized components. For providing access to the specialized implementation, descriptors, factories or facades are to be used where the implementation is provided by JSL.

C5. Protocol servers for testing such as Apache Qpid, HiveMq or Moquette shall be in testing components and no other component shall directly use classes from them (although Maven requires explicitly naming also those transitive dependencies).

It would be desirable to check and enforce these dependencies. However, so far tools that we tried, e.g., in the continuous integration, failed for multiple components using a central or even adequately distributed rule set as they require an application rather than a component to be checked. We will try to find and integrate a feasible tool as soon as possible.

5 Asset Administration Shells

As stated above, the IIP-Ecosphere platform heavily relies on asset administration shells (AAS) to describe the capabilities and interfaces of its components. Currently, only few standard structures for AAS/sub-models exist while many are still in development, e.g., a software type-plate or a description of qualities of service (QoS). However, it is not feasible for the work on the IIP-Ecosphere platform to wait until such standards are defined. Thus, we follow an agile and pragmatic approach to AAS modeling here:

1. As long as **no guidelines** for AAS in IIP-Ecosphere exist, the IIP-Ecosphere platform will draft AAS that contain the most relevant information for its operation, i.e., for now the IIP-Ecosphere platform relies on AAS prototypes. All names and sub-structures shall be defined in terms of constants so that names and structures can be adapted (within limits). AAS of the IIP-Ecosphere platform shall be tested individually and also in integration settings to handle and to judge the impact of modifications. Tests shall also rely on the defined constants rather than on local String literals. In this stage, we pragmatically focus on AAS describing instances.
2. Discussions with third parties on **(proto-)standardized AAS** structures are ongoing. As soon as results from these discussions are available, a guideline for AAS modeling in IIP-Ecosphere shall be drafted. The experience made with prototyping AAS in the IIP-Ecosphere platform will be considered in these guidelines. At this point, also type AAS shall be provided.
3. The IIP-Ecosphere platform will modify the AAS prototypes and augment the information (e.g. eclass references) to **comply with the guidelines**. This may lead to a re-structuring of the AAS prototypes.

With this approach in mind, we designed and partially realized the prototypical IIP-Ecosphere AAS structure shown in Figure 39. The layers and their main components are represented by individual sub-models, e.g., `transport`, connectors (in terms of `installedConnectors` and `activeConnectors`), services (in terms of `installedServices` and `activeServices`), but also the applications and the resources (hosting ECS runtime instances). In addition, `platform` is a simple variant of a software nameplate for the whole platform (or the running top-level component) including build information. The `types` sub-model represents types that are used in the communication, e.g., in/output types of connectors (each equipped with respective generated serializers).

We distinguish between installed/available descriptors and their active instances at runtime, in particular as in many cases only the active instances provide the full information about in/outgoing types. Examples are in particular the connectors, the services and their relations, the containers etc. These structures are dynamic, i.e., they change due to installed components as well as due to instantiated/terminated instances. This is in particular the case for connectors and services, subsequently also for applications. Some sub-models are active, in particular those providing operations. One example for an active AAS is the optional `netMgt` submodel, which provides access to the local/global `NetworkManagement` defined in the Support Layer.

It is important to emphasize that the structure shown here is not static. It is dynamic in its elements as explained above, but it is also dynamic in its overall structure and contributions, in particular if the AAS is centrally deployed and parts are added remotely. A specific example is the relation between resources and services. When an ECS runtime comes up, it contributes itself to the resources collection. When a service manager starts, it contributes further operations to the resource it is running on, i.e., both Layers contribute into the same AAS sub-model (elements collection), because in this case the components have information and operations that they only can share individually but that are part of the same topic, namely the runtime interface of a resource.

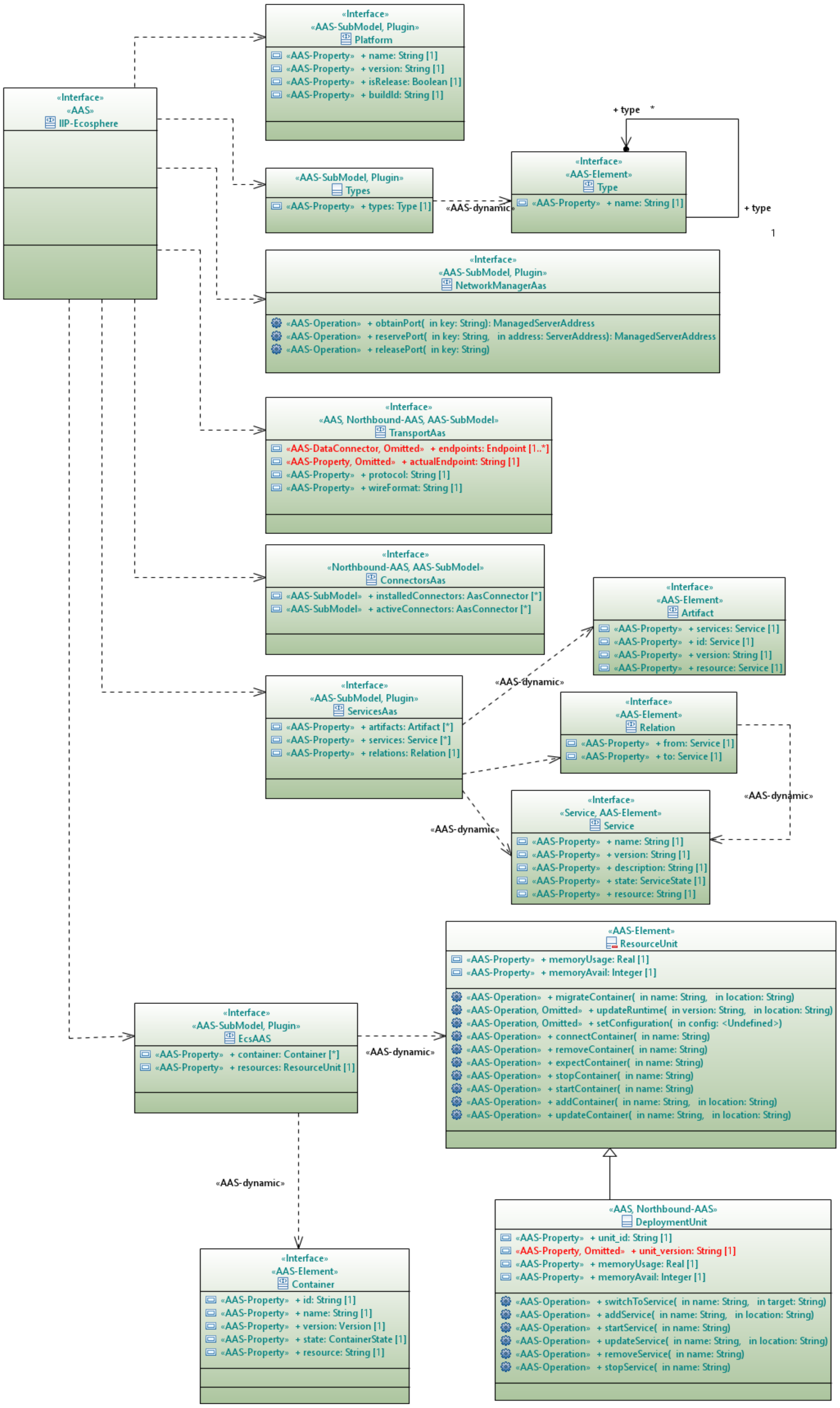

«Interface» «AAS» IIP-Ecosphere
«Interface» «AAS-SubModel, Plugin» Platform
«AAS-Property» + name: String [1]
«AAS-Property» + version: String [1]
«AAS-Property» + isRelease: Boolean [1]
«AAS-Property» + buildId: String [1]
«AAS-SubModel, Plugin» Types
«AAS-Property» + types: Type [1]
«AAS-dynamic»
«Interface» «AAS-Element» Type
«AAS-Property» + name: String [1]
+ type *
+ type 1
«Interface» «AAS-SubModel, Plugin» NetworkManagerAas
«AAS-Operation» + obtainPort(in key: String): ManagedServerAddress
«AAS-Operation» + reservePort(in key: String, in address: ServerAddress): ManagedServerAddress
«AAS-Operation» + releasePort(in key: String)
«Interface» «AAS, Northbound-AAS, AAS-SubModel» TransportAas
«AAS-DataConnector, Omitted» + endpoints: Endpoint [1..*]
«AAS-Property, Omitted» + actualEndpoint: String [1]
«AAS-Property» + protocol: String [1]
«AAS-Property» + wireFormat: String [1]
«Interface» «Northbound-AAS, AAS-SubModel» ConnectorsAas
«AAS-SubModel» + installedConnectors: AasConnector [*]
«AAS-SubModel» + activeConnectors: AasConnector [*]
«Interface» «AAS-Element» Artifact
«AAS-Property» + services: Service [1]
«AAS-Property» + id: Service [1]
«AAS-Property» + version: String [1]
«AAS-Property» + resource: Service [1]
«AAS-dynamic»
«Interface» «AAS-SubModel, Plugin» ServicesAas
«AAS-Property» + artifacts: Artifact [*]
«AAS-Property» + services: Service [*]
«AAS-Property» + relations: Relation [1]
«Interface» «AAS-Element» Relation
«AAS-Property» + from: Service [1]
«AAS-Property» + to: Service [1]
«AAS-dynamic»
«AAS-dynamic»
«Interface» «Service, AAS-Element» Service
«AAS-Property» + name: String [1]
«AAS-Property» + version: String [1]
«AAS-Property» + description: String [1]
«AAS-Property» + state: ServiceState [1]
«AAS-Property» + resource: String [1]
«AAS-Element» ResourceUnit
«AAS-Property» + memoryUsage: Real [1]
«AAS-Property» + memoryAvail: Integer [1]
«AAS-Operation» + migrateContainer(in name: String, in location: String)
«AAS-Operation, Omitted» + updateRuntime(in version: String, in location: String)
«AAS-Operation, Omitted» + setConfiguration(in config: <Undefined>)
«AAS-Operation» + connectContainer(in name: String)
«AAS-Operation» + removeContainer(in name: String)
«AAS-Operation» + expectContainer(in name: String)
«AAS-Operation» + stopContainer(in name: String)
«AAS-Operation» + startContainer(in name: String)
«AAS-Operation» + addContainer(in name: String, in location: String)
«AAS-Operation» + updateContainer(in name: String, in location: String)
«Interface» «AAS-SubModel, Plugin» EcsAAS
«AAS-Property» + container: Container [*]
«AAS-Property» + resources: ResourceUnit [1]
«AAS-dynamic»
«AAS-dynamic»
«Interface» «AAS-Element» Container
«AAS-Property» + id: String [1]
«AAS-Property» + name: String [1]
«AAS-Property» + version: Version [1]
«AAS-Property» + state: ContainerState [1]
«AAS-Property» + resource: String [1]
«AAS, Northbound-AAS» DeploymentUnit
«AAS-Property» + unit_id: String [1]
«AAS-Property, Omitted» + unit_version: String [1]
«AAS-Property» + memoryUsage: Real [1]
«AAS-Property» + memoryAvail: Integer [1]
«AAS-Operation» + switchToService(in name: String, in target: String)
«AAS-Operation» + addService(in name: String, in location: String)
«AAS-Operation» + startService(in name: String)
«AAS-Operation» + updateService(in name: String, in location: String)
«AAS-Operation» + removeService(in name: String)
«AAS-Operation» + stopService(in name: String)

Figure 39: AAS structure of the IIP-Ecosphere platform (preliminary, incomplete)

As the IIP-Ecosphere AAS is rather dynamic, we can already draw some conclusions on lessons learned with BaSyx (based on the integrated version through the support layer):

- Remotely deployed AAS with operations and properties realized in terms of attached functors typically require uniquely serializable functor objects, i.e., they do not work with simple lambda functors or serializable lambda functors.
- When obtaining a remotely deployed AAS, the AAS is turned into a serialized format as already briefly mentioned in Section 3.7.3, i.e., all functors such as getters, setters or operations are serialized, to obtain the values of the properties the getters are even executed. If getters are bound to an AAS implementation server, that server must be ready to serve connections at the point in time when the remote AAS is requested (which may happen in parallel initiated by other components) and currently for each property a network connection is created by the respective BaSyx connector and the value is requested. This seriously affects the performance of obtaining and using a remote AAS. It happened to us that in such a situation a potentially endless loop occurred forcing us to re-think a rather obvious implementation approach in terms of getter functors. As discussed in Section 3.7.3, we suggest using functors that map to local data rather than to remote data. The local data object may be updated in parallel through a different process, e.g., a Transport Layer connector. Dependent on the implementation, each serialized AAS then has its own remote data object, leading to a distributed setup of AAS that can be kept up to date via Transport Layer mechanisms. Directly writing values into an AAS might be an alternative, but in the remote deployment case, the serialized AAS implicitly performs update requests on the original remote AAS, i.e., probably leading to reduced performance.
- When writing larger portions of structured data, in particular binary data, there is a conversion problem in the BaSyx version that we are using. Types like `Base64Binary` are not handled correctly. Currently, we encode such data through a Base64 String encoder.
- The IIP-Ecosphere abstraction appears to be easier to use and requires less code than plain BaSyx [2], but this was a design goal. Moreover, the AAS implementation can be replaced seamlessly, also by a non-AAS interface realization.

6 Platform Configuration Model

This section provides an overview on the IVML configuration model and the concepts used to model configuration options for the IIP-Ecosphere platform. In essence, the configuration model mirrors the component hierarchy of IIP-Ecosphere and describes per component the configurable elements, their dependencies and constraints. IVML is the Integrated Variability Modeling Language [7] as realized by the EASy-Producer toolset [31]. The configuration model consists of three parts:

1. The **configuration meta-model** introducing the configurable elements, their structure, relations, properties and where adequate also consistency constraints.
2. A **platform configuration** based on the configuration model describing the configuration of a certain platform installation. Platform-specific structures (like services, service dependencies and service relations to form an application), but also the specific selection of alternative components, e.g., various transport protocols, service execution environments, container managers, are defined in the platform configuration. A platform configuration may introduce further, application/installation specific constraints.
3. A **valid platform configuration** complies with the configuration meta-model and fulfills all constraints. Such a valid platform configuration can be instantiated through an instantiation model, consisting of an instantiation process description (VIL, variability implementation language) and, where adequate, artifact instantiation templates (VTL, variability template language) [9]. In IIP-Ecosphere, both languages are used to instantiate a platform configuration into code and build specification artifacts, to execute and to package the created artifacts.
4. VIL and VTL can be used at **runtime to adapt the underlying system** [5]. These capabilities will be used in the last project year to allow for self-adaptation of the IIP-Ecosphere platform.

The configuration model is taken up by the configuration component (Section 3.11) and used for platform instantiation and runtime adaptation. The configuration component allows for high-level model operations.

As illustrated in Figure 40, the configuration meta-model reflects the layers and components of the IIP-Ecosphere platform, each given in terms of an IVML project. The most basic project (`MetaConcepts`) introduces even more abstract, i.e., meta-meta, concepts for generic adaptive software systems. These concepts are refined into IIP-Ecosphere specific concepts in the remaining models. The first IIP-Ecosphere specific model describes the `DataTypes` used in the platform, in particular `PrimitiveType` and `RecordType` consisting of files of `DataType` instances. Some specific primitive types are defined in this model and frozen[79] already on that level. The remaining levels will be described as soon as they are realized.

The platform instantiation takes up the data types and turns them into language-specific artifacts, e.g., Java or Python classes. Similarly, corresponding serialization mechanisms to be used with the `Transport` component are generated. So far, there are no basic settings for the `Connectors`.

[79] Frozen elements cannot be modified outside the defining IVML project. Only frozen elements can be instantiated before runtime, while the remaining elements may be frozen later or remain changeable for runtime adaptation. The `MetaConcepts` model defines mechanisms to conditionally control the freezing and also the `CReversibleProperty`, which explicitly re-defines its value to remain unfrozen.

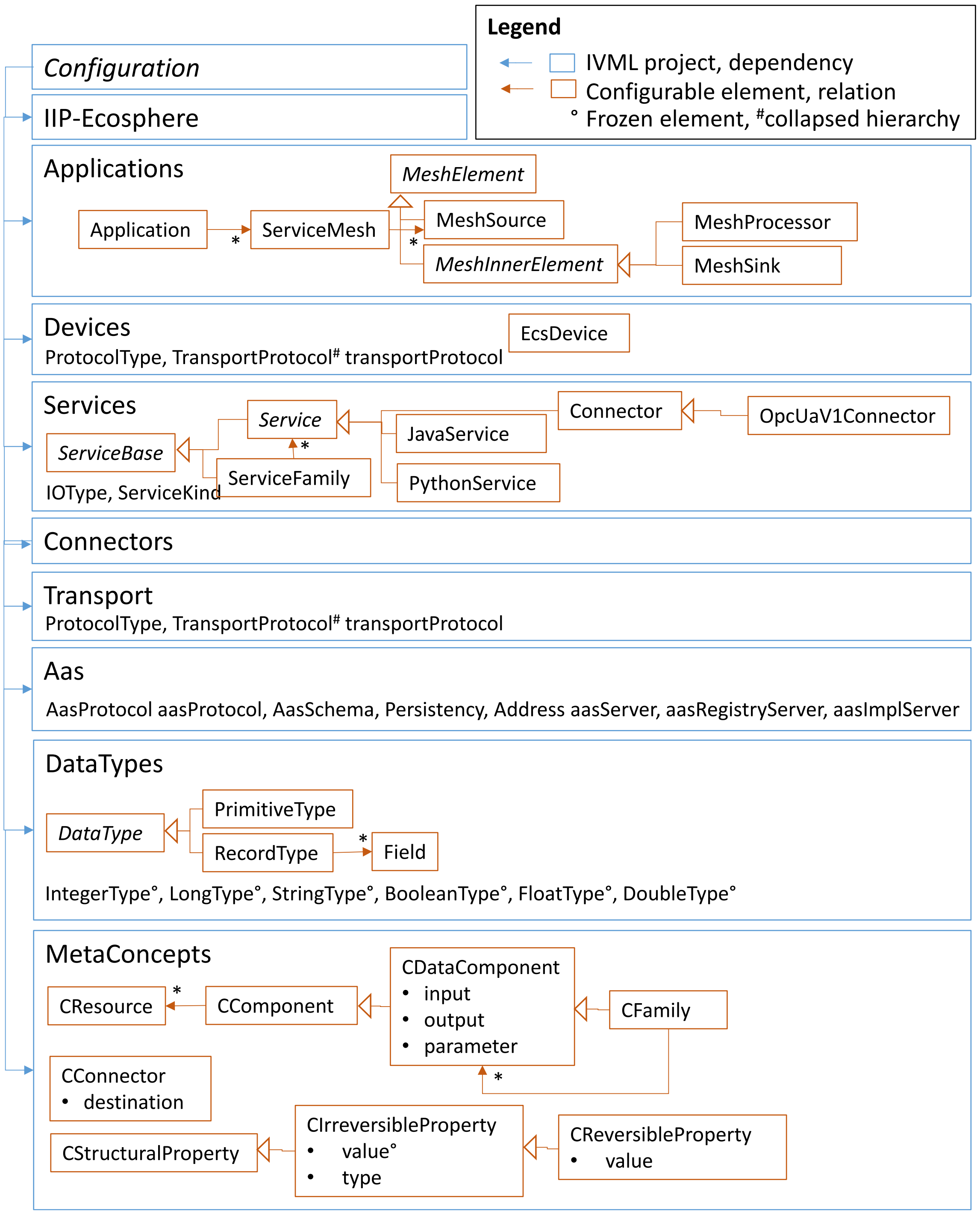


Figure 40: Structure of the IVML IIP-Ecosphere platform metamodel (in development).

On the service level, several refinements of the IIP-Ecosphere service term are defined as configurable elements. The `ServiceBase` is abstract and contains information common to all services, e.g., name, id, version, description, input types, output types, service kind or operation mode (synchronous/asynchronous). Already the `ServiceBase` defines constraints prescribing which information must be present for which kind of service. Although we might use the service kind as hierarchy discriminator here, we opted for building a hierarchy along the implementation levels rather than the service kinds, as service kind differences can easily be handled by constraints while the implementation type is more important for the subsequent code generation. A `Service` is a refinement of `ServiceBase` and also the parent of language specific services like `JavaService` (e.g., detailed by a Java qualified class name denoting the implementation) or `PythonService`. A special kind of Service is a machine/platform Connector, representing the specific connectors implemented in the Connector component (only OPC-UA is shown here, similar elements exist for AAS, MQTTv3, MQTTv5 and AMQP). A `ServiceFamily` represents multiple, alternative but functionally equivalent services with the same input/output types. Service families steer the selection of alternative services at runtime. Although strange at a glance, a `ServiceFamily` (representing a concrete selection of one out of many services) is defined as a kind of service (it inherits from `ServiceBase`). This allows to transparently use a `ServiceFamily` wherever a Service can be used. From the configured services, the code generation derives implementation interfaces (Java, Python) and service stubs (Java) for the integration of non-Java service implementations.

The `Devices` module defines the properties of the ECS runtime, in particular the container manager to use. Moreover, it defines the `EcsDevice`, which represents an installed/connected device. In the next release we plan that `EcsDevice` instances steer the automated creation of Docker containers as well as the automated and optimized assignment of containers to resources.

The `Applications` module introduces one or multiple applications consisting of one or multiple `ServiceMesh` instances. A `ServiceMesh` is a directed graph rooted by sources, linked by connectors/relations possibly leading to sinks. Each node in such a graph has an implementation in terms of a `ServiceBase`. Service properties are pulled up during model validation and allow for checking whether a service graph is valid (through correctly sequenced input/output types of the services). During code generation, individual applications or alternatively all applications are processed, i.e., the service meshes are traversed and stream engine glue code for each node is generated. In the default case, Java classes with Spring Cloud Stream annotations are created and bound to the respective service interfaces. Based on the given implementation class names, the implementing services are dynamically instantiated, mapped into the respective AAS (via the `ServiceMapper` from the service environment) and made available for monitoring and management.

Besides code artifacts also build specifications (Maven), assembly specifications, Spring application specifications, deployment descriptors, logging setting files, JSL specifications and, partially, test classes (for validating generated Yaml files) are created automatically. For the three major platform components, the platform AAS server (based on the `platform` component discussed in Section 3.13, currently without further services), the ECS runtime and the service manager, the basic AAS settings as well as further settings are instantiated into respective Yaml application specification. Finally, the generated build specifications are executed so that for a complete instantiation, three platform artifacts and one combined Java/Python artifact per application is generated.

We do not provide a more detailed discussion of the concepts in the meta-model or the instantiation process at this point in time because both models are still in development and usually it is not expected that users of the platform modify the models. However, as long as there is no user interface, a user must be able to describe a platform configuration in order to perform an instantiation. Therefore, we briefly provide an insight into a simple testing model.

```
project SimpleMesh {

    import IIPEcosphere;

    // binding annotation omitted

    // ------------ component setup ------------------

    serializer = Serializer::Json;
    // serviceManager, containerManager are already defined

    aasServer = {
        schema = AasSchema::HTTP,
        port = 9001,
        host = "127.0.0.1"
    };

    // ...

    // ------------ data types ------------------

    RecordType rec1 = {
        name = "Rec1",
        fields = {
            Field {
                name = "intField",
                type = refBy(IntegerType)
            }, Field {
                name = "stringField",
                type = refBy(StringType)
            }
        }
    };

    // ...

    // ------------ individual, reusable services ------------------

    Service mySourceService = JavaService {
        id = "SimpleSource",
        name = "Simple Data Source",
        description = "",
        ver = "0.1.0",
        deployable = true,
        asynchronous = true,
        class =
          "de.iip_ecosphere.platform.test.apps.serviceImpl.SimpleSourceImpl",
        artifact = "de.iip-ecosphere.platform:apps.ServiceImpl:" + iipVer,
        kind = ServiceKind::SOURCE_SERVICE,
        output = {{type=refBy(rec1)}}
    };
```

Figure 41: First part of a simple platform configuration.

Figure 41 depicts the first part of a simple platform configuration used for testing. A model is defined in terms of IVML, a textual DSL for variability modeling. Each model is surrounded by a project namespace, here named SimpleMesh. Within that namespace, first model imports are stated, here an import of the IIP-Ecospere configuration meta-model (IIPEcosphere). After this header, the first configuration value definitions are stated, typically as value assignments to typed variables (a typed variable indicates a configuration option in IVML). Typed variables can form complex types that we call compounds in IVML. Here, the serializer is defined to be Json, an enumeration literal for serializers defined in the meta-model. Then the global aasServer receives its schema, port number and host name (similarly but not shown for AAS registry and local AAS implementation server). Next, we define the application datatypes, typically records.

While the variables discussed before are pre-defined by the meta-model, the data type is now given in terms of an own variable named rec1 of type RecordType (defined in the meta-model as a compound, not illustrated here). A record has a name (turned e.g., into a Java class name during instantiation) and field, each with a name and a type. Types are references (stated by refBy), i.e., we define a link to an already defined variable, here the pre-defined Integer and String type.

Following the definition oft he variable rec1, we then introduce a Java service, a hand-crafted data source (for testing, it will create arbitrary data of type rec1). The source is described by its identification, its name, an empty description, a version, whether it is deployable, whether it is a synchronous or asynchronous service and its implementation class located in the given Maven artifact. Please note that we use here the implementation version of the platform defined by the meta-model in the variable iipVer. The service is a source service (one of the four main service kinds) and its output is constituted by one record, namely rec1. In fact, multiple types can be given, all in terms of a structured type currently just having a type field (to be extended later), therefore the double brackets, the outer one for a collection instance, the inner one for the structure type.

```
// ------------ application and service nets ------------------

Application myApp = {
    id = "SimpleMeshApp",
    name = "Simple Mesh Testing App",
    ver = "0.1.0",
    description = "",
    services = {refBy(myMesh)}
};

ServiceMesh myMesh = {
    description = "initial service net",
    sources = {refBy(mySource)}
};

MeshSource mySource = {
    impl = refBy(mySourceService),
    next = {refBy(myConnMySourceMyReceiver)}
};

MeshConnector myConnMySourceMyReceiver = {
    name = "Source->Receiver",
    next = refBy(myReceiver)
};

MeshSink myReceiver = {
    impl = refBy(myReceiverService)
};
```

Figure 42: Second part of the simple platform configuration.

The second part of the example in Figure 42 defines an application with a simple service mesh. First an application is defined, again with identification, name, version and empty description. Then the service meshes are stated, here a single reference to `myMesh`. `myMesh` potentially consists of multiple sources, we just have `mySource` as source mesh element. `mySource` uses the previously defined `mySourceService` as implementation, as well as the next mesh element in terms of a mesh connector/relation. A synchronous source may also define a polling interval. Currently, mesh connectors have just a name but further properties may follow (otherwise we could directly reference mesh elements among each other). The mesh connector links further to the receiver, which states its implementation as `myReceiverService` (similar to `mySourceService` but not shown here).

```
    freeze {
        aasServer;
        serializer;
        // ...
        .;
    };

}
```

Figure 43: Final part of the simple platform configuration.

The final part is important for the instantiation. For various reasons, variable values defined in IVML are not per se considered final, rather they can be overwritten in importing modules/project. Turning such a configuration into code is problematic, in particular if code parts are deleted based on non-final decision (deleted parts are usually deleted). Thus, IVML has the notion of freezing variables. Frozen variables are considered final and can be instantiated safely. Figure 43 illustrates the freezing of this model. Within the freeze block, first variables from the meta-model that have been configured are listed for freezing. Finally, every variable declared in this project (shortcut "." like in a command shell) is frozen. Typically in systems with dynamic instantiation at runtime, freezing is conditional, i.e., stated variables are filtered according to a given condition. In the original model used for testing, this condition is based on the so-called binding time, the latest time when a decision must be made (here compile time). As we just aimed at explaining how a platform configuration looks like, we intentionally left out the required attachment of binding times at the beginning of the model and the freeze condition here. Ultimately, Figure 43 ends with the closing bracket for the namespace of the `SimpleMesh` project.

Although the configuration shown here looks pretty structural and might be represented in any other nested configuration language, we did not detail the validation constraints that are imposed by the meta-model, e.g., that services are configured correctly and services meshes fit together. For now, the constraint setup is initial and several constraints are currently missing. However, the already defined constraints can quickly lead to validation errors issued by the EASy-Producer reasoner. This validation is important, as an invalid model typically leads to invalid artifacts that, e.g., cannot be compiled. Work is still needed here to make the validation messages more domain-specific and user friendly.

In summary, the code generation based on the IIP-Ecosphere configuration model creates 8 different types of artifacts (Maven XML, assembly XML, Java source, Python source, application Yaml, logging XML, Java test code, windows batch/linux shell startup scripts), which leads to different types of artifact structures, e.g., various forms of Java code. The number of generated artifacts varies with the number of services/mesh elements defined per application/platform configuration.

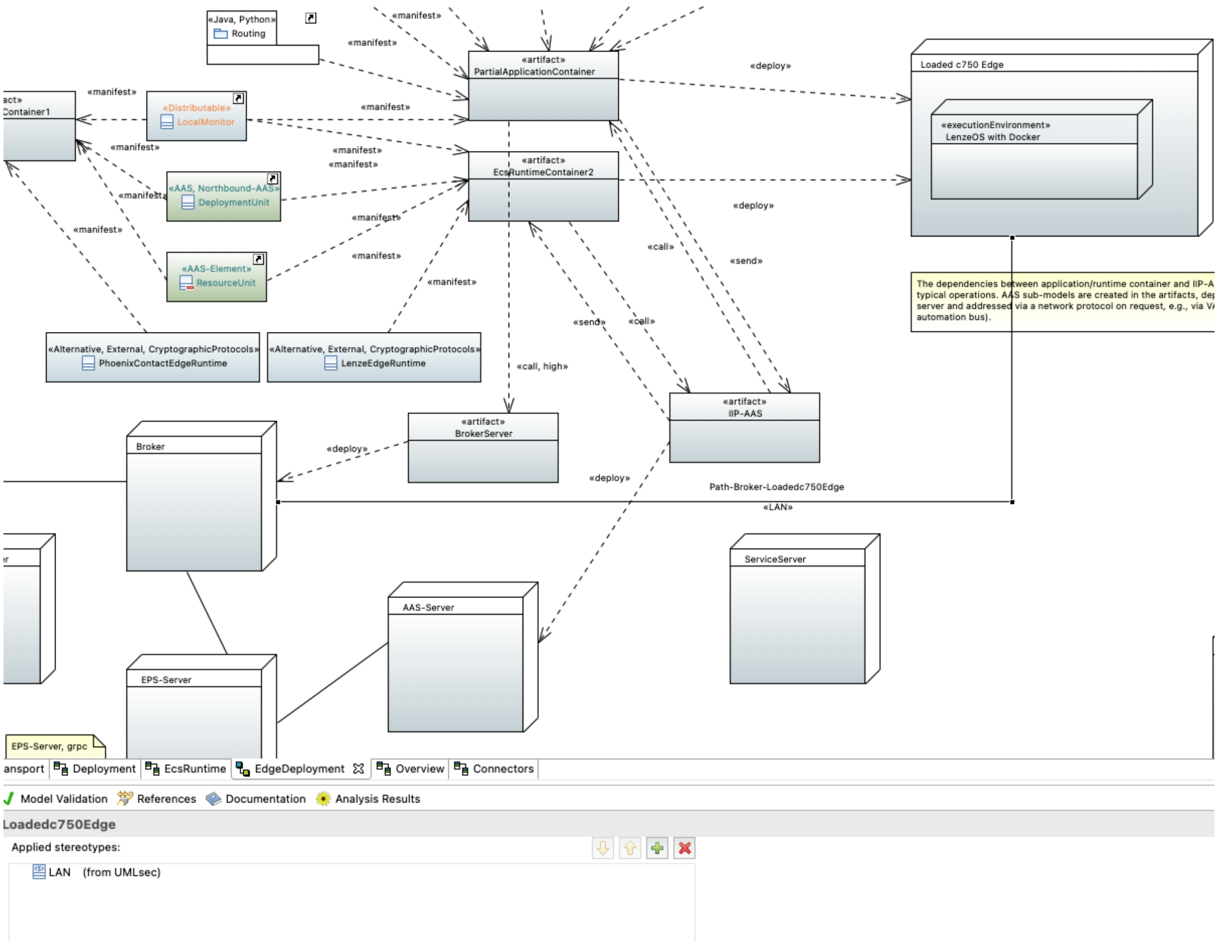
7 Platform Security and Data Protection

In this section, we discuss means to ensure the security and the data protection in the IIP-Ecosphere platform. We start with (cross-cutting) internal security and security analysis in Section 7.1 and external sercurity measures in 7.2.

7.1 Internal Security and Security/Privacy Analysis

One main step before managing security and offering security enhanced services is to review where in fact security is needed. Moreover, concerning the General Data Protection Regulation (GDPR), security and privacy aspects must be considered as early as possible in the design and development of a system (privacy and security by design principles). Architectural models, in fact, offer an excellent possibility to support the realization of privacy and security by design principles.

In Section 3.4.2 we introduced a UML profile called UMLsec. We further introduced two privacy checks secure links and secure dependency. Such checks provide a possibility to perform security and privacy checks on the design (architecture) of a platform provided using UML models.



The dependencies between application/runtime container and IIP-A
typical operations. AAS sub-models are created in the artifacts, dep
server and addressed via a network protocol on request, e.g., via VA
automation bus).

Loadedc750Edge

Applied stereotypes:

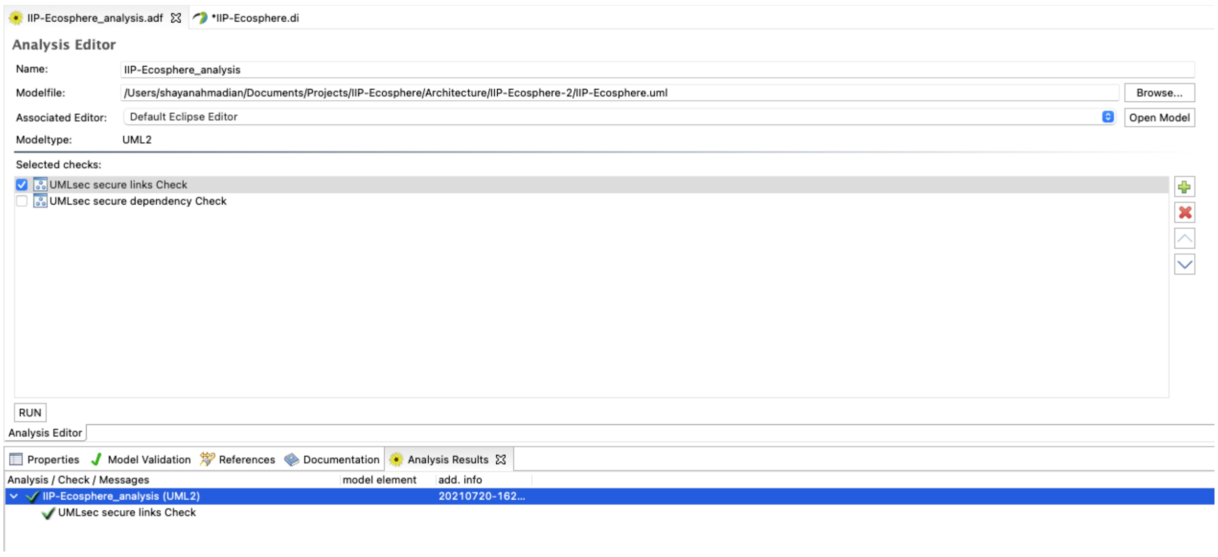
LAN (from UMLsec)

Figure 44: Architecture model for edge deployment annotated with secure links stereotypes (excerpt of Figure 34).

The process of checking and enhancing an architecture model is the topic of our ongoing research. In this section we describe how we can analyze the architecture of the IIP-Ecosphere data platform using CARiSMA.

In Section 3.4.2.1, we saw that the *secure links* annotation enables one to ensure the security of communications in a physical layer. The following model is annotated with stereotypes relevant to secure links (Figure 44).

The link between the node `Loaded c750 Edge` and the node `Broker` in this figure is annotated with the `Lan` stereotype. The stereotypes as shown in the lower side of the figure can be set in properties view. In Figure 16 we saw that a default adversary cannot delete, read or insert on a link annotated with the `Lan` stereotype. Furthermore, the dependency between the two artifacts deployed on these two nodes namely, `BrokerServer` and `PartialApplicationContainer` is high (as indicated by the «`high`» annotation), requiring that the adversary cannot read, delete or insert on the link. Concerning the fact that the link is annotated with the `Lan` stereotype, after performing an analysis the check should not show any problems. This is in fact true, shown in Figure 45 which demonstrate the results of the analysis.



Analysis Editor

Name: IIP-Ecosphere_analysis

Modelfile: /Users/shayanahmadian/Documents/Projects/IIP-Ecosphere/Architecture/IIP-Ecosphere-2/IIP-Ecosphere.uml

Associated Editor: Default Eclipse Editor

Modeltype: UML2

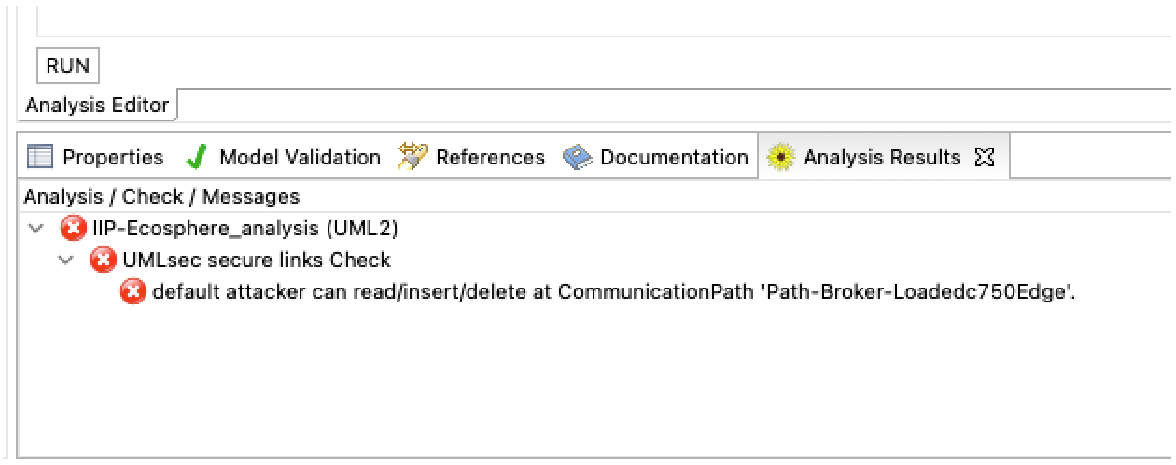
Selected checks:

- [✓] UMLsec secure links Check
- [ ] UMLsec secure dependency Check

```
Analysis / Check / Messages                          model element   add. info
✓ IIP-Ecosphere_analysis (UML2)                                      20210720-162...
   ✓ UMLsec secure links Check
```

Figure 45: The CARiSMA analysis result.

Assuming that the link between the two nodes is annotated with the «`internet`» stereotype, and concerning the fact that the default adversary can read, insert and delete on the communication link, the corresponding error is shown in Figure 46.

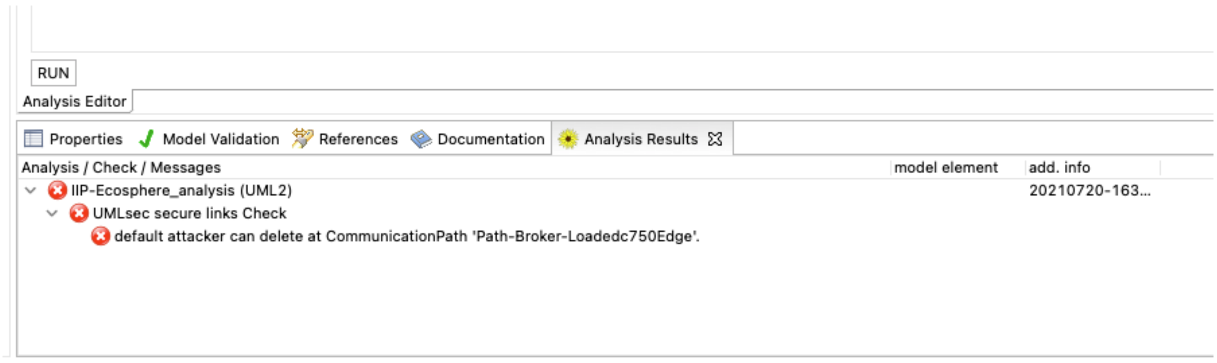


```
Analysis / Check / Messages
IIP-Ecosphere_analysis (UML2)
   UMLsec secure links Check
      default attacker can read/insert/delete at CommunicationPath 'Path-Broker-Loadedc750Edge'.
```

Figure 46: The result of the CARiSMA analysis.

If the communication path “`Path-Broker-Loadedc750Edge`” is annotated with the stereotype «`Encrypted`» and concerning the fact that a default adversary can only delete a message on an encrypted path, the corresponding error after performing a CARiSMA analysis is shown in Figure 47.

Such errors identified in the result of a CARiSMA analysis can inform a security expert or a system designer about potential threat and risks in a system in the very early phases of system design. Therefore, such an analysis facilitates the process of enhancing an architectural model with appropriate security and privacy mechanisms.



```
Analysis / Check / Messages                                              model element   add. info
IIP-Ecosphere_analysis (UML2)                                                            20210720-163...
  UMLsec secure links Check
    default attacker can delete at CommunicationPath 'Path-Broker-Loadedc750Edge'.
```

Figure 47: The result of the CARiSMA analysis.

7.2 External Security

For communicating with other platforms or other instances of the IIP-Ecosphere platform, external communication is required. In particular, external communication requires a certain level of security. Currently, two approaches are intended to support such external communication:

- Concepts and components of the International Data Spaces (IDS) that will partly be integrated into the GAIA-X initiative. We will discuss the plans for IDS support in one of the next versions of this document.
- An alternative, more lightweight approach is to transparently encrypt all communication between two parties linked via the internet. KIPROTECT has demonstrated such an approach in terms of the EPS-System (End Point Server, Figure 48 and Figure 49), which is already being used to secure data transfers between German health departments and contact tracing providers. In particular, it is easy to install, offers end to end encryption and mutual authentication via mTLS and supports encapsulation of arbitrary application-layer protocols like MQTT or REST interfaces. As BaSyx AAS are realized via REST, the partners believe that a transparent communication between two parties to their internal IIP-Ecosphere platform AAS via the EPS-System is possible. Moreover, the EPS-System supports role-based access management that can restrict access to specific services and methods based on group memberships as well as additional criteria, so that the platform communication can be provided in a selective manner to actors in the ecosystem. Further, we believe that an EPS instance could also server as protection mechanism for the platform AAS towards an (internal/external) user interface or other platform layers. Providing an additional level of authentication and access control via the EPS system on top of the existing ones implemented on the AAS level via BaSyx can be part of a good “defense in depth” strategy.

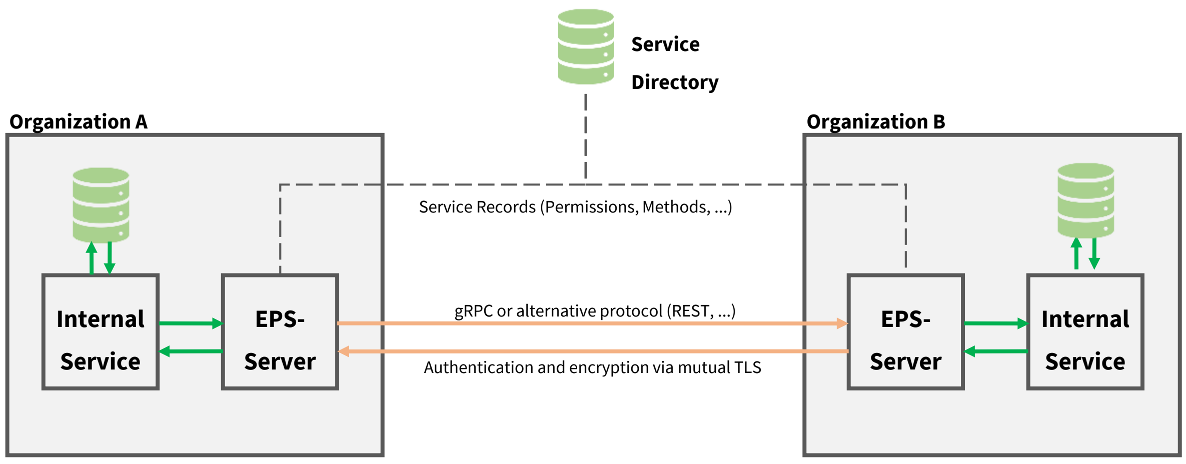

Service
Directory
Organization A
Organization B
Service Records (Permissions, Methods, ...)
Internal Service
EPS-Server
gRPC or alternative protocol (REST, ...)
Authentication and encryption via mutual TLS
EPS-Server
Internal Service

Figure 48: Direct communication between organizations through the EPS system

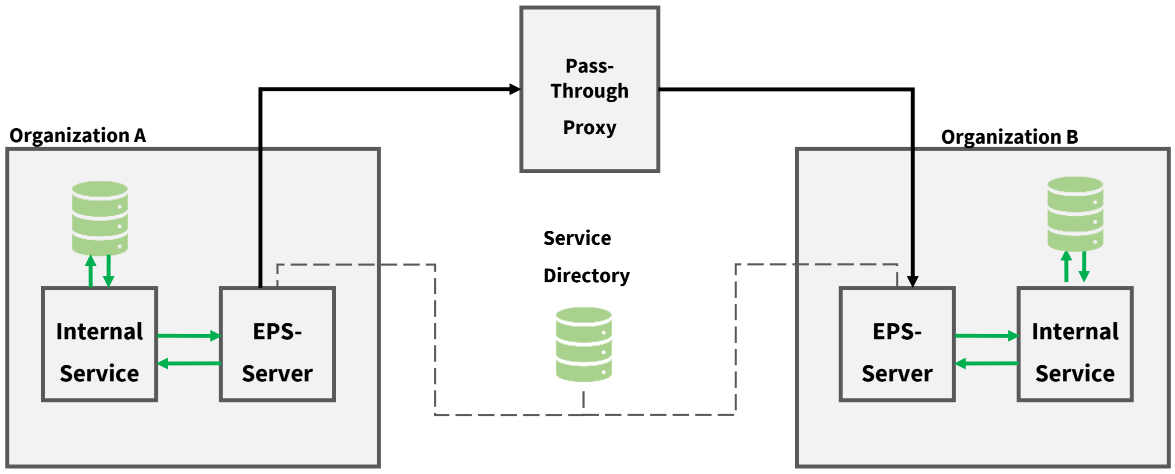


Figure 49: Indirect, proxy-based (end-to-end encrypted) communication between organizations through the EPS system.

8 Implementation

In this section, we briefly discuss aspects of the implementation of the IIP-Ecosphere platform, i.e., decisions we made during the implementation (Section 8.1), how to obtain a binary version (Section 8.2), the dependencies and how to compile the sources (Section 8.3), and how to install and to use the platform (Section 8.4). Section 9 on how-to's will take up some of the topics, but more from the perspective on how to extend or interact more deeply with the platform (code). Intentionally, we do not discuss code here. For this purpose, we refer the reader to the IIP-Ecosphere Github repository[17] and in particular the Markdown[80] readme files that are provided for the platform and for individual components.

8.1 Implementation decisions

We briefly discuss now technical decisions or issues that occurred during the development of the IIP-Ecosphere platform. This list may not be complete[81] and is subject to incremental extension:

- As more parts and pieces show up, e.g., AAS sub-models, the more decisions on the **startup process** of the platform have to be made. However, some of these decisions impact testing, as a full startup including AAS sub-models is not always desired or may even break tests. In these cases, it is possible to mock out the `AasFactory` or to create missing server instances for the platform AAS via the `AasPartRegistry`, both located in the Support Layer (Section 3.5).
- Akin, many components make assumptions on **default instances for alternative components in testing**. Typically, we use AMQP as testing protocol (the server is rather easy to use, runs with JDK 1.8 and the implementation is stable) as well as BaSyx as AAS implementation. It is important to clearly distinguish these dependencies from production code, i.e., they must be part of the testing scope as otherwise they may accidentally become part of the instantiated platform components and clash with the decisions made in the platform configuration.
- BaSyx and Spring use different versions of the **expression language** `javax.el.el-api`, which, when utilized together on the same classpath, prevent Spring Cloud Stream from starting. Wherever possible in installation packages, we try to separate AAS and stream processing, i.e., stream processing components shall run in their own JVMs controlled by a supervisor JVM containing the ECS runtime, which also maintains the representing AAS of the installation part. For uniform technical configuration, it is desirable that the ECS runtime is also started as a Spring application, while use of Spring Cloud Stream shall be prevented in there.
- Different external components depend on **Google Guava** in several versions. As a Guava version below 22 prevents some protocol test cases to be executed, we decided to fix Google Guava to version 22 in the platform dependency management. Similarly, further components may be fixed to rather narrow version ranges in the managed platform dependencies (and transitively in components such as EASy-Producer).
- So far, we use `org.slf4j` for **logging**, as this library is also used by BaSyx and Spring (although in different versions). Logging setup (also called configuration) is typically added during platform instantiation or for testing, also to avoid conflicting setups.
- We added a simple resilience mechanism for **failing connections** to AAS implementation servers. In the version of BaSyx that we are using, implementations of operations, property getters or setters are attached through functors (usually lambda functions) to the AAS. In such a functor, currently the preferred style seems to be to create one connector instance per operation or property call, which builds up a network connection to an AAS implementation

[80] https://de.wikipedia.org/wiki/Markdown
[81] We do not intend to repeat all coding conventions for the platform in this document. We just listed here the most important ones with their rationales as overview. For details, please refer to https://github.com/iip-ecosphere/platform/blob/main/platform/documentation/README.md

server. If the connection fails, e.g., because the AAS implementation server was intentionally shut down, let's say when stopping a service through the service manager, the AAS will continue connecting unsuccessfully to that AAS implementation server. At a glance, this only is an issue if the operation or property is addressed. However, in the BaSyx version used in this release, each access to a remotely deployed AAS causes an execution of all these functors (probably to serialize and transport the respective "value"), leading in some cases to (seemingly) endless trials to connect to the intentionally closed server. Although we delete the respective operation/property from the AAS before shutting down the service or the respective AAS implementation server, respectively, the described behavior occurs. As a mitigation and a first step towards **connection resilience**, the functors attaced by the AAS abstraction for BaSyx track erroneous connections for all connector instances and return a constant value on failures. As this decision is intentionally global for all connector instances, we also have to revert the decision if the server becomes available again or the same address is used in another context. Currently, erroneous connections are disabled by default for a time period of one minute. Later versions of the platform shall integrate this behavior with the port release of the network manager or with a connection trial after a given timeout.

8.2 Obtaining the IIP-Ecosphere platform

The sources of the IIP Ecosphere platform are available on Github[82]. Released binaries of the IIP-Ecosphere platform can be obtained from Maven Central[83]. Snapshots from the continuous integration can be obtained from the SSE Maven repository[84].

However, it is important to keep in mind that the IIP-Ecosphere platform consists of several alternative or optional components that must be consistently configured to obtain a valid installation for a certain setting. We will discuss in Section 8.4 how to utilize the configuration approach to obtain the binaries. Below, we summarize the (optional/alternative) components, the respective location of the configuration settings and the JSL descriptors that can be used to provide extensions. Table 16 summarizes the settings and the provided descriptors.

Table 16: Configuration and extension mechanisms used in the IIP-Ecosphere platform components (for descriptors, we abbreviate "de.iip_ecosphere.platform" by "d.i.p" for formatting reasons)

Layer/ Component	Settings	Provided JSL descriptors
Support	-	*d.i.p*.support.LifecycleDescriptor *d.i.p*.support.aas.AasFactoryDescriptor *d.i.p*.support.aas.ProtocolDescriptor *d.i.p*.support.net.NetworkManagerDescriptor *d.i.p*.support.iip_aas.AasContributor *d.i.p*.support.aas.AasServerRecipeDescriptor
Transport	-	*d.i.p*.transport. TransportFactoryDescriptor
Connectors	-	*d.i.p*.connectors.ConnectorDescriptor
Services	iipecosphere.yml	*d.i.p*.services.ServiceFactoryDescriptor
Resources / Monitoring	iipecosphere.yml	*d.i.p*.ecsRuntime.EcsFactoryDescriptor
Configuration	iipecosphere.yml	-
Platform	iipecosphere.yml	-

[82] https://github.com/iip-ecosphere/platform/
[83] https://repo1.maven.org/maven2/de/iip-ecosphere/platform/
[84] https://projects.sse.uni-hildesheim.de/qm/maven/de/iip-ecosphere/platform/

The *Support Component* does not take specific settings into account rather being set up through upper platform layers/components. In contrast, the Support component defines several fundamental JSL descriptors to allow the upstream platform components to hook into at defined points or to allow for external extensions. We summarize the descriptors now and link them to the variability provided by the platform and the platform configuration approach. The descriptors are:

- `LifecycleDescriptor` with allows adding components to the startup/shutdown process of a platform component. These descriptors can indicate a certain startup level and they even can cause a shutdown of a platform component. Adding certain descriptors to a platform binary causes the respective components to be started. Upper platform components ship with their descriptor file (in `META-INF/services`) so that either we add a certain component or a descriptor to the instantiated platform binaries (positive variability) becomes then active. In seldom cases, we may add the component and remove the descriptor to disable the respective registration (negative variability).
- `AasFactoryDescriptor` indicating the AAS factory to be used. A specific descriptor is shipped with the AAS (abstraction) implementation. The default implementation is `support.aas.basyx`. The platform just takes the first available descriptor (excluding potential descriptors used in testing), allowing here only for a single choice variability. By including a certain AAS implementation component, i.e., adding it to the platform classpath, the descriptor is made available and the respective factory becomes active (positive variability) as done during platform instantiation.
- `ProtocolDescriptor` is an optional extension descriptor indicating AAS implementation protocols that are not shipped with the platform. By default, TCP and HTTP/REST protocols for the BaSyx Virtual Automation Bus are provided, but other protocols may be desired in a certain installation. Here, additional external components can add arbitrary protocols (positive, unlimited variability) as long as the protocol names are unique. New protocols must be added to the configuration model as potential alternative so that the selected/desired protocol can be specified while instantiating the settings of the upper components.
- `NetworkManagerDescriptor` is an optional descriptor that indicates which network manager shall be used by a component. The Support component does not ship with any descriptor information so that the platform instantiation must provide respective files (in `META-INF/services`). One alternative to the local manager is a global AAS-based network manager for some ports backed by a local network manager.
- `AasContributor` is an extension descriptor for higher level platform components to conveniently build a common AAS for the platform. AAS contributors define specific sub-models and announce their presence through the `AasContributor` descriptor, i.e., any descriptor found will be used to set up the common AAS for the platform. Examples in the Support Component are the platform “nameplate” sub-model and the network manager AAS (providing access to the network manager selected by the `NetworkManagerDescriptor`). Although the descriptors shipped with the platform are intended to be executed, the platform instantiation may add or remove specific contributors to customize the AAS of a certain component. Moreover, the descriptors can declare themselves as invalid as, e.g., instances required to implement the AAS are not present.
- `AasServerRecipeDescriptor` defines the specific recipe to be used when creating an AAS server. The AAS abstraction defines a local server recipe for in-memory storage. However, on a server sided installation, also a persistent storage of the AAS may be required, which can lead to a large set of dependencies and unnecessary allocation of resources on edge devices. The required behavior, storage options but also dependencies can be defined by a specific AAS server component.

The *Transport Component* in the Transport and Connectors layer does not define own settings. This is done by the `TransportFactoryDescriptor` to allow concrete transport protocol implementations to hook themselves into the `TransportFactory`. Moreover, it offers adding serializer implementations to the `SerializerRegistry`. For a concrete application, the respective serializers are created during platform instantiation and registered through generated code in the `SerializerRegistry`.

The *Connectors Component* in the Transport and Connectors layer defines the `ConnectorDescriptor` for announcing available descriptors that can be used / shown up in the AAS sub-model of the Connectors Component.

The *Services Component* takes setup information from a unified YAML file called `iipecosphere.yml`, which must be present on the classpath of the component[85]. This file is instantiated through the configuration model during platform instantiation and added to the respective instantiated components. Depending on the service manager to be used, specific setup information may be required, e.g., for Spring Cloud Streams the full breath of the used Spring Components can be configured in this file[86]). Moreover, the Services component defines the `ServiceFactoryDescriptor` that announces the actual `ServiceManager` to be used.

Similarly, the *ECS Runtime Component* in the Resources and Monitoring Layer utilizes own entries in its `iipecosphere.yml` file and provides an own descriptor (`EcsFactoryDescriptor`) to announce the configured container manager.

In the Configuration Layer, the *Configuration Component* considers specific settings in its `iipecosphere.yml` file, e.g., where to find the configuration meta-model, the platform configuration, where to write instantiated components to etc. As the configuration component will offer own operations to modify the configuration, it also utilizes the descriptors defined by other components/layers, e.g., the `AasContributor`, to hook itself into the platform mechanism to create a joint platform AAS.

The *Platform Component* is a collection of the basic services to be started, in particular a (persistent) AAS server or a global network manager. Thus, it requires specific setup information in its `iipecosphere.yml`, e.g., on which port and using which implementation protocol the global platform AAS shall be set up (the individual AAS are then remotely deployed into this AAS server).

Besides the services and their technical network addresses, the platform also uses some pre-defined Transport Layer channels. These channels are briefly summarized in Table 17. It is important that channels are independent of the transport protocol, i.e., apply equally to, e.g., MQTT or AMQP. Moreover, the default metrics channels currently use a fixed JSON format and rely on a default String serializer defined in the Transport Layer. The service channels use an application-specific format determined by the active serializer and the code generation of the platform instantiation process.

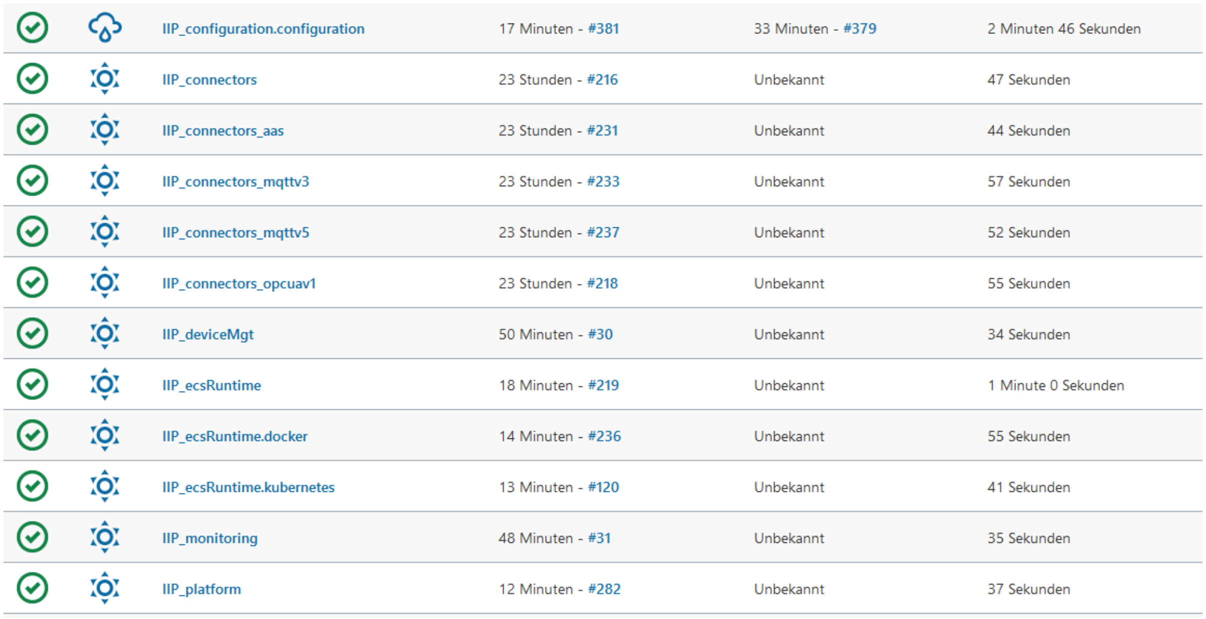
[85] The detailed settings are documented in the `README.MD` file of the respective components.
[86] In Spring applications, this file is typically called `application.yml`. The name for the IIP-Ecosphere platform is different, also as Spring is only used in alternative components.

Table 17: Transport Channels used by the platform

Channel	Kind	Component	Format	Explanation
EcsMetrics	global	ECS Runtime	JSON	Metrics reporting by ECS Runtime
ServiceMetrics	global	Service Mgt., Service Env.	JSON	Metrics reporting by Services, can be augmented by application-specific metrics.
<service>_<function>	global	Application	*application specific*	Inter-device transport channels per service and function[87].
<service>_<function>	local	Application	*application specific*	Intra-device transport channels per service and function[87].

8.3 Compiling the IIP-Ecosphere platform

Due to the various optional and alternative components in the IIP-Ecosphere platform that we manage in individual artifacts/Eclipse projects, compiling the IIP-Ecosphere platform is not trivial. As mentioned above, for the project the SSE Continuous Integration (CI) server as shown in Figure 50, knows about all the build dependencies among the components and builds the parts and pieces along the dependency tree when the code of a single component changes. As part of building, it executes the respective component tests, assembles the documentation and, if successful, deploys the respective snapshots to the SSE Maven repository or the stable releases from Maven (or related repositories).



✓	☁	IIP_configuration.configuration	17 Minuten - #381	33 Minuten - #379	2 Minuten 46 Sekunden
✓	☀	IIP_connectors	23 Stunden - #216	Unbekannt	47 Sekunden
✓	☀	IIP_connectors_aas	23 Stunden - #231	Unbekannt	44 Sekunden
✓	☀	IIP_connectors_mqttv3	23 Stunden - #233	Unbekannt	57 Sekunden
✓	☀	IIP_connectors_mqttv5	23 Stunden - #237	Unbekannt	52 Sekunden
✓	☀	IIP_connectors_opcuav1	23 Stunden - #218	Unbekannt	55 Sekunden
✓	☀	IIP_deviceMgt	50 Minuten - #30	Unbekannt	34 Sekunden
✓	☀	IIP_ecsRuntime	18 Minuten - #219	Unbekannt	1 Minute 0 Sekunden
✓	☀	IIP_ecsRuntime.docker	14 Minuten - #236	Unbekannt	55 Sekunden
✓	☀	IIP_ecsRuntime.kubernetes	13 Minuten - #120	Unbekannt	41 Sekunden
✓	☀	IIP_monitoring	48 Minuten - #31	Unbekannt	35 Sekunden
✓	☀	IIP_platform	12 Minuten - #282	Unbekannt	37 Sekunden

Figure 50: Screenshot of the SSE Continuous Integration server (IIP-Ecosphere view, cropped)

For completeness, we discuss below the dependencies among the individual components of the IIP-Ecosphere platform (as illustrated in Figure 51). The `platformDependencies` project collects the dependency information of (optional or required) external components that are used by at least one component and do not constitute singleton wrapped components (cf. Section 4). In other words, the `platformDependencies` project defines the managed dependencies of the platform with their respective version number (range) but without actually using them. The dependent components rely on this information and just state the required components without replicating their version numbers (Maven parent POM mechanism). As we usually do not build external components, e.g., protocols, rather than relying on available release binaries, these dependencies are out of scope.

[87] An application identifier will be added in one of the next releases.

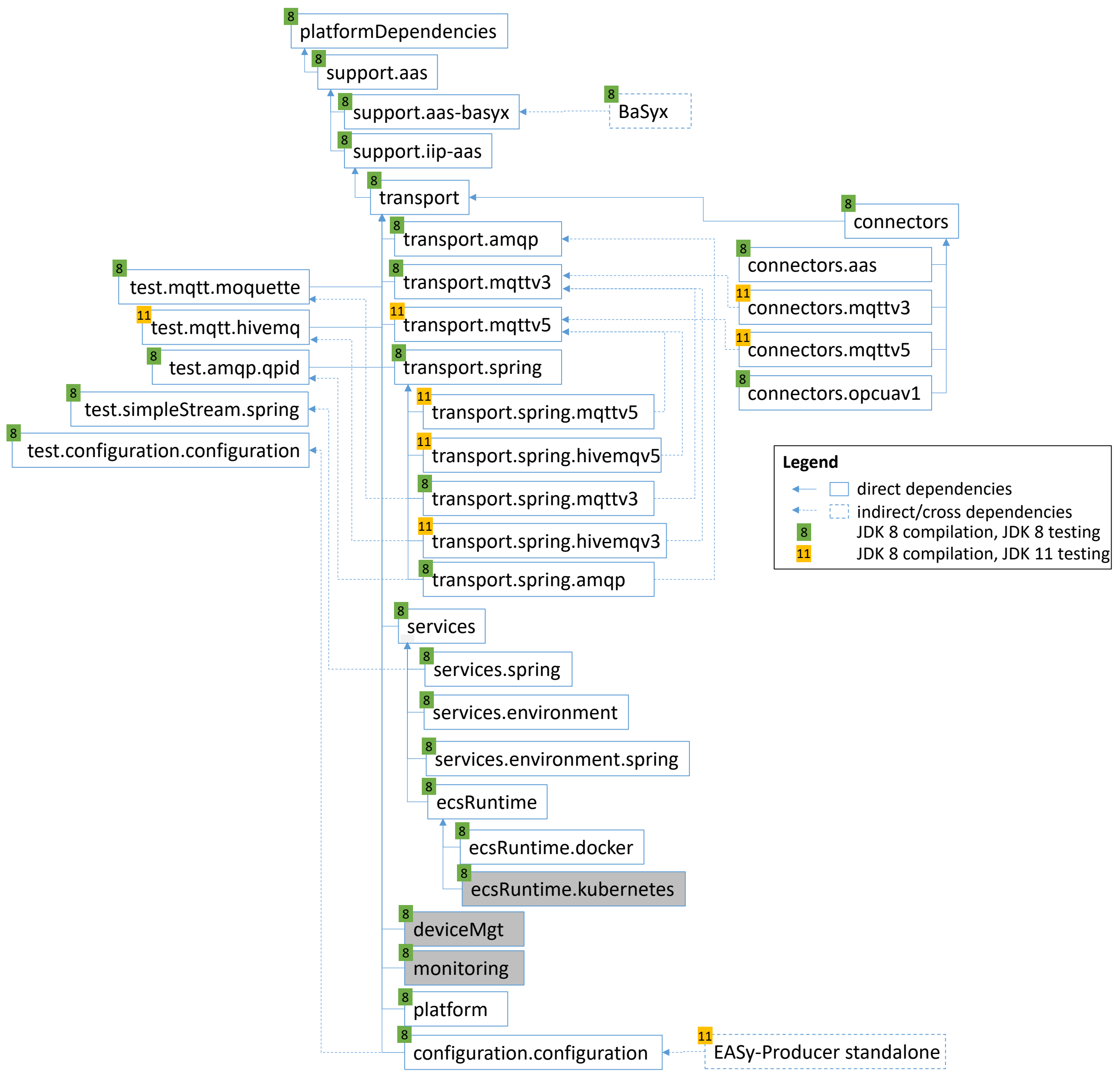


Figure 51: Dependencies among the components (folder names in github)

The Support Component (`support.aas`) is the most basic IIP-Ecosphere component without further dependencies to the platform. The BaSyx default implementation and the `iip-aas` support functions depend directly on `support.aas` and are build when `support.aas` changes.

While all platform components receive their dependencies from Maven central as managed dependencies, the BaSyx library (and the dependent `support.aas.basyx`) is an exception, because BaSyx has so far not been deployed as versioned binary component. Moreover, the Github development of BaSyx is done without release tags, i.e., so far there are no release versions of BaSyx. To rely on a safe version that is common for the development in IIP-Ecosphere, we decided to take a feasible snapshot of BaSyx (January 2020), to build that snapshot with the SSE CI server and to deploy that version to the SSE Maven repository[88]. If the partners decide about a new stable version of BaSyx (based on successful tests against the Support Layer), we will manually trigger the build process of BaSyx, which then will trigger the build process of directly dependent components such as `connectors.basyx`.

[88] For releases, BaSyx is included with group id `de.iip-ecosphere.platform.org.eclipse.basyx`.

The Transport Component (`transport`) is then the next component to be built after the Support Layer. If `transport` is changed, it triggers the building of the transport connectors (`transport.*`), the basic (optional) Spring integration (`transport.spring`) and the Spring binders (`transport.spring.*`) utilizing the transport connectors. The Connectors Component (`connectors`) relies on the type translation and serialization mechanisms of the Transport Component and, further, the individual platform/machine connectors (`connectors.*`) depend on the Connectors Component. The MQTT platform/machine connectors are, in turn, based on the corresponding transport connectors.

The components of the service layer (`services.*`) consist of the service manager interface including the abstract creation of the AAS (`services`), the specific implementation for Spring Cloud Streams (`services.spring`) as well as the generic service environment (`services.environment`) and the Spring-specific service environment (`services.environment.spring`).

The resource/deployment components (`ecsRuntime.*`) are partially realized, e.g., the ECS runtime and the container manager for Docker. The container manager for Kubernetes, the device management and the platform monitoring are in planning/realization and not part of this release.

The `platform` server(s) component provides the startup sequence for central services as well as the preliminary command line interface for platform functionality.

At the end of the hierarchy, there is currently the integration of the configuration model (`configuration.configuration`), which depends on the capabilities of EASy-Producer (stand-alone, Maven-based integration). EASy-Producer in terms of a standalone library can be executed by Java 8, however, also here test dependencies force partially an execution with Java 11. Fortunately, the configuration component of the IIP-Ecosphere platform only utilizes parts that require Java 8.

The Test Components (`test.*`) are a side track but required for testing. The protocol related test components contain integrations of embedded protocol brokers, such as Apache Qpid, HiveMq or Moquette, which shall be explicit testing dependencies rather than part of the production code. Moreover, `test.simpleStream.spring` is a testing artifact containing a simple stream processor chain for testing the Spring service manager in the Services Layer. Further, `test.configuration.configuration` provides implementations for the streaming interfaces created by the Test configuration instantiations in `configuration.configuration`. This introduces cyclic dependencies, but only for the very first build. For such a build, the configured artifacts in the respective tests can be set to an empty string, the test passes, creates and deploys the interfaces and the test artifacts can be build and deployed. In a second round, the artifact configuration is restored and the test runs again, now correctly packaging the streaming artifacts.

Figure 51 also indicates the actual state of using JDK 1.8 in the platform. There are some MQTT protocol components that require JDK 11, but only in testing. This also applies to `connectors.mqttv3`, which can be tested properly with HiveMq, but unfortunately not with Moquette. As soon as we become aware of more stable embeddable MQTT brokers for JDK 1.8, we will replace the dependencies to the test servers and switch the CI for those components to JDK 1.8.

8.4 Installing and using the IIP-Ecosphere platform

As discussed above, the IIP-Ecosphere platform must be configured and instantiated before it can be executed. Thus, the continuous integration does not provide complete platform dependencies (except for those created as part of testing `configuration.configuration`). Below are the required steps to run the actual release of the IIP-Ecosphere platform (this will change in the future).

1. Prepare the operating system. For the next steps in this section, we assume a Ubuntu 20.4.1 Linux installed on two machines (assuming 147.172.178.145 as “server” and 147.172.178.143

as “device”, we will adjust the IP addresses in the fourth step). Install `unzip`, Java JDK[89] and `maven` (version 3.6.3), `docker` (version 20.10.2):

```
sudo apt install unzip
sudo apt install openjdk-13-jdk-headless
sudo apt install maven
sudo apt install docker.io
```

On devices, the installation may differ as Java/Maven could be part of the container hosting the ECS runtime or the Service Manger/Services. By default, Docker requires root permissions to execute functions. If you want to use docker as “normal” user[90], execute

```
sudo usermod -aG docker $USER
```

Log out and log back so that your group membership is re-evaluated.

2. Obtain the IIP-Ecosphere platform install package, for snapshots from SSE Jenkins for Windows[91], Linux[92] or from github[93] for the actual release from github[94].
3. Unpack the install package. Obtain and install the IIP-Ecosphere platform dependencies first.

```
unzip Install.zip
cd Install/platformDependencies
mvn install
cd ..
```

This must be done once before other Maven installation steps.

4. Install a broker. The decision for the broker will influence the configuration changes in step 6. By default, the installation package ships with a basic setup of the AMQP broker Apache Qpid. To obtain that broker, execute

```
cd broker
mvn package
cd ..
```

[89] A JDK is needed for the platform instantiation rather than a JRE. As also discussed in Section 2, for devices a restriction to Java 8 may apply. You may install JDK 13 on the „server“ and JDK 8 on the „device“ or in respective containers. If your installation does not set the `JAVA_HOME` variable, the Maven scripts created during platform installation may issue a stacktrace warning that `Javadoc` cannot be executed, but the scripts shall pass. To prevent this, set `JAVA_HOME` so that it points to the JDK installation home directory.
[90] https://docs.docker.com/engine/install/linux-postinstall/
[91] https://jenkins-2.sse.uni-hildesheim.de/view/IIP-Ecosphere/job/IIP_Install/lastSuccessfulBuild/artifact/install.zip
[92] https://jenkins-2.sse.uni-hildesheim.de/view/IIP-Ecosphere/job/IIP_Install/lastSuccessfulBuild/artifact/install.tar.gz
[93] https://downgit.github.io/#/home?url=https://github.com/iip-ecosphere/platform/tree/main/platform/tools/Install
[94] https://downgit.github.io/#/home?url=https://github.com/iip-ecosphere/platform/tree/v0.2.0/platform/tools/Install

To run the broker, execute the respective script[95] in the `broker` directory. The broker is needed on both, the server and the device installations/containers. On the "server", the broker acts as global platform broker running by default on port 8883. On the "device", the broker acts as broker for local service communication. For installing on a device, copy the respective script and the folder `brokerJars` in `broker` to the device.

5. Run Maven[96] now on the install package itself

   ```
   mvn package
   ```

   You may perform `install`, but `package` is sufficient here. This step also obtains and unpacks the respective platform configuration model into `src/main/easy`.
6. Edit the example configuration file `TestInstall.ivml` in `src/main/easy` so that your local IP address is used. In this release, the devices are not listed in the configuration, i.e., search only for 147.172.178.145 and replace this IP by the IP of your "server" machine[97]. You may do more changes, but this requires the background knowledge from Section 6 on the platform configuration model. Currently, the selection of code artifacts is restricted to the Maven servers used for development, i.e., further artifacts cannot be obtained from further repositories, e.g., the future platform service store. This will be targeted by one of the next releases.
7. Instantiate the platform using

   ```
   mvn exec:java -Dexec.args="InstallTest src/main/easy gen"
   ```

   This executes the `PlatformInstantiator` mentioned in Section 3.11 through maven, passing it three parameters, namely the name of the model to instantiate (`InstallTest`), the relative folder where the model is located (`src/main/easy`) and the folder where to store the instantiated artifacts (the relative folder gen). Please note that this may fail if your modifications to the configuration file are syntactically or semantically incorrect. Alternatively, you can check out the full code from github and run the `PlatformInstantiator` from your IDE or force maven to copy all dependencies into a folder and run Java manually on the command line.
8. Copy the created artifacts in `gen` (`ecsJars/*`, `ecs.sh`, `svcJars/*`, `serviceMgr.sh`, `SimpleMeshTestingApp-0.1.0-SNAPSHOT.jar`) to the respective devices. For each artifact, the instantiation creates a folder with all dependencies and the respective startup script. In future versions of the platform, this step will be taken over by the device management, the automated container creation and the distribution of containers by the platform.
9. Install and start a protocol broker/server instance complying with the configuration settings. Just as a reminder, the IIP-Ecosphere platform does not ship with a particular broker, e.g., for MQTT or AMQP although the regression tests utilize specific brokers as discussed e.g., in Section 8.3.
10. Start the platform components, first the platform server(s) component, then the ECS runtime and, finally, the service manager through the startup scripts.

[95] So far, scripts are not shipped with file attributes, i.e., on Linux you may have to execute `chmod u+x` *`file`*. As an example, we provide a script to set the execution permissions of all files.
[96] The Maven commands are suitable for the first or a release version installation. If you use an actual snapshot, i.e., a version that was created and deployed as part of the development process and want to force Maven to use the most recent versions (by default, snapshots are updated only once a day), than add `-U` to the respective command, e.g., `mvn -U package`. During instatiation, the instantiation process takes care of snapshot updates.
[97] For this release we suggest not using HTTPS as schema or a non-empty endpoint path for the AAS server. Also for VAB, HTTPS is currently disabled. We plan to add certificate support in one of the next releases.

11. Run `cli.sh`. For deploying services, the created application artifact must be on the device running the service manager. Add the artifact to the service manager via the `cli` (through a local file URL on that device) and start the services. Please note that artifacts and containers are added through their URI, whereby local URIs may differ from system to system, e.g.,
    - Windows: `file:///C:/.../SimpleMeshTestingApp.jar`
    - Linux: `file:/home/ecouser/SimpleMeshTestingApp.jar`
    - Service container: `file:/apps/SimpleMeshTestingApp.jar`
12. To avoid timeouts, a shutdown shall happen in the opposite sequence, i.e., services, service manager, container manager, platform, brokers.

If you want to exercise the full cycle, create a Docker container with the service manager and the application artifact first and copy the container to the device running the ECS runtime[98]. It is important to emphasize that these steps shall be automated in future releases.

Copy the `container` folder from the installation package to your "device". Copy/move the artifacts from step 8 also into the `container` folder and execute there

```
docker build -t iip/simplemesh:0.1 -f SimpleMeshTestingApp/Dockerfile .
docker save iip/simplemesh:0.1 | gzip >
  SimpleMeshTestingApp/simplemesh-0.1.tar.gz
```

For convenience, both commands are available as `createAppContainer.sh` and `saveAppContainer.sh` in the install package. At a glance, the second step may appear superfluous, but it is required for the deployment and execution of the container through the ECS runtime. Please take care that the tag `iip/simplemesh` and the file name `simplemesh-0.1.tar.gz` are the same[99] as in the container descriptor `SimpleMeshTestingApp/image-info.yml`. Add and start the container (similar as described for the services above) through the platform command line interface before starting the services in the container.

With a running platform server and a running ECS runtime, you may also start the container manually. This would then require setting the AAS implementation server port correctly as stated in the container descriptor, i.e., `--network=host --expose` *`port`* `-e "IIP_PORT=`*`port`*`"`. If feasible, you may use the default port 9000 and use `--expose 9000` or more generically `-P` as parameters. An example script is included in the install package as `runAppContainer.sh`.

If you also want to containerize the ECS runtime (one of the possible edge device installations), ensure that the folder `container/EcsRuntime` is on the "device". For simplicity and to save resources, we map the `SimpleMeshTestingApp` folder as volume into the ECS container (mount point `/SimpleMeshTestingApp`).

```
docker build -t iip/ecsruntime:0.2 -f EcsRuntime/Dockerfile .
docker run -v /var/run/docker.sock:/var/run/docker.sock -P --network=host
  --mount type=bind,source="$(pwd)"/SimpleMeshTestingApp,target=/SimpleMeshTestingApp
  -it iip/ecsruntime:0.2
```

Akin to the app container, both steps are available as respective scripts in the install package. Before running the ECS container, it is important that the the app container has been created and stored. As administrative operations for installing Docker into the container are executed, Docker may issue certain warnings during the creation of the container. The default port for the ECS Runtime AAS implementation server in this Dockerfile is 9000.

[98] We provide scripts for creating, saving and running the container as part of the install package.
[99] The version number may differ but shall be the same as in the container descriptor.

9 How to apply, extend or contribute

In this section, we summarize procedures for some tasks that you may want to perform with the IIP-Ecosphere platform. In the last sub-section (Section 9.3), we provide answers to frequently asked questions.

9.1 Defining an own application-specific service

1. Adjust your platform configuration[97] and define a new service (as discussed in Section 6). Don't define an implementing artifact. Change the name of the application artifact to avoid overriding an existing artifact.
2. Execute the platform instantiation once so that the service interfaces are build. As part of that execution, your application artifact is created and deployed, in particular also a sub-artifact just containing the `interfaces` is created.
3. Create a Maven Eclipse project, use the IIP-Ecosphere platform dependencies as parent and add only required components as dependencies, in particular your configured application artifact (see step 1, use `interfaces` as type in the Maven dependency). Alternative and optional components such as AAS implementations or protocols may be added as dependencies in the test scope.
4. Realize the service, e.g., as Java class(es) implementing the new interface(s). We do not discuss Python-based services in this release as the service environment is still in development.
5. Modify the platform configuration by adding the artifact specification of your service implementation artifact to the configuration of your service(s).
6. Run the platform instantiation again so that the complete artifact is built.
7. Copy the artifacts to your installation devices, start the platform and try out your service as discussed in Section 8.4.
8. Let IIP-Ecosphere know about your work. In case of a potential open source component, please consider contributing it to IIP-Ecosphere.

9.2 Extending the platform by adding a component or a platform service

1. Make yourself familiar with the design of the respective component. Identify the interfaces to implement, e.g., the Service interface in `services.environment`.
2. Create a Maven Eclipse project, use the IIP-Ecosphere platform dependencies as parent and add only required components as dependencies. Alternative and optional components such as AAS implementations or protocols may be added as dependencies in the test scope, not in the (default) production scope.
3. Implement your component and test it.
4. Consider extending the platform configuration meta-model, i.e., search for the part describing the components. In some cases, e.g., AAS protocols, this may just be an additional entry in an enumeration. For other components, this may require a new typed IVML compound with default values (akin to the already given compounds). For services, no changes to the meta-model are required.
5. Adjust your platform configuration[97] so that your new entry is taken up. In case of a new enum value, use that value. In case of a new compound, replace the existing compound value by a value of your type (providing also the respective settings in the compound value). For a new service, add the service to the application part of your platform configuration and link it into the service mesh (as discussed in Section 6).
6. Run the platform instantiation as discussed in Section 8.4, copy the artifacts to your installation devices, start the platform and try out your extension.
7. Let IIP-Ecosphere know about your work. In case of a potential open source component, please consider contributing it to IIP-Ecosphere.

9.3 Frequently Asked Questions (FAQ)

In this section, we summarize some questions and issues that repeatedly occurred.

9.3.1 Error parsing HTTP header

Symptom: A part of the platform (platform server, ECS runtime, service manager or platform command line interface) issues an exception with the following message:

```
org.apache.coyote.http11.AbstractHttp11Processor.process Error parsing HTTP
request header Note: further occurrences of HTTP header parsing errors will
be logged at DEBUG level.
```

Reason: One reason may be that a client such as the command line interface tries to access a platform server (AAS server, registry) with an encrypted protocol (HTTPS) while the server is running a non-encrypted protocol (HTTP).

Solution: Ensure that the certificates for client and server side match. For this release, do not run the platform with an encrypting protocol[97].

9.3.2 Maven artifact missing

Symptom: While working with the platform against a release version in Maven, it appears that one of the (non-java) artifacts is missing.

Reason: Although we carefully check the artifacts before a release, it may be the case that the automatic deployment (script) missed some.

Solution: Please let us know about the problem via the IIP-Ecosphere website or via github.

9.3.3 Platform code cannot be setup in Eclipse, e.g., parent POM missing

Symptom: Your IDE reports missing Maven artifacts and shows compilation errors, in particular the parent POM of the platform is missing. Similarly, the code style checking may fail due to missing style definition file.

Reason: The parent POM of the platform defines the versions of non-singleton/wrapped libraries (cf. Section 4). Without that particular POM, compilation cannot run successfully as the artifact version numbers/ranges are missing. If you are working with a release version, it may also be the case that one of the released artifacts is missing (cf. Section 9.3.2).

Solution: Please refer to the code setup guide in Github[100].

9.3.4 Unknown platform coding conventions

Symptom: After a first contact with the platform code it seems that you are missing detailed information about applied conventions on how to write code and you cannot find all conventions in this document.

Reason: Although we tried to capture the most important conventions in this document, this document is not intended to be a programmer's guide, i.e., we do not necessarily repeat all coding conventions here.

Solution: Please refer to the platform coding guidelines in Github[81].

[100] https://github.com/iip-ecosphere/platform/blob/main/platform/documentation/Guideline.pdf?raw=true

10 Summary & Conclusions

Realizing an open (experimental) IIoT/I4.0 platform is a significant amount of work. IIP-Ecosphere is performing that work and this whitepaper provides technical insights into the ideas, concepts, rationales, designs and implementation state of the current release of the IIP-Ecosphere platform. The rationale behind this document is to enable interested parties to discuss with IIP-Ecosphere on a technical level, to try out the platform or to provide extensions. As the platform is evolving, this document is just a snapshot in time. Moreover, particularly this version of the document is the first of its kind – future versions may learn from feedback in order to improve the platform and also this document.

We discussed the technical basis for architecture modeling, the overview of the layered architecture, the individual layers and the components they contain. For each component, we provided a requirements analysis [based on [8, 30]) and a discussion of the realized requirements. We discussed architectural constraints, the actual use of Asset Administration Shells (AAS), the approach to platform configuration and instantiation, future contributions to the (external) security of the platform, selected implementation details as well as how-to's on applying and extending the platform.

In order to conclude about the actual state of the realization, we provide below some insights into selected realization Key Performance Indicators (KPI), namely requirements fulfillment, connectors, developed components, testing, use of open source components, and use of Asset Administration Shells.

Table 18 summarizes the discussed and realized requirements. The platform handbook of the current version discussed more than half of the top-level and sub-requirements for the platform. As several components are not yet realized or in realization but not part of this release, i.e., for which we do not discuss the requirements (status), we can also conclude that about a third of the requirements are already either completely or partially realized (and tested).

Table 18: KPI-based summary of discussed/realized requirements

KPI: Requirements (from [8], 141 top-level and 181 sub-requirements)	
Discussed top-level requirements	78 (55% of all top-level requirements)
Discussed sub-requirements	93 (51% of all sub-requirements)
Completely realized top-level requirements	37 (26% of all top-level requirements)
Completely realized sub-requirements	47 (26% of all sub-requirements)
Partially realized top-level requirements	5 (4% of all top-level requirements)
Partially realized sub-requirements	12 (7% of all sub-requirements)

Table 19 summarizes the number of "connectors" realized so far. In particular, various basic protocol connectors for transport and streaming have been realized and tested. For some components, even more connectors do exist, e.g., for Spring Cloud Stream.

Table 19: KPI-based summary of realized connectors

KPI: Connectors (*requiring application-specific extensions)	
Support Layer AAS connector	**1** factory connector for BaSyx
Spring transport connectors	**8** binders: RabbitMQ, Kafka, Kafka Streams, Amazon Kinesis, Google PubSub, Solace PubSub+, Azure Event Hubs, Apache RocketMQ
IIP-Ecosphere transport connectors*	**3** connectors: MQTT v3, MQTT v5, AMQP **5** connector binders: MQTT v3 (Paho, Hive), MQTT v5 (Paho, Hive), AMQP
Machine/plattform connectors*	**4** connectors: OPC UA v1, AAS, MQTT v3, MQTT v5
Sum	**13** connectors realized, 8 further available
Security	transport connector extensions for IDS
Data integration	-
Cloud connectors	semantic-based optional cloud connectors (if within resources, AWS and Gaia-X)
Application	northbound external platform connectors for data exploration and linking of IIP-Ecosphere platform instances
Compliance	AAS connectors for Sennheiser and SAP
Further planned	**7**

Table 20 summarizes the number of developed components categorized by the layers or logical components.

Table 20: KPI-based summary of developed components

KPI: Components developed	
Support Layer	3 (including 1 optional)
Transport Component	10 (including 8 optional/alternative)
Connectors Component	5 (including 4 optional/alternative)
Services Layer	4 (including 2 optional/alternative)
Resources Layer	3 (including 2 alternative/alternative)
Configuration Layer	1
Platform server(s) Component	1
Sum	**27 (including 17 optional/alternative)**

Table 21 summarizes the number of test cases realized by the platform. For judging the overall number, it is important to recall that the granularity of tests differs significantly, ranging from classical unit tests over integration tests up to validation and instantiation of a configuration in a single test. Also the number of tests differes, e.g., in the Services Layer, many fine-grained monitoring tests from [2] are defined that increase the number significantly. Moreover, the number of test cases is only one side of the testing medal. It is also import to consider coverage metrics. The line coverage is typically between 69% and 89% except for the following: The Spring environment is currently not directly tested rather than indirectly via the Spring service testing artifact, the test components that either consist of testing code only or define an artifact for component testing, e.g., the service testing artifact, or that are currently not part of the release (Kubernetes resource manager, device management, platform monitoring).

Table 21: KPI-based summary of tests

KPI: Tests (of various granularity, from unit to integration)	
Support Layer	58
Transport Component	19
Connectors Component	11
Services Layer	119
Resources Component	7
Configuration Component	4 + 2 (generated)
Platform server(s) Component	1
Sum	**221** test cases (including 2 generated cases), **69%-89%** line coverage

Table 22 summarizes the number of open source components used in and integrated into the platform.

Table 22: KPI-based summary of used open source components

KPI: Used Open Source Components (only distinct/top-level ones are listed)	
Support / AAS factory connector	**1** BaSyx
Transport component	**4** Eclipse Paho, HiveMQ client, Rabbit MQ client, Spring Cloud Stream **3** for testing: Apache Qpid Broker J, Apache HiveMq, Googlecode JSON simple, Google protobuf)
Connectors component	**2** Apache Milo (and as above Apache Paho) Testing relies on the components mentioned above as well as the server implementations provided by the used components.
Services component	**2** Micrometer (with Spring Cloud Stream), Spring Cloud Stream Local Deployer
Resources component	**2** Docker, Java docker client
Configuration	**1** EASy-Producer
SUM	**11 in production code, 3 only for testing**
Planned: Data Lakes	At least one feasible database
Planned: Security	IDS, KI-Protect
Planned: Installation	Broker like Eclipse Mosquitto or RabbitMQ
Further planned	at least **7**

We plan to provide a coherent Asset Administration Shell for the platform on each installed device, e.g., through the ECS runtime installations. Thus, the number of individual AAS, which differ due to the heterogeneity of the devices, depends on the actual on-site installation and probably leads to ***s+1*** **linked AAS** with *s* being the number of devices with ECS runtime installations and the +1 for the central platform installation in a factory (assuming a remotely deployed AAS). From a type perspective, this leads to two AAS types, one for the ECS installations and one for the central IT installation. Instead of accounting for that number, we count the number of sub-models (more precisely **sub-model types**) contributing to the IIP-Ecosphere platform AAS.

Table 23: KPI-based summary of realized asset administration shells

KPI: Asset Administration Shells (here sub-model types)	
Support Layer	**1** dynamic sub-model (types)
Transport Component	**1** static sub-model (transport connectors)
Connectors Component	**2** dynamic sub-models (installed connectors, active connectors), while the active connectors change dynamically at runtime
Services Component	**3 active** sub-models (services, artifacts, relations), change dynamically at runtime
Resources Component	**2 active sub-models (resources, containers)**
Configuration Component	**0** not realized so far
Platform (via Support Layer)	**1** static sub-model (platform nameplate)
SUM	**10 in production code, 3 additional for testing**
Further planned	**12** at least one per layer

In summary, the second basis release accompanied by this platform handbook already realizes roughly a third of the intended functionality and, thus, provides a good basis for platform research and case studies. However, also basic functionality that would be desirable for certain work is still missing. Thus, for the next release, we plan in particular for the following missing functionality:

- Python service environment
- Automatic creation of containers and their accessibility for devices
- Optional integration of Kubernetes based on flexible protocols
- More detailed configuration model with even more code generation
- Potentially, an initial version of the device management and the platform monitoring
- Integration of initial security/privacy mechanisms
- Integration of first services.
- An initial AAS guideline

11 References

[1] A. S. Ahmadian, Model-based privacy by design, PhD thesis, University of Koblenz and Landau, Germany, 2020.

[2] M. G. Casado, Service and device monitoring on devices in IIP-Ecosphere, IT-Studienprojekt, Universität Hildesheim, 2021

[3] J.-H. Choi, J. Park, H. D. Park, O. Min, DART: Fast and Efficient Distributed Stream Processing Framework for Internet of Things, ETRI Journal, 39 (2), pp. 202-211, 2017

[4] R. Dautov, S. Distefano, D. Bruneo, F. Longo, G. Merlino, A. Puliafito, Pushing Intelligence to the Edge with a Stream Processing Architecture, International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), pp. 792-799, 2017

[5] H. Eichelberger, A Matter of the Mix: Integration of Compile and Runtime Variability, *Workshop on Dynamic Software Product Lines, FAS'16*, 2016.

[6] H. Eichelberger, C. Qin, K. Schmid, Experiences with the Model-based Generation of Big Data Applications, Lecture Notes in Informatics (LNI) - Datenbanksysteme für Business, Technologie und Web (BTW '17), S. 49-56, 2017

[7] H. Eichelberger, S. El-Sharkawy, C. Kröher, K. Schmid, IVML Language specification, http://projects.sse.uni-hildesheim.de/easy/docs-git/docRelease/ivml_spec.pdf

[8] H. Eichelberger, C. Sauer, A. S. Ahmadian, M. Schicktanz, A. Dewes, G. Palmer, C. Niederée, IIP-Ecosphere Platform – Requirements (Functional and Quality View), Version 1.0, March 2021, IIP-2021/02-en, DOI: 10.5281/zenodo.4485774

[9] H. Eichelberger, K. Schmid, EASy Variability Instantiation Language: Language Specification, http://projects.sse.uni-hildesheim.de/easy/docs-git/docRelease/vil_spec.pdf

[10] X. Fu, T. Ghaffar, J. C. Davis, D. Lee, EDGEWISE: A Better Stream Processing Engine for the Edge, USENIX Annual Technical Conference, pp. 929-945, 2019

[11] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995

[12] Ø. Haugen, Common Variability Language (CVL) – OMG Revised Submission, OMG document ad/2012-08-05, 2012

[13] C. Hochreiner, M. Vögler, P. Waibel, S. Dustdar, VISP: An Ecosystem for Elastic Data Stream Processing for the Internet of Things, pp. 19-29, EDOC'16, 2016

[14] C. Hochreiner, M. Vögler, S. Schulte, S. Dustdar, Elastic Stream Processing for the Internet of Things, CLOUD, 2016

[15] J.-H. Hoepman, Privacy design strategies - (Extended Abstract). In ICT Systems Security and Privacy Protection - IFIP TC 11 International Conference (SEC'14), pages 446–459, 2014.

[17] International Data Spaces, IDS reference architecture model version 3.0, https://internationaldataspaces.org/ids-ram-3-0/

[18] The Industrial Internet Reference Architecture Technical Report, https://www.iiconsortium.org/pdf/IIRA-v1.9.pdf

[19] J. Jürjens, Secure Systems Development with UML, Springer, 2005

[20] H. Koziolek, S. Grüner, J. Rückert, A Comparison of MQTT Brokers for Distributed IoT Edge Computing, ECSA, 2020

[21] LNI 4.0 Testbed Edge Configuration – Usage View, https://www.plattform-i40.de/PI40/Redaktion/EN/Downloads/Publikation/LNI4.0-Testbed-Edge-Configuration_UsageViewEN.pdf

[22] N. Martz, J. Warren, Big Data - Principles and best practices of scalable realtime data systems, Manning, 2015

[23] D. O'Keeffe, T. Salonidis, P. Pietzuch, Frontier: Resilient Edge Processing for the Internet of Things, VLDB Endowment, 11 (10), pp. 1178-1191, 2018

[24] OMG, Unified Modeling Language, Version 2.5.1, https://www.omg.org/spec/UML/About-UML/

[25] Plattform Industrie 4.0, Die Verwaltungsschale im Detail, 2019, https://www.plattform-i40.de/PI40/Redaktion/DE/Downloads/Publikation/verwaltungsschale-im-detail-pr%C3%A4sentation.html

[26] Reference Architecture Model Industrie 4.0, https://www.plattform-i40.de/PI40/Redaktion/EN/Downloads/Publikation/rami40-an-introduction.html

[27] C. Sauer, H. Eichelberger, A. Ahmadian, A. Dewes, J. Jürjens, Aktuelle Industrie 4.0 Plattformen – Eine Übersicht, IIP-Ecosphere Whitepaper IIP-2020/001, 2020, DOI: 10.5281/zenodo.4485756

[28] B. Satzger, W. Hummer, P. Leitner, S. Dustdar, Esc: Towards an Elastic Stream Computing Platform for the Cloud, IEEE International Conference on Cloud Computing, pp. 348-355, 2011

[29] M. Staciwa, Experimentelles Container-Deployment auf Industrie 4.0 Geräte, Projektarbeit, Uni Hildesheim, 2020

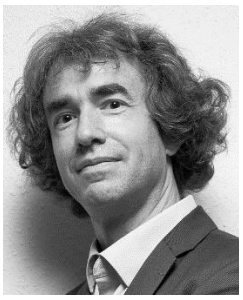
[30] H. Stichweh, C. Sauer, H. Eichelberger, IIP-Ecosphere Platform Requirements (Usage View), Version 1.0, Januar 2021, IIP-2021/001, DOI: 10.5281/zenodo.4485801

[31] K. Schmid, H. Eichelberger, EASy-Producer: From Product Lines to Variability-rich Software Ecosystems, SPLC' 15, 20215

[32] F. van der Linden, K. Schmid, E. Rommes, Software Product Lines in Action - The Best Industrial Practice in Product Line Engineering, Springer, 2007

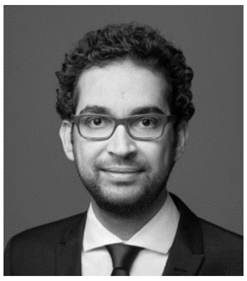
[33] T. Ziadi, L. Hélouët, J.-M. Jézéquel, Towards a UML profile for Software Product Lines, Intl. Workshop on Software Product-Family Engineering, 2003

Über die Autoren



Dr. Holger Eichelberger is deputy head of the Software Systems Engineering group at the Institute of Computer Science at the University of Hildesheim. He conducts research in the areas of software product lines, model-based engineering, performance monitoring, and performance analysis. In particular, he is interested in the integration of these areas to create adaptive software systems. In IIP-Ecosphere he leads the think tank "Platforms" as well as the AI Accelerator. He studied computer science at the University of Würzburg, where he received his PhD on the automatic layout of UML diagrams.

Fotograf: Daniel Kunzfeld



Dr. Amir Shayan Ahmadian is a postdoctoral researcher at the Faculty of Computer Science at the University of Koblenz-Landau. His research interests focus on the challenges of designing and implementing secure and privacy-friendly software systems as well as on the current developments in Industry 4.0. He studied computer science at the University of Paderborn and received his PhD in computer science from the University of Koblenz-Landau. During his doctorate, he developed a methodology to operationalize the principle of "data protection through technology design".

Dr. Andreas Dewes holds a PhD in experimental quantum computing from the Sorbonne University of Paris and the French Nuclear Energy Agency (CEA). He has founded several software companies and is the CEO of KIProtect GmbH, which develops advanced technical software solutions for data protection and data security. Within IIP-Ecosphere, KIProtect GmbH is developing a solution for the secure and privacy-compliant use of industrial & IoT data together with the consortium project partners and associated companies.

Marco Ehl is a research associate in the Software Engineering working group under the direction of Prof. Dr. Jan Jürjens at the Institute for Software Technology at the University of Koblenz-Landau. He researches model-driven methods for software development. His focus is on the analysis and explainability of automated production systems. He obtained his Master of Science degree in computer science at the University of Koblenz-Landau on the topic of model-based monitoring of integrated state machines.

Monika Staciwa studies computer science at the University of Hildesheim. In IIP-Ecosphere, Monika works in particular on container management, (virtualized) asset administration shells, the python service environment and (automatic) creation of containers.

Miguel Gómez Casado studies computer science at the university of Valladolid. During his ERASMUS+ visit at the University of Hildesheim, Miguel worked on service monitoring, representing monitoring information in and querying monitoring information from asset administration shells.