oktoflow

OKTOFLOW PLATFORM HANDBOOK

Version 0.8-SNAPSHOT (1/8/2026)

Holger Eichelberger, Ahmad Alamoush, Monika Staciwa

Disclaimer

The contents of this document has been prepared with great carefulness. Although the information has been prepared with the greatest possible care, there is no claim to factual correctness, completeness and/or timeliness of data; in particular, this publication cannot take into account the specific circumstances of individual cases.

Any use is therefore the reader's own responsibility. Any liability is excluded. This document contains material that is subject to the copyright of individual or multiple IIP-Ecosphere or ReGaP consortium parties. All rights, including reproduction of parts, are held by the authors.

This document reflects only the views of the authors at the time of publication. The Federal Ministry for Economic Affairs and Energy (BMBF), the Federal Ministry for Research, Technology and Space (BMFTR, previously BMBF) or the responsible project agency are not liable for the use of the information contained herein.

Publication: XXX, 2025

DOI: 10.5281/zenodo.8429685

As oktoflow is now used in ReGaP as technical innovation platform and this handbook has been revised for the use in ReGaP, we decided to remove authors that were not active for a longer time or are not active anymore. Thus, the author list of this handbook is dynamic and may be extended again in the future.

Gefördert durch:
Bundesministerium für Wirtschaft und Energie
IIP-Ecosphere
Next Level Ecosphere for Intelligent Industrial Production
GEFÖRDERT VOM
Bundesministerium für Bildung und Forschung
ReGaP
reduce and gain

Executive Summary

The oktoflow platform (initially developed in the BMWK project IIP-Ecosphere) is a novel platform for Industry 4.0, e.g., based on 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 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 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.

This version of the handbook focuses on the platform release as of XXX 2025 (version 0.8) and supersedes older versions of this handbook/the platform.

Acknowledgements: We are grateful to Dr. Christian Sauer and Alexander Weber from the Software Systems Engineering Group of the University of Hildesheim for cross-reading this document and providing valuable feedback and ideas for improvement. Further, we would like to thank Christian Nikolajew for his work on the MODBUS/TCP and REST connectors, Jobst Hillebrandt for his work on the ADS connector as well as Thomas Lepper and Aleks Arzer from PZH/IFW of the Leibniz University Hannover for their testing support and input.

oktoflow was partially supported by the BMWK project IIP-Ecosphere (grant 01MK20006C) and DatiPilot ReGaP-PgE (grant 03DPC1511B) and by the BMFTR DatiPilot Innovationcommunity ReGaP, sub-project ReGaP-PgE, (grant 03DPC1511B).

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 [ESA+25].

We understand the term platform as a coherently integrated set of software frameworks or libraries to allow for and enable the execution of user-defined apps, here, in the domain of Industry 4.0 [EN23]. A platform may support distributed execution of the apps, may be installed in a cloud, locally on-premise or in a hybrid form exploiting the edge-cloud continuum.

The oktoflow platform was created in the context of the BMWi-funded[2] project IIP-Ecosphere, which purused the vision of enabling 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 aimed 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 was 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. This platform, oktoflow, became in BMFTR DATIpilot ReGaP[3] in 2025 the technological innovation core for energy applications in the industrial context.

Towards the design of such a platform, we analyzed in [SEA+20, ESA+25] more than 40 research IIoT platforms and 21 industrial IIoT platforms with specific relevance to IIP-Ecosphere. In [SSE21, ESA+21], we discussed the requirements for the oktoflow platform from two different perspectives, namely the usage view and the functional/quality requirements view [ESS22]. The resulting platform design 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 [SEA+20] rather than enhancing them. To a certain extend, this also covers the need of openly integrating various solutions into the platform using different mechanisms. 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 [ESA+25]. Besides the desirable abilities mentioned above, following the initial decisions made in [SSE21, ESA+21], the platform shall be service-based and virtualized through containers. One relevant Industry 4.0 standard to use and to integrate the parts and pieces of the platform is the **Asset Administration Shell** (AAS) that we aim to apply as self-description and interface to software components across all platform layers. The IIP-Ecosphere consortium discussions regarding a vision of the platform also emphasized the need to directly communicate with production machines, in particular, to utilize edge devices as compute resources and, if feasible, cloud technology. This re-shaped the character of the envisioned platform from a purely virtual to a mixed-virtual platform with

[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://regap.de

stronger aspects of an 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 handbook, we aim at discussing and documenting the architecture and the implementation of the 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 implementation realizes and fits the architecture. Thus, this document reflects the current state at hands, while we aim at updating this document as part of improving the 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. This version of the handbook focuses on the platform release as of December 2025 (version 0.8) and supersedes older versions of this handbook/the platform.

This platform release comprises the oktoflow plugin-architecture, several upgrades (Java 17/21, Python 3.13 with virtual environments, Angular 19), multiple-in-multiple-out connectors, generic transport connector on service level, multiple-in/out anonymization/pseudonymization, application templates (for ReGaP), the integration of AAS metamodel version 3 (BaSyx2) as well as a series of new connectors (MODBUS/TCP, REST, INFLUX, serial, file).

1.2 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 [SSE21] and the functional/quality requirements view [ESA+21]. 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 [SSE21, ESA+21] 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.

In Section 3 we introduce and discuss the architecture of the platform, ranging from 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 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 our work 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 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

[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.

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 particular for Sections 3 to 7 it is important to recall that the platform is currently under agile and incremental development, i.e., details and structures may change. Faster access to such information, we started turning modeling- and implementation level details into github documentation, which is easier and more agile to change than the handbook focusing on the more fundamental structures and decisions.

Ultimately, in Section 8 we summarize and conclude this document. In Section 9 we list references to other work that we rely on.

We recommend **different reading flows** for different audience groups:

- Readers with **architectural interests** find most relevant information in Section 3, which explains the layered architecture from bottom to top as well as the architectural decisions and constraints in Section 4. Potentially, also Section 2 is helpful regarding the utilized tooling and the basic technical decisions as helpful background.
- **Users**, who want to **install the platform** and do first steps are adviced to start with the github online documentation and may then turn for background to Sections 7.2 - 7.5. For first steps in configuring the platform or for creating applications on model-level, we recommend the online documentation on modeling concepts and properties as well as Section 6 as background. Please consider that at its heart, the platform is a distributed system and requires certain programs running on the nodes that shall perform application tasks. Depending on your installation, different user permissions also for installing programming language libraries may have to be considered.
- **Users**, who want to create **own applications**, are adviced to read the online documentation on modeling concepts and properties as well as Sections 6.8, 6.9, and 6.10. If you are also responsible for (your own) software installations on your computer(s), please take the reading flow for installing the platform into account.
- **Users**, who want to use the **graphical management user interface**, we recommend the github online documentation for platform installation and modeling concepts as well as Section 7.4 and the modeling basics from Section 6, in particular Section 6.4. Further, the overview of the user interface in Section 3.12 is recommended.
- **Developers for services and connectors** shall know the platform architecture basics, i.e., Section 3.3-3.5 as well as how to develop applications, in particular for testing, i.e., Section 6 for configuration and creating own applications. Further, the online documentation, the guidelines and the code documentation are relevant.

As already indicated above, we are about migrate detailed technical information to github, e.g., besides the modeling concepts, the FAQ and the manual installation steps are now part of the online documentation in github[5].

[5] https://github.com/iip-ecosphere/platform/blob/main/platform/documentation/README.md

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) [UML]. We will not introduce UML in this document rather than assuming that the reader is sufficiently familiar with UML. As tool support, we use Eclipse Papyrus[6]. While there is a broad range of modeling tools available, in particular commercial ones, we decided to use Papyrus, because 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 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, e.g., if technical screen resolutions change, which could be presented in more pleasing manner if only some more diagramming functionality would 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. However, if actual updates of Papyrus still fail with displaying the correct layout of our diagrams on recent resolutions, we plan to migrate the architecture model into a different tool so that we can provide updates of the diagrams in this handbook.

Along with the architecture and the design of individual components, also architectural constraints arise. We will discuss the architectural constraints of the 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 consider that in particular AI services will be realized in different programming languages.

- For the **Java** components constituting the platform core and the connectors, we rely on Eclipse with Maven[7], Git[8] and checkstyle[9] integrations[10]. Fundamental technical decisions are documented in the architectural constraints and, more detailed, in code. As we use Maven for the platform installation, a Java Development Kit (JDK) is 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, thus, all dependencies must be available through official or own/local Maven repositories. The platform provides various own Maven plugins, realizing specific functionality like variability installation or end-to-end testing. Templates for code formatting and validation of the formatting are available for checkstyle in the source code repository as part of the most fundamental platform dependencies project

[6] https://www.eclipse.org/papyrus/ version 4.8
[7] https://maven.apache.org/
[8] https://git-scm.com/
[9] https://checkstyle.sourceforge.io/
[10] For the required versions, please see https://github.com/iip-ecosphere/platform/platform/documentation/PREREQUISITES.MD

and shall be applied prior to any commit. A common logging abstraction was realized and must be used at least in the platform core. Components of the platform are represented as individual projects using Eclipse as main integrated development environment (IDE). With version 0.8 of the platform, we upgraded the code to Java 17 and test against Java 21 (except for some components like RapidMiner Real Time Scoring Agent (RTSA) still requiring an installed JDK 8 for execution). For the continuous integration, the build/deployment process is specified due to technical reasons in ANT, partially setting CI specific variables, ultimately calling Maven.

- While some AI methods may also be realized in Java, nowadays AI methods are typically implemented in **Python**[10]. For Python services (as for Java-based services), a service execution environment is provided, which is responsible for the communication with oktoflow (there the Java service counterpart), so that an AI developer does not have to deal with both languages, protocol details or a plethora of different protocols. The service environment and the integrating Python services can operate with virtual Python environments. Python services must explicitly declare their dependencies, e.g., used AI frameworks as well as (if need) target virtual environments in the application model of the platform configuration to enable automated creation of installation artifacts, in particular containers, and execution in the desired Python environment as services and their dependencies may require the installation of different virtual environments or even Python versions.
- In particular, we prioritize **dependency reduction** over alternative, potentially more modern programming approaches as well as isolated loading of classes and their dependencies to cope with incompatible libraries. Thus, we decided not to use frameworks like OSGi or Spring as foundation as they may lead to (future) dependency conflicts, even among different versions of these frameworks needed in the same platform instance (as we experienced for Spring Cloud Stream as well as the AAS implementation Eclipse BaSyx and BaSyx2). Akin, we prefer in some places boilerplate code over annotation-style programming, e.g., in platform parts where a later revision with yet unclear external decisions can be foreseen. Where adequate, we leverage own, abstracting annotations and interfaces (as basic for plugins) and prefer them over technology-dependent annotations or interfaces.
- Some components require **technical settings** for their startup, e.g., certain internet addresses 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. So far, no automated discovery mechanisms (for I4.0) settings was integrated (BaSyx2 discovery could be an option if AAS metamodel version 3 is enabled), which could ease the setup. 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[11], for machine-readable complex data in AAS on JSON[12]. Regarding terminology, we distinguish between **Setup** (the technical information, e.g., in Yaml, in practice often also called configuration) and the **Configuration** (the managing part, in terms of a configuration model, used for generating consistent setup information as well as related code and further artifacts). Related source code shall be named accordingly[13].

[11] https://en.wikipedia.org/wiki/YAML
[12] https://www.json.org/json-en.html
[13] Initially, we aimed for an alignment with Spring, also calling the technical setups a “configuration”. However, this led to some confusion, so we decided for version 0.3.0 to refactor the platform code according to the setup/configuration naming convention introduced above. However, some parameter/variable names and

- Components shall internally communicate via **interfaces** that encapsulate technical dependencies. Alternative and optional components shall be realized based on interfaces and register themselves into the platform. For Java, we use the Java Service Loader (JSL) mechanism[14], which associates concrete implementations to their respective (descriptor) interfaces through text resource files declaring the actual implementations. We use that mechanism to define, e.g., plugin (setup) descriptors, factory instances, to compose AAS but also to set up the component lifecycle, e.g., to handle the start and shutdown process[15].
- Since version 0.8, third party libraries must be encapsulated for **isolated loading** into **oktoflow plugins** and, as stated above, be used via an interface defined by the platform. The main reason is to actively mitigate dependency conflicts. As a prerequisite, the platform core components must be free of third-party dependencies (except for the Java library). Further, plugins allow for isolated testing of the integration of dependencies and enable the individual evolution of technical dependencies. For flexible integration with the platform, oktoflow plugins also employ JSL, one descriptor for defining a plugin's contribution, a second determining how the plugin is actually loaded.
- 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. **Test artifacts**, e.g., setup files created specifically for testing components or dependencies used only for testing, must be **strictly separated from production code**, e.g., reside only in test resource folders. In particular for Java components this is important as setup files that are accidentally placed in production resource folders may take precedence over generated setup information, i.e., prevent that the configuration decisions made by the user are enacted.

As stated in Section 1, for several reasons one objective of the platform is to use existing Open Source solutions wherever feasible. However, not all **Open Source licenses** are per se permissible in industrial contexts. Therefore, the we reviewed Open Source licenses and categorized them into four categories:

1) Usable without limitations, e.g., MIT, BSD-2-Clause, BSD-3-Clause, ISC, CDDL1.0, Eclipse-Dist-1.0.
2) Permissible, but potentially problematic, e.g., Apache 2.0, LGPL-2.1, Artistic-1.0-Perl, EPL-2.0, MS-PL, MPL-1.1.
3) Commercial licenses.
4) Problematic and potentially not allowed (as default or core dependencies), in particular due to copy-left implications, e.g., GPL-2.0, GPL-3.0, EPL-1.0, AGPL-3.0. In some cases, the use of binary artifacts of software under such licenses may still be permitted as long as the license information and the origin are stated, the underlying code is not modified or included and the integrating component is optional.

These categories shall be considered already during the design of the 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. Thus, platform components relying on commercial licenses shall be optional by

---

comments may still use `configuration`, `config` or `cfg` where `setup` would now be correct. We will try to clean up these (local) inconsistencies incrementally over time.

[14] https://docs.oracle.com/javase/8/docs/api/java/util/ServiceLoader.html

[15] However, due to the oktoflow plugin mechanism, the Java service loader shall be created directly only if a specific classloader shall be applied. Otherwise, the platform classloader shall be used via `ServiceLoaderUtils.load`.

default. Similarly, also software under not permissible licenses could be used in optional components, but to avoid later license conflicts, licenses of the fourth category shall be avoided wherever possible.

The source code of the platform is made publicly available in the **GitHub** space of IIP-Ecosphere[16]. Moreover, to foster transparency, the development of the platform happens in public. In the future, 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[17] of the SSE group. Java parts including additional artifacts (binary, python, configuration model) of stable releases are deployed to Maven central[18].

[16] https://github.com/iip-ecosphere/platform/
[17] https://projects.sse.uni-hildesheim.de/qm/maven/
[18] 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 oktoflow platform aims at realizing the requirements collected in [SSE21, ESA+21] as well as further requirements that are collected or detailed in further work or projects like ReGaP. 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 development approach, 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 in the remainder of this document. At the end of Section 3.1, we detail some further basic aspects, namely, relation to reference architectures in Section 3.1.1, the concept of data flow processing in Section 3.1.2, a brief introduction into Asset Administration Shells in Section 3.1.3, high-level component interactions in Section 3.1.4, and the virtual character of the platform in Section 3.1.5. Section 3.2 takes up the general requirements from [ESA+21] as context for the platform architecture. As basis for the architecture description, we discuss then the layers of the platform, first as overview and then one section per layer, starting at the bottommost (generic) layer.

3.1 Overview

The overall architecture of the platform follows a layered style (see Figure 1) based on components and services (R4 in [ESA+21]). 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.

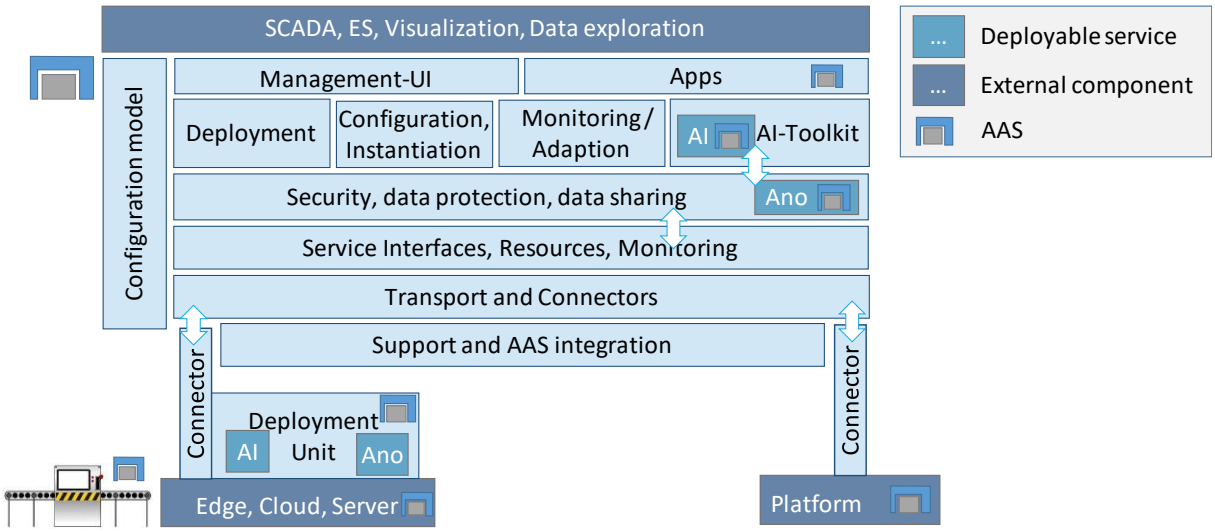

Figure 1: Main platform components as block diagram.

Figure 1 depicts an overview of the high-level building blocks of the platform. The overall goal is to enable service-based AI-enabled applications to be deployed to available resources utilizing standardized protocols. Therefore, the platform encompasses on the two lower layers (middle of Figure 1) the platform-internal data transport, connectors to external data, e.g., to machines, basic service interfaces, resource abstraction and monitoring facilities. These two layers form the basis for units that can be deployed to edges, servers or to a cloud. The remaining layers add services, e.g., on data protection, data integration, AI, runtime adaptation/centralized monitoring, overall configuration

and deployment of services/containers. If permitted, services from those layers can be deployed to available resources (bottom of Figure 1) by the platform.

- **Asset Administration Shells** (R7) are used in the platform in two forms: On the one side, AAS represent assets created by different vendors (e.g., a product, a machine, an edge device, an already installed platform, a service, a storage mechanism or an App composed of services). The respective AAS may be provided by a third party. Also, the platform itself forms such an asset that deserves an own AAS. On the other side, we utilize the mechanisms of AAS also for describing interfaces of interacting components within the platform. These components may be internal or external, i.e., also interfaces provided by external AAS may be used. In particular, AAS (submodels) for internal components may be created for the purpose of internal communication rather than external component realization and, thus, may not follow official standardized formats[19]. An integration of AAS (different implementation frameworks in terms of plugins) as well as support for realizing (internal) Asset Administration Shells forms the bottom-most layer of the platform.
- The platform contains an event-based **transport 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 the indented layering (R7 [ESA+21]) 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 [ESA+21]). As event-based communication and data streaming are essential to the platform, they occur on one of the fundamental layers (`Transport`) utilizing the external components `Broker` and `StreamingLibrary` through plugins.
- **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 [ESA+21], but also in the (variability-based) configuration model that crosses several building blocks/components in Figure 1. 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, and, if required, provide access to the individual configuration via respective interfaces. Also the configuration technology is encapsulated as a plugin. Similarly, only some few selected mechanisms to instantiate variability shall be utilized, in particular code generation, generation of setup files and artifact selection while packaging.

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

- **Support Layer:** The support layer (not shown in Figure 1) realizes basic abstractions and helpful functions for the upper layers of the platform. This includes logging, resource loading, plugin management as well as basic data format abstractions for YAML and JSON. Further, a set of utility functions for files, archives, system access or network are provided to reduce repetitions in higher leayers. This also involves non-trivial management functions or functions to create

[19] At the point of writing, several forms of AAS are in standardization, but most known to us do not aim at platform components. Wherever possible, we utilize existing standards, e.g., for device nameplates, or try to adopt the style of related standards to express proto-AAS, e.g., for software services.

common AAS structures and to foster internal conventions, e.g., how to represent certain information in AAS. Moreover, this layer contains an abstraction of AAS as well as AAS implementation plugins.

- **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 data transport protocols such as MQTT[20], AMQP[21] or OPC UA pub/sub[22] and their wire formats (e.g., JSON for MQTT), to provide implementing transport plugins and to integrate the abstraction with the streaming technology (`StreamingLibrary`). In contrast to recent platforms [SEA+20], 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. Further, 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 or AAS relying on an own information model. In contrast to the Transport Component, projections and transformations of the original input data of a connector may be ingested into user-defined apps and information/commands originating from the apps may be transported back, e.g., to reconfigure an underlying machine.
- **Services Layer:** Openness and extensibility through services of different kinds, in particular AI services, are at the heart of oktoflow. 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. On this layer, Connectors are wrapped into services so that they can be used seamlessly in user apps. Services may be realized in different programming languages and, thus, demand different integration capabilities, ranging from direct calls (Java services) to communicating operating system processes (Python services, GO, or even standalone Java programs). Services are wrapped by code generation into service units for a certain **service execution**. A specific service execution, e.g., Spring Cloud streams, is an implementation plugin of the services layer.
- **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 the platform, 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 and the platform AAS. For deployment, the `Deployment Unit` (called "ECS runtime" in [SSE21]) receives commands via its AAS from the platform and, in its containerized form, downloads a container including installed components and starts the container (ECS implementations plugins for Docker and LXC are part of oktoflow). In this containerized environment, apps are then started through the constituting service implementations[23] by the service execution. Also the execution of the services in the container

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

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` 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 (a default realization in terms of Prometheus[24] is a monitoring implementation plugin of oktoflow). 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.

- **Storage, Security and Data Protection Layer:** Security and data protection in the 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, this layer realizes in particular components (optionally) enhancing the security and data protection as platform-supplied app services, e.g., for `Anonymization` and `Pseudonymization`.
- **Reusable Intelligent Services Layer:** The components described so far (as well as not mentioned administrative services provided by the platform) can be used to develop applications similar to existing platforms [SEA+20]. This layer shall pave the way for open, extensible and reusable intelligent services. In particular, the `AI-Toolbox` contains re-usable AI services that can be parameterized and orchestrated to form a running application, e.g., Federated Learning with flower based on generated Python services, an optional integration of the RapidMiner RTSA as generic, re-usable AI service or a re-usable basic data processing library.
- **Configuration Layer:** The configuration layer contains components to manage the platform configuration. The `Configuration and Instantiation` component is responsible for composing reusable and application-specific services and representing the information in terms of the application parts 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, containers are created automatically and made available. In particular, this involves code generation of various artifact types, from app/service code templates over integrating service wrappers for the service execution to build specitivations. Further configuration operations target the re-configuration of services or the runtime-selection of alternative services.
- **Applications Layer:** Applications are described in the app part of the configuration model and may ship with application-specific components, e.g., AI services. Although not visible here, glue or transport code generated for services implicitly belongs to the apps. The execution of the apps shall be visualized by (as far as feasible) generic `Dashboard` components.
- **Management User Interface:** Ultimately, a Web user management interface (`UI`)[25] relying in particular on components of the `Configuration` layer and the AAS of the platform allows for configuring apps, supports the implementation of apps as well as their distributed execution. It is important to emphasize, that although the management interface is realized as a Web UI, the platform must not necessarily be installed/deployed in a Web/Cloud setting, i.e.,

[24] https://prometheus.io/

[25] As discussed in [ESA+21], 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.

on-premise installation and use of the Web UI via a browser is one important installation alternative for oktoflow.

The platform may or may not interface with Clouds or dataspaces as desired by the user, e.g., to not include/remove respective connectors and components completely from the individual platform instance/apps upon platform instantiation.

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 [SSE21]) to take control over containers and services, while monitoring, device management and platform AAS shall be running on central IT. For example, the service manager can be composed from a subset of the layers as indicated in Figure 2, in particular support, transport and connecors and services (using the respective oktoflow plugins indicated in light blue/italics). Similarly, the ECS runtime, in particular its variant including the service manager can be composed from lower layers and the respective components from devices and monitoring. For managing containers, at least the deployment unit (implemented as ECS runtime plugin) from the Resources and Monitoring Layer is needed. Service manager and ECS runtime can run in the same container/on the same device, as individual processes or combined. However, ECS runtime and service manager may also run as invidual containers, the one for the service manager then also containing all dependencies that apps do require, e.g., respective Python installations. The platform monitoring component can be instantiated as individual service, in Figure 2 intentionally without plugin, i.e., not relevant. On top, the “platform service” hosts the AAS with device management, excluding ECS runtime and service manager.

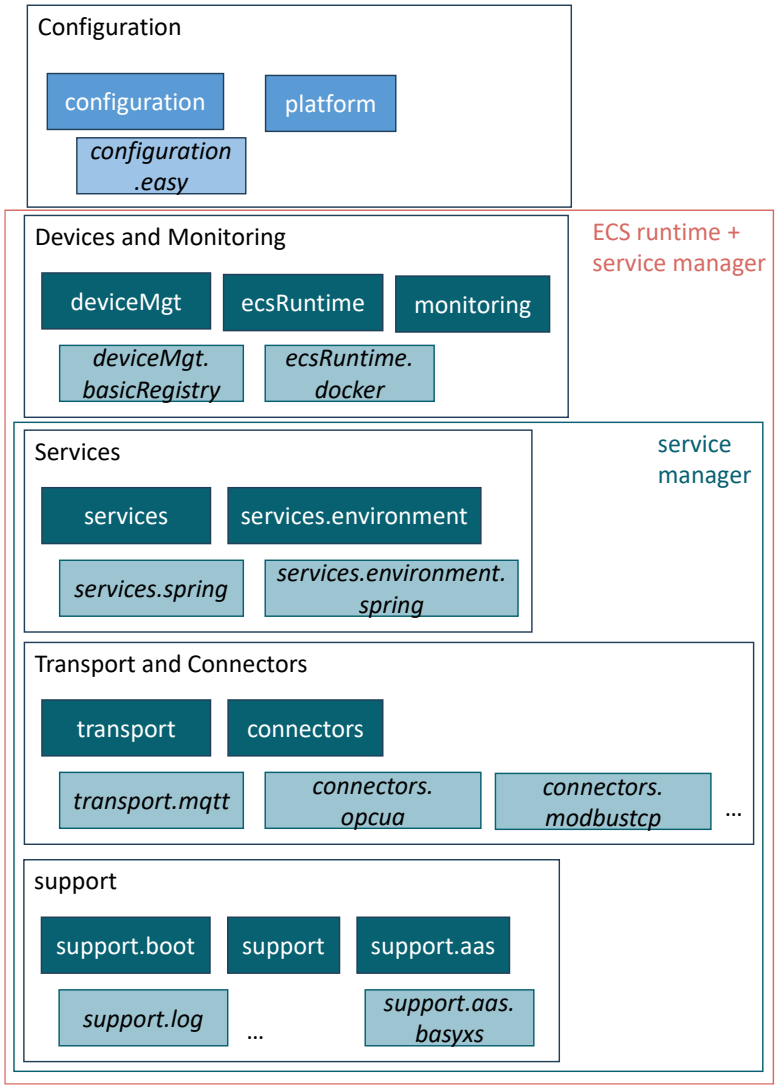

Figure 2: Layers, components and plugins required to build a service manager, the ECS runtime and the platform service.

3.1.1 Relation to Reference Architectures

The platform aims at interrelating and adhering to reference architectures such as RAMI 4.0 [RAMI]. 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 platform was initially planned to be a virtual platform, i.e., it shall be able to build on existing installations without implementing a complete IIoT platform stack. Thus, it is for us not relevant to meticulously adhere to all RAMI levels, in particular not to the lower levels targeting field devices (as already scoped out in [SSE21, ESA+21]). In addition, our architecture includes some (crosscutting) layers that do not directly fit into the picture of RAMI[26].

Table 1: Mapping RAMI 4.0 and the platform architecture

RAMI 4.0 Axis	RAMI 4.0 Level	Oktoflow Layer/Component
Layers	Asset	Not in scope [SSE21, ESA+21], 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 [SSE21, ESA+21], represented through edge AAS
	Control Device	ECS runtime [SSE21] with deployed services, in particular Resources and Monitoring Layer with contributions from upper layers
	Station	ECS runtime [SSE21], possibly with access to more powerful resources or UI capabilities for executing or controlling deployed services. Includes 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 platforms.
Life Cycle Value Stream	Type	Component and AAS types prescribing structures
	Instance	Deployed component and AAS instances

In terms of the Industrial Internet Reference Architecture [IIRA], this document can further be understood as a continuation of the usage view(point) [SSE21], the functional view [ESA+21] In terms of a platform architecture as well as its implementation.

3.1.2 Stream (Data) Processing

In an IIoT/Industry 4.0 setting, often the processing of data is viewed in terms of streams of data items (or tuples), e.g., produced in regular fashion by a machine, taken up by edge devices for pre-processing, protocol transformation or retro-fitting, handled further by other devices and (partially) stored in some data stores, e.g., time series data bases. In contrast to other forms of data processing, e.g., batch processing, data stream processing can fulfill (soft) realtime requirements, of course, depending on

[26] Crosscutting aspects are better covered by IIRA [IIRA].

the data ingestion frequency (overload, backpressure) and the (relative) speed of the individual data processors.

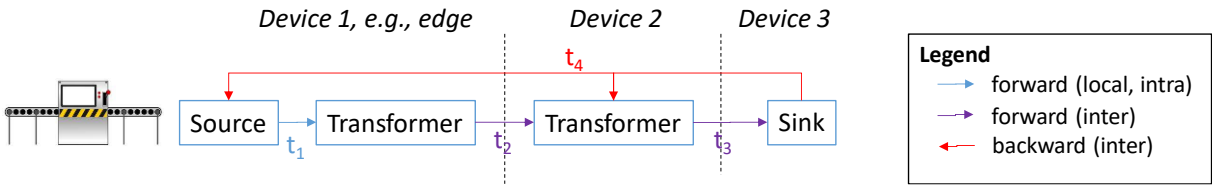

Figure 3: Viewing IIoT and Industry 4.0 as data streams.

Figure 3 illustrates the basic components of such a stream processing approach, considering "the machine" on the left side as constant (conceptually endless) data source. The data tuples/items produced by the machine is taken up by a data transformer (e.g., preprocessing, anonymization), passed to a second transformer (e.g., AI service) and finally to a sink (e.g., data store, dashboard). From a different point of view, the data flows in forward manner from source to sink. The edges in such a graph indicate the data flow and the nodes the data processors. There could be more processors, different kinds of processors or more complicated forward flows that we do not touch in this brief introduction. Please note, that there is no need for synchronous processing in the nodes, in particular in the transformers. With synchronous processing, we mean that a transformer operates like a mathematical function, i.e., for an input tuple it produces in the same step an output tuple. In contrast, asynchronous means that the processor receives data item(s) and at some future point in time it may emit any number of tuples (including none at all).

In Figure 3, there are also two horizontal lines, indicating borders of physical devices, e.g., the first two streaming components could be running on an edge device, the second transformer on a further device, and the sink on a third device, e.g., a central server. The distribution of components is not fixed, e.g., depending on resource usage, the second transformer could also be executed on the first device or the first transformer on the second device.

Several approaches to stream processing rely on untyped data, i.e., the transformer implementation decides based on the available data fields, what to process. Such an approach can easily fail at runtime, when processing nodes are combined that cannot work together, with negative outcomes ranging from loss of data to runtime errors or exceptions. In contrast, we rely on typed data flows, i.e., for each forwarding edge the type of data item(s) is known during design and built into the respective app. As the design of data processors and data flows will be captured in the configuration model of the app, checking for type and streaming compliance before realizing or instantiating the system becomes possible. In Figure 3, the forward flow indicates three data types, t_1, t_2 and t_3. Please note that depending on the requirements and the design of the data processing, the types may be the same or they could differ, e.g., indicating that a processor adds or removes data fields.

While in many applications, a forward flow is sufficient, in particular in IIoT/Industry 4.0 settings it could be desirable, that an upstream processor shall send back data to a downstream processor, e.g., a decision node after one or multiple artificial intelligence nodes shall inform the machine at the data source that some processing parameters must be changed. Akin to the forward flow, we allow types for backward flows. It is just a matter of modelling convenience that we define the forward flow in terms of nodes and connecting edges, while we consider the backward flow as typed notification data channel (t_4) of one or multiple senders and potentially multiple receivers.

3.1.3 Asset Administration Shells

The 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 model standards such as the Asset

Administration Shells (AAS). For short and without aiming for a complete description, an AAS is an information model, which targets a physical or virtual asset in terms of nested, detailing sub-models. Sub-models may consist among other kinds of elements of typed properties, operations and heterogeneous collections/lists 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. In particular, AAS for different assets of different vendors can be provided, related and integrated, e.g., to link the AAS of a device utilized by the platform into the platform AAS to provide, e.g., a digital nameplate for industrial equipment [BBB+20, ZVEI-N] or the documentation of the device at hands. Moreover, composite AAS can be created, representing, e.g., a complex machine consisting of AAS of the utilized components.

According to the requirements (here R7), the platform shall describe all (distributable) components, interfaces, functions and deployment targets in terms of AAS. Thus, each of the components of the platform that forms an individual asset (of a certain vendor) shall receive an own AAS (as indicated in Figure 1). Moreover, the platform itself shall provide an own AAS and each of the discussed layers shall provide one or more sub-models to link the layers against each other (whereby the sub-models may and shall link to the vendor AAS of the individual assets, e.g., edge devices). As far as feasible, the 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 [SSE21]. We will detail the platform AAS and its structure in Section 5.

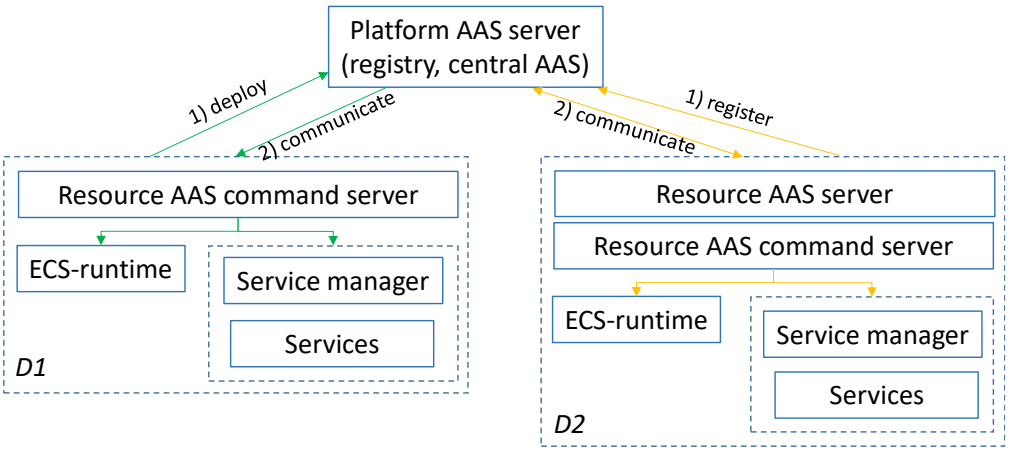

Figure 4: AAS deployment options (D1 remote deployment, D2 local deployment)

As typically several distributed compute resources are involved in a platform installation and each compute resource shall be used for running oktoflow apps, it shall be described/registered with an own AAS (model, sub-model or as part of joint model/sub-model). Therefore, it is helpful to introduce two basic AAS and component deployment patterns. Figure 4 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 operation calls, also to redirections of requests via the central server to the resource (which may anyway be the default behaviour of an implementation, e.g., BaSyX2 operation delegation). 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 manager running in an own container, can be linked to the AAS and directly contacted to serve AAS requests.

3.1.4 Component Interaction Overview

In the previous sections, we introduced the layers and the high-level components of the platform as well as the basic concepts of AAS. In this section, we provide a brief overview on the component interactions for a basic walk-through of platform operations. The individual sections on the components in Sections 3.3-3.12 will provide more detail on the interactions. In addition, Section 3.13 will address the cross-cutting topic of testing support for services and applications.

The aim of this walk-through is to bring up the ECS runtime, the service manager (in terms of a container), some services, to let the services run and to stop all parts in reverse order. Services are modeled as a service mesh forming individual applications (we will detail how to define such a mesh in Section 6). The required high-level interactions are illustrated in the sequence diagram in Figure 5. We will go through them now from top/left to bottom/right.

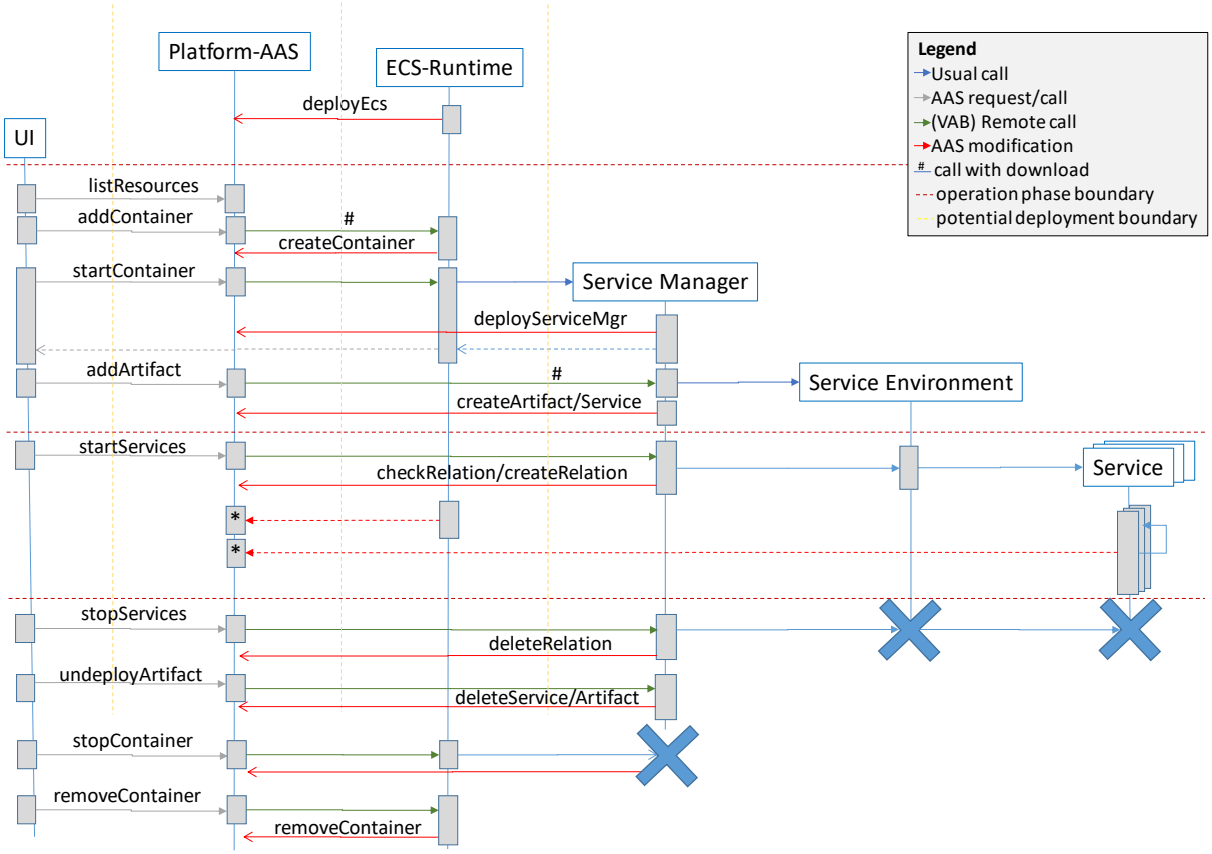

Figure 5: High-level component interaction for basic platform interactions.

1. At the beginning, the platform AAS-Server is running. An ECS runtime is started for a certain resource, e.g., an edge device. The ECS runtime instance then deploys its own sub-model characterizing the device with container operations and a collection of available containers (initially empty) into the platform AAS. A scheduled background process of the ECS runtime is started to inform the platform AAS about the actual resource state (resource monitoring, not shown in Figure 5). Depending on the device, the ECS runtime may provide information about

an existing device AAS or create a device AAS on its own (one particular point of openness as the device vendor may or may not provide an AAS). This information is linked from the platform AAS.

2. Via the user interface (UI), the user requests a list of available resources. The UI reads out the AAS submodel for resources including the ECS runtime instance started in step 1 and displays device information including the actual resource usage. In a similar manner, further information can be obtained, e.g., the available services, the defined applications, the packaged service artifacts or the available containers.
3. The UI requests adding a container via the ECS operations known to the platform AAS, leading to a remote method call to the ECS runtime (AAS implementation server). For this walk-through, we assume that the container contains the service manager and provides the technical dependencies for services to be executed on the respective device. Starting a container may lead to a download of the pre-built container from a central platform server (indicated by #) or from a file system of the device. Information about the container instance is made available to the platform AAS by creating a structure in the containers submodel of the platform AAS.
4. The user requests starting the container added in step 3, i.e., the UI calls the respective platform AAS operation, leading to a remote call to the ECS runtime (AAS implementation server), respective operations in the container management implementation, e.g., Docker, and, ultimately, when the container is running, to an automated start of the service manager. In turn, the service manager deploys information about itself, e.g., service operations, into the platform AAS, more precisely into the device entry created by the ECS runtime so that services on the underlying device can be managed.
5. So far, no app/service is known. The user requests to add an app via an operation of the platform AAS, leading to a remote method invocation to the Service Manager (AAS implementation server). In turn, as for the container, the service manager may download an implementation artifact containing the app, the services, and the related service execution environments for the actual device. The service manager adds entries for the artifact and all contained services to the respective sub-models of the platform AAS.
6. The user requests the start of the app, i.e., all services for the device addressed in the steps above through a deployment plan. The involved Service Managers start the service environment and creates the service instances in the sequence of dependencies, i.e., starting with the service having no data dependencies or for which all prerequisite services are already running. During this step, several network ports may be acquired for internal communication, relations to a global or a local protocol server/broker may be established and individual operating system processes for the services may be started. Further, connector settings may be adjusted to device-provided services as stated in the device AAS, e.g., specific IP address, actual port or even the startup-time selection of the right protocol implementation depending on the device-specified protocol version, e.g., MQTT v3 vs. MQTT v5. These detailed technical procedures are not shown in Figure 5. During service startup, the Service Manager checks the service relations in the platform AAS (services sub-model) for service availability and, as soon as the service is up, creates a relation entry linking two subsequent services in the service mesh of an IIoT application running on the platform.
7. The services are running now, receiving data via the machine/platform connectors, executing functionality specified for the actual application, e.g., AI-based inference. During the execution, background processes collect data for the device and the individual services and inform the platform AAS about actual runtime states, e.g., resource consumption. Here we also indicate in Figure 5 the resource monitoring of the resource mentioned in step 1.

8. The user requests to stop the running app via a respective operation of the UI/platform AAS, which causes a remote method invocation to the Service Manager(s). In turn, the Service Manager removes the service relations in the platform AAS and stops the service environment and the services.
9. The user (directly or through the deployment plan) indicates that the artifact will not be used any longer, i.e., a platform AAS operation is called and causes a remote method call to the Service Manager, which removes service and artifact entries from the platform AAS.
10. As also the service management container shall not be used anymore, a command from the UI to the respective AAS operation leads to a remote method call to the ECS runtime, which commands a stop of the container through the underlying container implementation.
11. Ultimately, the container shall also be removed from the management realm of the device, leading to a further remote method call to the ECS runtime, performing a removal of the container information from the platform AAS.

The horizontal dashed, red lines in Figure 5 indicate phases of the operations, i.e., startup (step 1), preparation of containers and services (steps 2-5), service operation (steps 6-7), shutdown (steps 8-11). The vertical yellow dashed lines indicate a potential distribution to different logical or physical devices. Extreme cases are that all components run on the same device, e.g., for testing, or that UI, platform AAS, ECS runtime and service manager/services are installed on separate devices.

It is important to emphasize that the “user” in this walk-through may be a human, a deployment plan selected in the UI or the platform itself acting on behalf of the user. A deployment plan lists the assignment of containers and services to resources so that the UI can execute the desired deployment automatically.

3.1.5 Virtual Character of the Platform

As stated in Section 1, the platform shall also have the character of 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 maps relevant underlying platform information and functionality into the 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:

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 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, as far as we know, such a standard currently does not exist.
2. The AAS connector of the platform can map the AAS of an underlying platform into oktoflow. Of course, this may add additional overhead and in some cases a mapping may not be possible at all.
3. One of the other connector types provides a protocol that allows mapping the underlying platform and its operations into the platform AAS. This approach may require manual programming, while the second approach might be realized easier through mapping and code generation.

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

3.2 Overall Requirements

In general, all platform layers and components discussed below must take the following general requirements from [ESA+21] into account:

Table 2: General platform requirements in [ESA+21]

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 the platform
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 realtime 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, [ESA+21] 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 [ESA+21]. 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 platform aims at the deployment of components to edge devices, both, the services as well as the platform operations 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), 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 platform. As also mentioned in [ESA+21], 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 [ESA+21], that the 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 [ESA+21]. Further, as also discussed in [ESA+21], 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

Requirement	Summary
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 data 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 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.

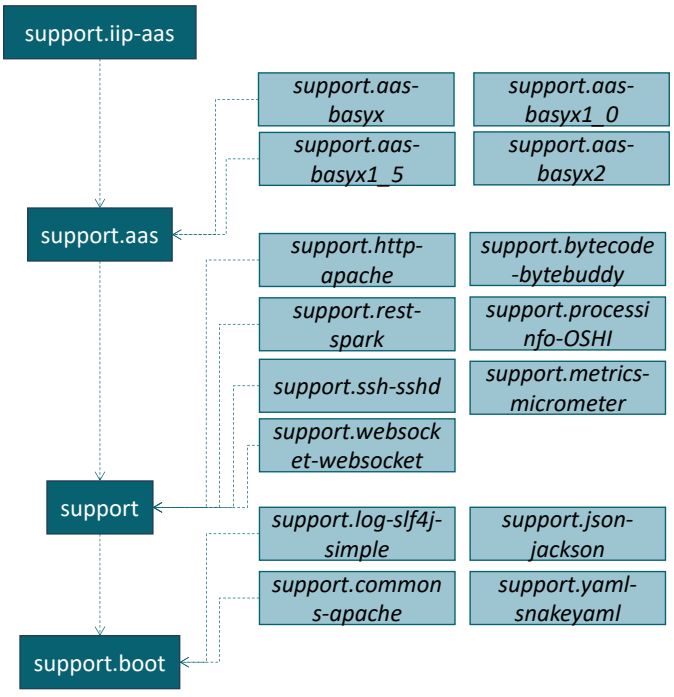

Figure 6: Structure of the Support Layer, core components left and plugins right (not all plugins are connected)

3.3 Support Layer

The Support Layer aims at providing useful common functions and abstractions to ease the realization of the 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. Below, we first discuss the structure of the whole layer, than it's four main components and finally, in Section 3.3.6 the recommended approach to implement platform AAS as well as in Section 3.3.7 the plugins realized for this layer.

3.3.1 Component Structure of the Support Layer

As illustrated in Figure 6, the most abstract component in the Support Layer is `support.boot`[27], which introduces the plugin mechanism and the resource loading as well as the fundamental plugin interfaces for logging, common operations, JSON and YAML.

The `support` component adds plugins that (partially) depend on the plugins introduced in `support.boot` as well as further common mechanisms. `support.aas` defines the AAS abstraction, i.e., the plugin interface for Asset Administration Shells. Further, `support.iip-aas` are specific AAS support functions including the AAS-based component lifecycle support as they are used in oktoflow (already introduced in IIP-Ecosphere, thus, "iip").

3.3.2 The support.boot Component

The `support.boot` component introduces the most basic mechanisms including some common functionality classes for collections, file/zip access, JSL, basic network functions, exception-enabled functional interfaces, etc. Moreover, this component defines the plugin interfaces for basic technical dependencies, such as common functionality, logging, YAML and JSON.

Figure 7 depicts a coarse-grained summary of the structure of `support.boot`. Below we focus on the plugin manager, the resource loader and the task tracking realized in `support.boot`.

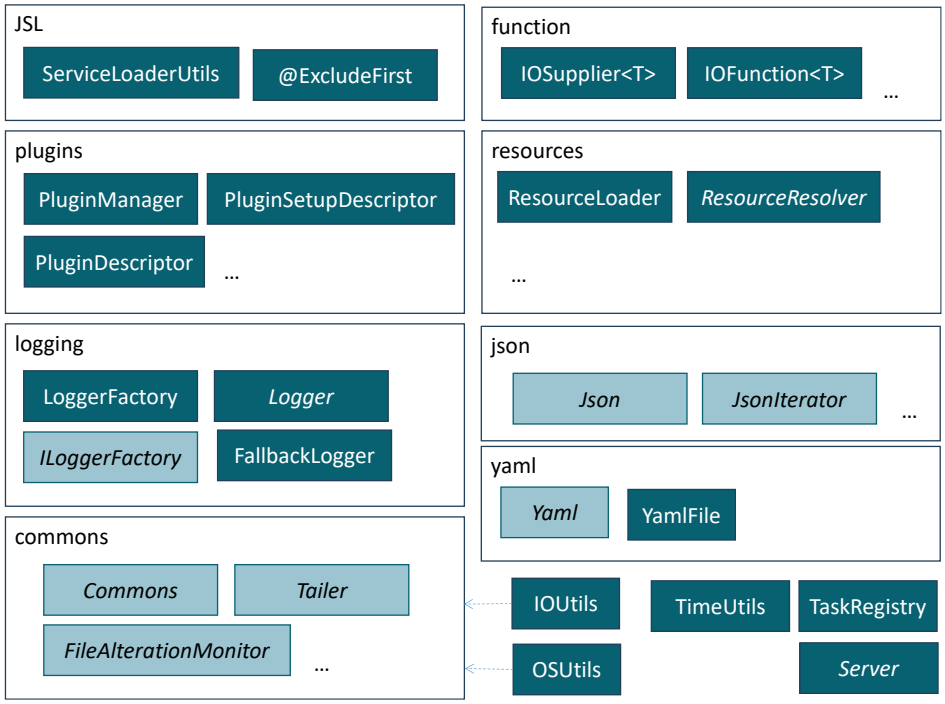

Figure 7: Simplified Structure of `support.boot`: core implementing classes, plugin interface (lighter background), delegating frontend utility classes like `IOUtils`. Plugin descriptors are not shown.

3.3.2.1 Plugin Manager

While most of the alternative oktoflow components can be combined without dependency or classpath conflicts, some (versions of the same) components would introduce conflicts, e.g., different versions of the AAS implementation BaSyx. To prevent that the development of oktoflow and oktoflow apps is forced by external configurations to certain dependency versions (the opposite direction is not

[27] `support.boot` was introduced in version 0.8 while support already existed; `support.boot` contains fundamental parts without using plugins while `support` uses the plugins defined in `support.boot`. As used in many build specifications, changing the name of `support` was considered too dangerous. As all support components share the same Java package namespace, future versions may move functionalites that are still in components based on past decisions.

realistic), we introduced a simple plugin management mechanism in version 0.8. Although proven implementations of such capabilities do exist, e.g., OSGi, and to prevent unpredictable conflicts with actually used and future dependencies, we decided to rely on a rather, simple classloader-based mechanism based on two JSL descriptors.

These are the `PluginSetupDescriptor`, which creates the plugin classloader and the `PluginDescriptor`, which creates specific instances of the plugin. The classpath of a plugin may rely on common components of underlying architectural layers, but not of higher layers, alternative components or other plugins (see also architectural rule C1). The `PluginManager` loads these descriptors and makes instances available through unique plugin identifier names declared by the plugins. Plugins may have a single identifier as well as multiple alternative identifiers, which eases migration from classname based instance creation of the previous platform version to plugins in this version. Different forms of `PluginSetupDescriptor` do exist, most are based on a persisted list of dependencies created during the build process of the plugin. Some example descriptors are: Loading of unpacked plugins from the file system (`FolderClasspathPluginSetupDescriptor`), from already loaded classpath resources or from FAT[28] plugin assemblies containing an extended classpath file (`ResourceClasspathPluginSetupDescriptor`). For installation, the platform instantiation shall obtain and unpack the plugins as determined by the configuration model so that the setup descriptors (as determined in the respective extended classpath file) can take them up.

This combination of JSL descriptors allows for:

1. Separate, priority-based class loading for isolating plugins that require potentially conflicting dependencies.
2. Limited class loading while running the plugin as an own JVM process, e.g., in case of server instances with heavily conflicting dependencies.
3. Proxy plugins using the same classloader to enable a unified plugin architecture, e.g., if similar alternative components are loaded through (and require) priority classloading while others use plugins internally or are free of conflicts.

Java class loaders are organized hierarchically and typically the class loader of the actual class also loaded the application, i.e., may be suitable as parent for isolated class loading. However, this is not always correct as, e.g., in server environments sometimes the so-called context class loader represents the application class loader. To select the correct class loader, the class `PluginSetup` defines the class loader for oktoflow. Dependent on the startup, e.g., of apps, this class loader may be re-defined adequately and shall be used for any dynamic loading operations.

3.3.2.2 JSL support

Initially to cope with different versions of JSL in different JDK implementations, we created a set of utility methods in the class `ServiceLoaderUtils`. Since the introduction of the plugin mechanism, the class loader to be used for JSL is also taken from `PluginSetup`, i.e., all service loaders in the platform shall be created via `ServiceLoaderUtils`.

3.3.2.3 Resource Loader

In many cases, platform components rely on (file) resources that must be resolved and loaded at runtime. In Java, this usually happens via the class loader (in oktoflow the one in `PluginSetup`), i.e., Java archive files (JARs) contain such resource files and the Java class loading mechanism provides transparent access to them. However, besides the standard path starting at the root of such an archive file, in some cases the packaging of FAT JARs may dictate further paths. In the platform, FAT JARs are an alternative for packing app services into service artifacts. As an unknown number of additional

[28] FAT Java archive files (JARs) are specialized ZIP files in which dependencies are included, either as contained JAR files or dissolved into individual files or folders.

resolution strategies may be required, `support.boot` realizes the `ResourceLoader`, which allows registering additional `ResourceResolver` instances directly or via JSL. All platform components are encouraged to utilize the `ResourceLoader` or to contribute required resolution strategies.

3.3.2.4 Task Tracking

For longer running tasks, such as service deployments, tracking and reporting the state of the execution to the user is required, e.g., on a user interface. However, the platform is a distributed system, i.e., task information must be passed among the executing resources in a manner, that multiple resources can collaborate on the same task. For this purpose, the oktoflow platform provides a thread-based task tracking mechanism.

3.3.3 The support Component

Below, we detail the `support` component as illustrated in Figure 8. Besides further plugin interfaces for REST (server), HTTP (client), websockets, ssh, Java bytecode manipulation and runtime metric probes, this component also implements basic pogram setup classes, a registration mechanism for already installed programs/dependencies required by the platform or services as well as a runtime data collector for tracing build and testing times. The mechanisms defined in this component are allowed to utilize the plugins from `support.boot`, i.e., logging, commons, YAML, Json, also for testing.

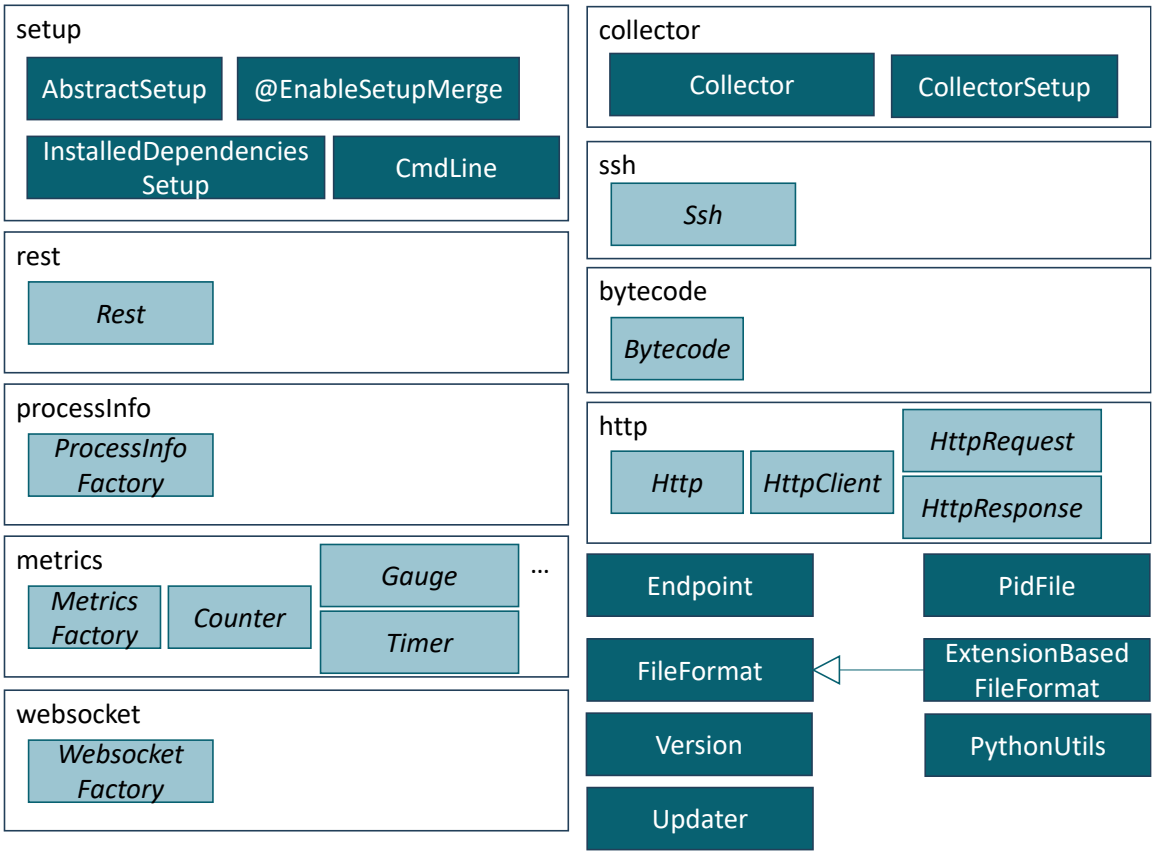

Figure 8: Simplified structure of `support`*: core implementing classes, plugin interfaces (lighter background), utility classes like* `PidFile`*. Plugin descriptors are not shown.*

3.3.3.1 Setup

As stated above, we differentiate between a Configuration (overarching, for the entire platform and its applications, basis for code generation) and Setup (instantiated configuration information to be read by platform components upon startup. Inspired by the Spring framework, we represent the startup information uniformely in YAML. Further a simple mechanism to uniformely read command line arguments (also in Spring style) is implemented here.

3.3.3.2 Installed Dependencies

Services may require individual technical dependencies, e.g., a certain version of Java or Python. However, operating systems like Linux ship with certain versions of these dependencies, which, in turn, affect also the available versions of related libraries. Similarly, Windows makes assumptions and even, during updates, suddenly tends to switch to newer versions, e.g., of Python. To allow the platform and in particular the applications and services to run with expected versions of such underlying technical dependencies, we introduce the "installed dependencies" mechanism. This is a simple YAML file which declares the paths of the executables for certain keys, e.g., `PYTHON39`. As services in the configuration model of the platform can specify also the required system dependencies, the service implementation can request the actual path from the installed dependencies and execute the respective binary. In particular, for generated containerized environment, the platform ensures that the required dependencies are installed and respective paths are declared in the installed dependencies file. For application testing, it is important to know that the installed dependencies YAML file is searched primarily in the Java classpath, the operating system root (intended for containers) and in the location specified by the Java system property `iip.installedDeps`.

3.3.3.3 Other classes

`support.support` also defines abstract file formats, a `Version` class (representing platform and service versions), the `PythonUtils` and the dependency `Updater` for plugin dependencies. While some of these classes may be migrated in the future to `support.boot`, the `Updater` is one class which needs Json functionality, i.e., an implemented plugin, and, thus, must reside one "layer" above the required plugin interfaces.

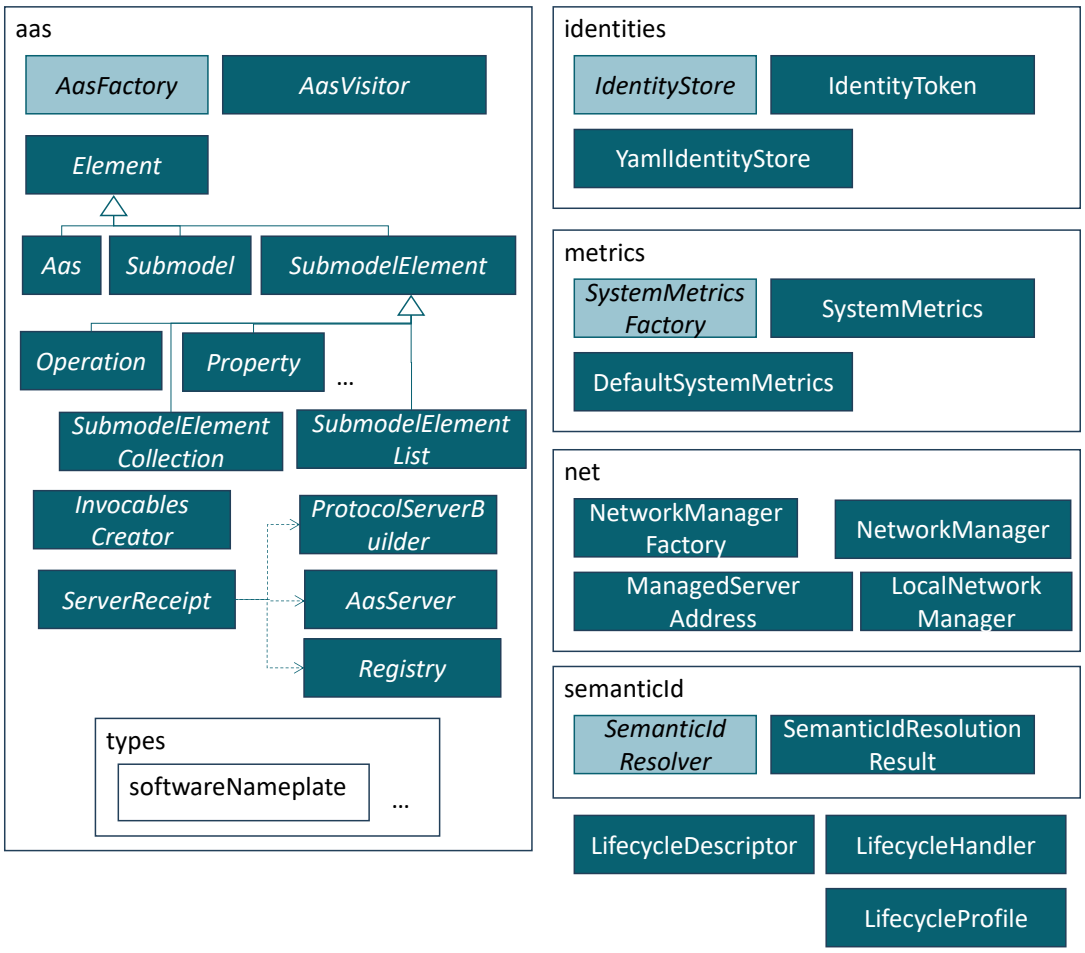

Figure 9: Simplified structure of `support.aas`*: AAS abstraction, core implementing classes, plugin interfaces (lighter background) and the platform lifecycle mechanism. Plugin descriptors are not shown.*

3.3.4 The support.aas Component

Reusing the mechanisms of `support` (and transitively of `support.boot`), the `support.aas` component introduces the interfaces for over abstracting Asset Administration Shell implementations, the identity (authentication) mechanism as well as basic mechanisms to be used as implementations for AAS submodels, e.g., system metrics, the distributed network manager and semantic id resolution.

3.3.4.1 Asset Administration Shell Abstraction

A core aim of this component is to abstract over the used AAS implementation. This allows for flexibility (the AAS implementation can be exchanged), embraces oktoflow's plugin mechanisms for isolating technical dependencies, 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).

Currently, we employ BaSyx1 as the default AAS implementation of oktoflow and, besides three different versions of BaSyx1 (AAS metamodel/API v2) realized as plugins, we also provide in the same fashion an integration of BaSyx2 (AAS metamodel/API v3). Please note that not all classes and types defined by the abstraction are depicted in Figure 9, e.g., there are also multi-language properties, reference elements or entity elements.

The `aas` component mainly consists of the instance factory (`AasFactory`) as well as interfaces defining the functionality to be provided by an AAS implementation[29]. It is important to distinguish here between AAS interfaces (such as `Aas`, `SubModel`, `Property` and `Operation` following the AAS meta-model) and the associated (nested) builder interfaces used to create concrete instances of these interfaces in an abstract manner, i.e., so that the client code needs no knowledge about the actual underlying instance creation approach employed by the AAS implementation. The AAS interfaces provide access to the respective information and, to a certain degree, also allow for modifications of local or deployed AAS elements. Moreover, the builder interfaces allow 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 `AasFactory` 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 `ProtocolServerBuilder`) 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 (as realized by the underlying AAS implementation). This (related) `InvocablesBuilder` creates function objects delegating the respective operation to the protocol/implementation, while the corresponding `ProtocolServerBuilder` registers these function objects with a matching server implementation. Ultimately, the `AasFactory` is responsible for creating a matching pair of instances for a given protocol.

[29] We follow a pragmatic and agile approach here, i.e., we follow the meta-model, 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.

In addition, the AAS 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, a visitor instance can be applied to any element in the data structure and, thus, also 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.

In situations, where many AAS elements shall be created, deployed or manipulated in a short time, the AAS abstraction, which, by default, applies implicit caching of the AAS element instances, may be a performance or resource usage obstacle. For such situations, caching can be disabled, e.g., below submodel level, and certain operations allow for a direct non-cached interaction with the underlying AAS implementation, in particular the operations `create` (receiving a parent-level builder instance) and `iterate` (receiving a temporary, non-cached AAS abstraction instance of type-filtered submodel elements).

Along with the further evolution of the AAS concept, more and more standardized AAS structures will be defined. One such structure is the Technical Data Submodel [BBB+20] including manufacturer information, nameplate etc. The AAS abstraction layer takes up relevant submodel specifications (in `types`) and allows to create and read such structures in terms of an API based on the AAS abstraction. Since version 0.7.0, we provide here generated generic, uniform realizations based on the abstraction for all realized AAS implementations of the

- Generic Frame for Technical Data for Industrial Equipment in Manufacturing [IDTA 02003-1-2]
- Handover Documentation [IDTA 02004-1-2]
- Hierarchical Structures enabling Bills of Material [IDTA 02011-1-0]
- Draft Submodel PCF [IDTA 2023-01-24]
- Time Series Data [IDTA 02008-1-1]
- Submodel for Contact Information [IDTA 02002-1-0]
- Nameplate for Software in Manufacturing [IDTA 02007-1-0]

A concrete implementation of the AAS abstraction provides an `AASFactory` along with required (plugin) descriptors and impementations of the elements. 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 from the `AASFactory`. Concrete AAS factories are supposed to be realized as dependency isolating plugin announcing themselves via the `AasFactoryDescriptor`. Multiple AAS implementations may be part of a specific platform installation, one playing the role as default plugin (used for platform operations), while the others can be requested in specific situations, e.g., an AAS connector may state the specific plugin id of the underlying implementation to use.

The current default implementation of the AAS abstraction is based on Eclipse BaSyx. The `aas.basyx` plugin and similary the `aas.basyx2` plugin implement the interfaces, typically in terms of adapter/wrapper[30] classes, i.e., classes that delegate the actual operations to the underlying BaSyx implementation. Each of the plugins ships with its own communication protocols, e.g., BaSyx1 with the Virtual Automation Bus (in variants TCP, HTTP and HTTPS) while BaSyx2 relies on operation delegation through REST. 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

[30] https://en.wikipedia.org/wiki/Adapter_pattern

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.

3.3.4.2 Network Management

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. Furthermore, the network management support can record the number of instances accessing a certain service represented by it's known key. This is in particular important if services shall be started/stopped dependent on the actual use, i.e., if no further instance is using a service it can be stopped and the resources can be freed.

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.

3.3.4.3 Platform Component Lifecycle

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, this component defines the `LifecycleDescriptor`, 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. Furthermore, to handle conflicting functionality, the `LifecycleProfile` specifies a set of `LifecycleDescriptor` instances to be executed when the profile is stated as command line parameter of the component startup. These profiles also allow for virtualization of such partial component lifecycles.

3.3.4.4 System-level Monitoring Support

System-level properties such as number of CPUs or GPUs, their actual load or temperature are particulary difficult to access in Java. Moreover, edge devices may have vendor specific interfaces including OPC UA or MQTT to access such information. To enable the generic use of such information, also in the platform AAS, we included the required basic access functionality as an interface and a rather simple default implementation into the support layer. Specific implementations can be added via JSL. One example is the `support.dfltSysMetrics` plugin, which relies on JSensors[31]. Alternatively, the process information plugin interface could be used/extended.

The platform includes an optional system-level monitoring plugin for Phoenix Contact PLCnext, which accesses some system properties like CPU or board/case temperature via GRPC/protobuf provided by PLCnext (starting with firmware released in 2022). Similary, oktoflow provides an optional system-level monitoring plugin for the Bitmotec Bitmoteco system.

[31] https://github.com/profesorfalken/jSensors

3.3.4.5 Identity Support

Some mechanisms in the platform require a certain form of authentication, ranging from anonymous over username/password up to X509 tokens[32], keystores with certificates or (public) cryptographic keys as well as SSL key managers. However, storing such information in the configuration model or even in code is not acceptable. Therefore, the platform provides an `IdentityStore` with a pluggable implementation. By default and in particular for demonstration installations or testing, a YAML file with the identities is read either from the classpath, a file from the home directory of the actual process or a file determined by an environment variable. Moreover, advanced and sophisticated implementations for central identity and authentication token management can be plugged in here. Upstream components shall refer to an identity through a logical logical name, which provides access to the registrered authentication token provided (if known) by the identity store. To allow for more flexibility and to ease identity management, several default names, e.g., starting with a specific device name, if not found, the name of a device group, e.g., edges or servers, etc. can be used.

An example for a named YAML identity store is shown below:

```
name: HM'22
identities:
    "amqp":
        type: USERNAME
        userName: user
        tokenData: p**
        tokenEncryptionAlgorithm: UTF-8
```

The `name` of the store is used for logging when the identity store is loaded, i.e., which identity store is actually taken up in case that a differentiation among alternatives is needed. The `identities` are described as token objects, here the identity token with key `amqp` as a `username` token for user `user`, password `p**` and token "encryption" algorithm `UTF-8`. If a file entry is specified, e.g., pointing to a relative keystore, the token data is used to open the keystore and, depending on the keystore type, may then omit the user name.

3.3.4.6 Semantic Id Resolution Support

One specific ability of AAS is to mark used elements with a so-called semantic identifier, i.e., a reference to a dictionary detailing what is contained in a certain AAS element. With increasing use of semantic identifiers in the platform AAS, also a resolution of these identifiers becomes important, e.g., on the user interface to display associated value units and descriptions. Besides ECLASS[33] IRDI identifiers, also URL-like IRI identifiers are used, e.g., in the specifications of AAS submodel formats. A semantic id resolution mechanism must take care of such identifiers, potentially considering mechanisms implemented by the AAS framework as well as potentially commercially licensed access to catalogs and web services as they apply for ECLASS.

For this purpose, oktoflow provides a flexible semantic id resolution support. The `SemanticIdResolver` interface provides access to the resolution mechanism. The result of a successful resolution (inspired by the ECLASS dictionary) returns the version, the revision, and, in multiple languages, the name, structure name and a free text description of the referenced value unit or concept. The actual resolution shall be realized in terms of oktoflow plugins and, as fallback, through fallback catalogues provided by the platform (to be able to at least resolve the semantic ids required

[32] Originally, a generic form of identity tokens was provided by the `connectors` component, mainly for OPC UA. This now became a more general mechanism of the platform.

[33] https://eclass.eu/ we are grateful for the support of Eclass and the ability to us the Eclass catalogue in the context of a research license.

on the user interface even without internet access). One example plugin performs online resolution using the ECLASS web service relying on the identity management (Section 3.3.4.5) to access the required authentication certificate.

3.3.5 The support.iip-aas Component

The `iip-aas` component on top specializes the AAS abstraction for use within oktoflow[34], e.g., further functionality that eases the realization of the platform, e.g., mechanisms how to dynamically link alternative and optional AAS sub-models of different components into the platform AAS as illustrated in Figure 10.

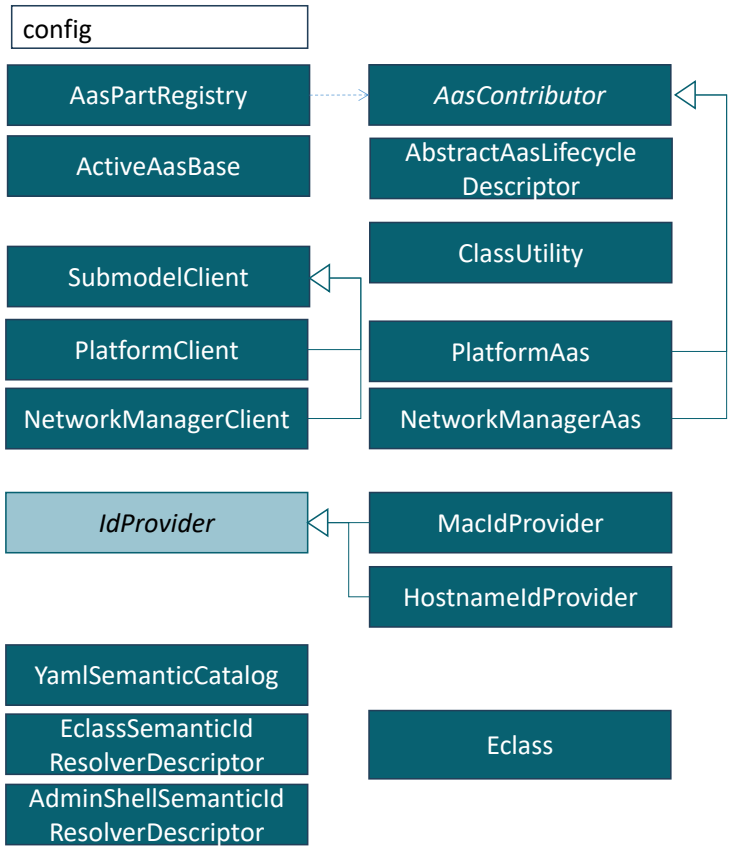

Figure 10: Simplified structure of `support.iip-aas`*: Dynamic AAS realization mechanisms, basic AAS structures relying on* `support.aas`*, device identity providers and fallback semantic id resolution.*

One basic ability is that AAS (sub-models) for the different platform layers can be collected and deployed as a single representation 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-models) and to register the implementing function objects with the protocol builders of the AAS abstraction in `support.aas`. 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 JSL. The `AasPartRegistry` provides access to those plugin instances and, e.g., triggers the creation and the deployment of an entire AAS for an installation. These contributor instances are also used to increase availability (R9) and resilience of the platforms with respect to downtimes of the AAS. If the AAS server disappears, as a basic mechanism the deployment of contributed AAS and sub-models is executed again to re-populate the AAS server and to allow for

[34] As the initial name was IIP-Ecosphere platform, there is still the “iip” in some names.

continued (management) operations. The `AasPartRegistry` also maintains the AAS setup information (supported by the classes package `config` – to be renamed to setup in the future).

To ease the development of platform components that supply an own AAS, `support.iip-aas` defines a basic lifecycle descriptor for AAS-contributing components. Further, the `ActiveAasBase` realizes supporting utility methods for typcical AAS runtime modifications, in particular for executing them asynchronously (production code) or synchronously (test code).

The next set of classes shown in Figure 10 focuses on the realization of specific AAS as well as access to the contained structures from platform code. `SubmodelClient` is a base class for all platform parts that need to access AAS submodels, for reading individual values, writing values or, in particular, for calling operations. Subclasses of `SubmodelClient` shall add specific operations and, thus, to provide a helpful code-level API to the AAS submodels. This is illustrated by the platform (nameplate) AAS submodel (`PlatformAas` as `AasContributor`) and its `PlatformClient` as well as by the `NetworkManagerAas` and the `NetworkManagerClient`. The `NetworkManagerAas` wraps the network manager interface from support.aas and, in particular, acts as a distributed network manager. The `PlatformAas` is an extended nameplate with certain overarching platform operations, in particular for semantic id resolution. Further helper functions support, e.g., the resolution of images, e.g., for AAS nameplates or the generic representation of Java data classes in AAS as realized by the `ClassUtility`

For semantic id resolution, the for oktoflow relevant fallback cataloges (based on the generic YamlSemanticCatalog) are realized, i.e., a simplifying small excerpt of ECLASS (with associated constant definitions in `Eclass`) and AAS IRI definitions from [ZVEI-N].

3.3.6 AAS Creation and Usage Pattern

For using the provided capabilities in an upstream component or layer, we suggest the creation and usage/design pattern illustrated in Figure 11. As stated in the previous sections, we are using the AAS abstraction (`support.aas`) as a frontend, i.e., through the `AasFactory` without direct dependencies to the underlying AAS implementation. The platform-specific AAS library (`support.aas.iip-aas`) provides helpful classes and mechanisms that we use in this pattern.

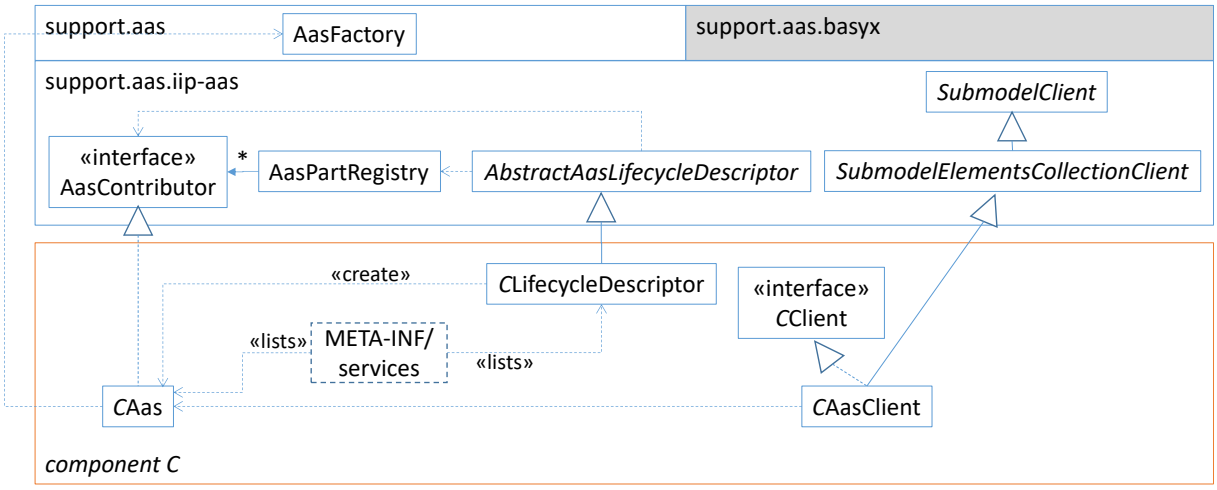

Figure 11: AAS creation and usage pattern involving support layer classes and mechanisms.

To illustrate the pattern, some classes of `support.aas.iip-aas` as well as some classes of a prototypical component providing an own AAS (*component C*) are depicted in Figure 11. The AAS of *C* is implemented in *C*`Aas` (typically using the name of the component or a suitable shortform as prefix of the class names following Java and platform conventions for the naming, e.g., *C* could be `ecsRuntime`, the AAS could be in `EcsAas`). *C*`Aas` uses the `AasFactory` to create sub-models,

properties and operations. However, to be part of the platform AAS, *C*Aas is also an AasContributor, which defines methods for creating a sub-model (for a given AAS) and for hooking into the AAS implementation server using the recipe interfaces of the platform's AAS abstraction. To become active, *C*Aas (or the AasContributor, respectively) are specified via JSL and, through JSL, become automatically active in the AasPartRegistry.

However, to obtain a single, central AAS server and to hook the individual parts into that server with the right setup information, we need a lifecycle descriptor. A basic form, that creates also the AAS platform server instances if needed, is provided in terms of the AbstractAasLifecycleDescriptor, which utilizes the AasPartRegistry to build the AAS. To become active, the *C*LifecycleDescriptor must be specified as JSL service. In this combination, the AAS of component *C* is created at the right point in time in the life cycle (of the containing platform componend) and automatically deployed to or registered with the platform AAS. During this creation process, also further AAS may be created, e.g., to represent a device AAS including vendor information [ZVEI-N, BBB+20].

For using the information in the AAS during the execution of other platform components, one could now request the platform AAS instance from the AasPartRegistry and operate on it through the abstraction interfaces provided by support.aas, e.g., to find a certain operation and to call it. However, if all platform parts do that directly, evolving the structure of individual sub-models becomes nearly impossible (or simply a mess). Thus, each component defining a part of the platform AAS shall also provide a (submodel) client implementation. For this purpose, support.aas.iip-aas provides basic client implementations, e.g., the AbstractSubmodelClient (for properties and operations defined on sub-model level) or the AbstractSubmodelElementsCollectionClient (for an element located in a submodel elements collection in a certain sub-model). The component providing the client shall now define an interface for the respective operations (*C*Client) and implement that interface in terms of either a specialized basic client, in Figure 11 shown as *C*AasClient. Upstream components that want to access the AAS, shall use the client interface and the concrete client implementation. While the *C*Client interface does not seem to be required here, it helps testing against mocked instances, e.g., in the command interface of the platform.

For modeling AAS operations, we follow the convention, that usual AAS operations behave like synchronous calls and, thus, must not be tracked via the TaskTracker in support.boot. Top-level AAS operations that shall be tracked shall be equipped with name suffix "Async", return their task identification immediately and continue running in parallel. Lower-level operations that can be tracked are marked with the name suffix "ByTask", offer an additional parameter "taskId" and use the task id for reporting their status. The actual distributed status reporting is realized in the transport layer.

As we started our work on oktoflow using very early implementations of AAS frameworks, e.g., when no user-defined types were available for properties or operation parameters/return types, we still simplify the modeling. For AAS properties, we usually rely on primitive types. Where possible, we avoid complex types in operation parameters and, if required, use JSON strings to transport complex or multiple values, e.g., objects, arrays or maps. Thus, to simplify later code revisions of the platform and to avoid conflicts with, e.g., annotation-based JSON libraries, we decided to provide some support for JSON marshalling in iip-aas, e.g., to handle return values and alternative exceptions that may occur during operation execution. Similarly, as there were no mechanisms to programmatically resolve AAS references, we decided to represent references as Strings carrying the name of an element in a submodel element collection denoted by dependencies or associations or as URLs.

3.3.7 Plugins

The support layer also implements most of the oktoflow plugins. Table 4 summarizes the core plugins defined/used by the support layer. Plugins can be utilized through the PluginManager or, in

particular for testing, as classical dependency via JSL. Through the `PluginManager`, usually dependency separation through isolated classloading is achieved, i.e., while the oktoflow core is free of direct dependencies, implementation components such as connectors may use these plugins or rely on own dependencies. In contrast, using plugins as dependencies does not lead to isolatated loading and, thus, must be handled with care, i.e., cannot be applied in all situations.

Table 4: Summary of core plugins in the support layer.

Plugin	Purpose	Based on	Default-Impl.	As Test-Dependency
`support.log-slf4j-simple`	Logging	slf4j[35] including slf4j-simple	x	exclude slf4j
`support.yaml-snakeyaml`	YAML reading/writing	snakeyaml[36]	-	
`support.json-jackson`	JSON reading/writing	FasterXML/Jackson[37], glassfish[38], jsoniter[39]	-	
`support.websocket-websocket`	Websocket client/server	Java-websocket[40]	-	
`support.processinfo-oshi`	Native process information	OSHI[41]	-	
`support.rest-spark`	HTTP/REST server	spark[42]	-	
`support.http-apache`	HTTP/REST client	Apache HttpComponents[43]	-	
`support.commons-apache`	Common utility functions	Apache commons[44], jodatime[45]	-	
`support.ssh-sshd`	SSH client/server	Apache Mina SSHD[46]	-	
`support.metrics-micrometer`	Monitoring probes	micrometer[47]	-	
`support.bytecode-bytebuddy`	Java bytecode manipulation	bytebuddy[48]	-	
`test.amqp.qpid`	AMQP broker for testing	Apache QPID	-	

One special case is the logging plugin which, for which an implementation is already required before and when the `PluginManager` is started, thus, we provide a default/fallback implementation.

In special situations, in particular for the logging and the metrics plugin, it may make sense to rely for a more consistent integration or to reuse exisng setup/instances on the version provided by the respective platform component (e.g., the Spring Cloud Stream plugin for service execution). Then the

[35] https://www.slf4j.org/
[36] https://github.com/snakeyaml/snakeyaml
[37] https://github.com/FasterXML/jackson
[38] https://mvnrepository.com/artifact/org.glassfish/javax.json
[39] https://jsoniter.com/
[40] https://github.com/TooTallNate/Java-WebSocket
[41] https://github.com/oshi/oshi
[42] https://github.com/perwendel/spark
[43] https://hc.apache.org/
[44] https://commons.apache.org/
[45] https://www.joda.org/joda-time/
[46] https://mina.apache.org/sshd-project/
[47] https://micrometer.io/
[48] https://bytebuddy.net/#/

dependency to the underlying implementation used in the plugin can be excluded in the component's POM and, as it is implicitly replaced by the provided dependencies of the component at hands. In such cases, the tests of the "customized" plugins shall be executed as part of the component tests to ensure future compatibility. Akin to the discussed plugin, all implementations of upstream platform components have been turned into plugins for isolated loading. The platform instantiation may decide whether plugins or usual (JSL) dependencies shall be used.

Besides the plugins mentioned in Table 4, further upstream components, e.g., the service execution (for Spring Cloud Stream) or the configuration modeling/code generation (EASy-Producer) constitute own plugins, primarily for dependency isoluation, but also to allow exchanging the respective technology if desired.

3.4 Transport and Connection Layer

The Transport and Connection Layer is responsible for connecting devices, services and resources among each other. We will discuss the two interrelated components in this layer, the Transport Component (Section 3.4.1) for the low-level platform-internal data transport and the Connectors Component (Section 3.4.2) for external data input/output.

3.4.1 Transport Component

The Transport Component is responsible for turning objects into a specified wire format and to transport the data using that wire format from a sender to a receiver, e.g., among (distributed) services. Wire format and transport protocol shall be exchangeable and extensible. The Transport Component is in particular responsible for fast (soft-realtime) communication while, in contrast, AAS is more for storing stable data of low frequency changes and for representing (distributed) operations/component interfaces. This decision was made based on early experiments [Sta20], where AAS operation calls showed a round-trip time of 23 ms and property accesses of about 4-10 ms. For comparison, plain Java Remote Method Invocations operate in this setup at 2-4 ms, which may impact the required 8 ms machine pulse in R28 if multiple sources/sinks are involved.

Please refer to older versions of this handbook for a discussion of potential data transport and data streaming technologies and how we made our decision for the technologies integrated into oktoflow.

3.4.1.1 Design

Figure 12 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.
- To avoid creating transport connectors again and again, `Transport` holds a global (inter-device) and a local (intra-device) transport connector.

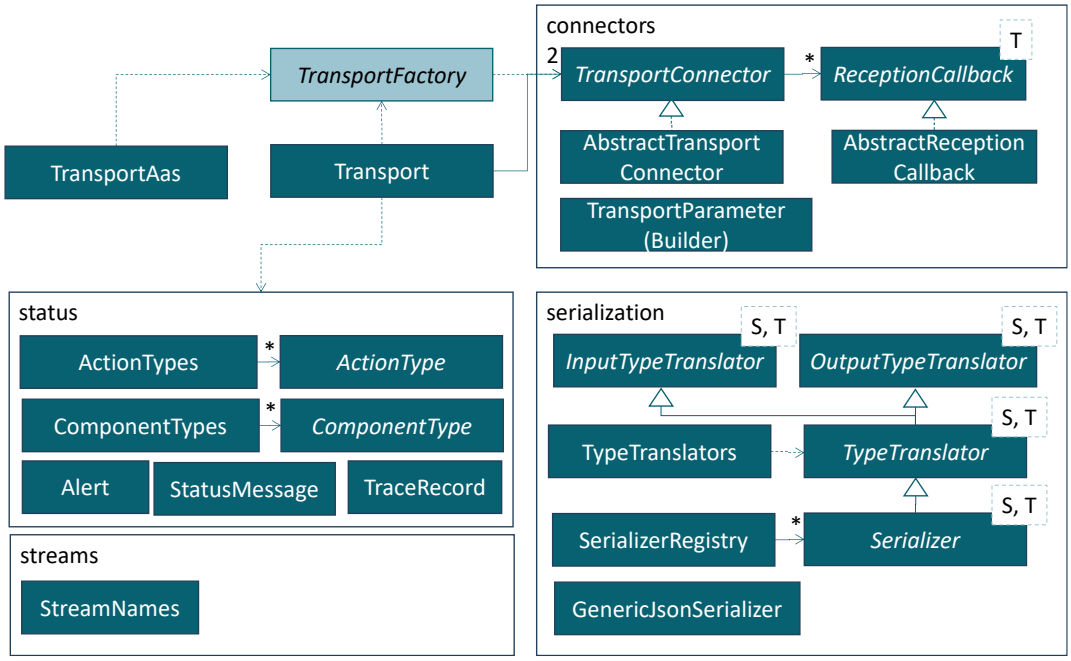

Figure 12: Transport Component overview (comments partially cropped)

- 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 UA may 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. 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 `OutputTypeTranslator` with cross-over template bindings[49]. Intentionally, we leave the actual technical approaches for transcoding open here (some candidates are JSON, OPC-JSON or protobuf[50]). 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`. Certain default type translators for primitive types are defined in `TypeTranslators`.
- 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` (basis for transport plugins) and exhibit the actual protocol, the wire format and the broker data connector(s) from the platform configuration in the `Transport` AAS.

[49] 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.

[50] https://developers.google.com/protocol-buffers

- Three default protocol plugins are shipped with the 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 default protocol plugins support optional Transport Level Security (TLS) and, thus, contribute to the realization of R40.
- The streaming approach currently located in 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, e.g., to add serializers to the `SerializerFactory` through the component setup 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 platform ships with six alternative Spring Cloud Stream protocol binders, a generic one just using `Transport`, one 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). These binders support optional Transport Level Security (TLS).
- The transport component defines several global platform streams (`StreamNames`), e.g., for status (`StatusMessage`), alert (`Alert`) and trace (`TraceRecord`) messages or, as forward declarations, for upstream components, all accessible via `Transport`. The status notification mechanism informs interested parties when containers or services are dynamically added or removed. The notifications consist of a message data structure, which is sent on a pre-defined transport channel. Further, status notifications may report on their progress and may be task-related, i.e., carry a task id (cf. Section 3.3.2.4). As task-based execution does not have a result, status messages may carry result or failure information in this case. Alerts are created by monitoring components to signal abnormal or undesired situations. Traces make the operations of the platform visible. Moreover, the transport component defines a global instance of the default `TransportConnector` and send methods, that may queue messages until the transport connector can be utilized.

As several transport protocols rely on a central server instance, often called a Broker, 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. If needed, the platform shall create a matching (test) broker implementation during platform instantiation. Further, the platform configuration provides 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, different levels of resilience or recovery can be realized, while failover to alternative broker servers may require additional implementation work.

3.4.1.2 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 discussed streaming capabilities. This is reflected in the communication setup shown in Figure 13. 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 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 3.4.2) 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.

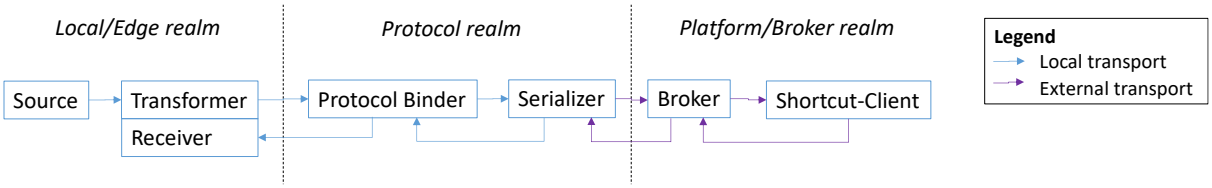

Figure 13: Regression testing data flow for the Transport Component.

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 14 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 14 can be executed locally (in one process, in multiple processes) or distributed on separated hosts as indicated in Figure 14. 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.

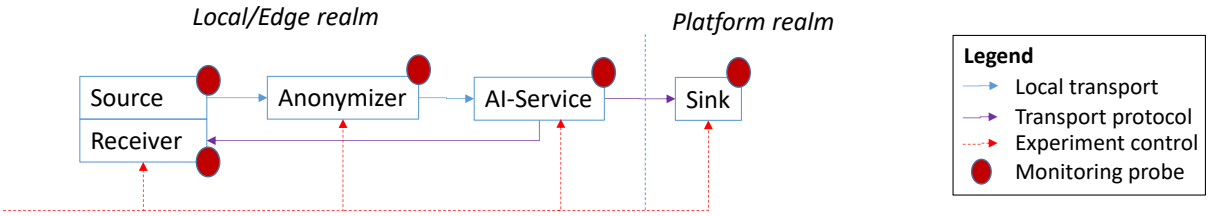

Figure 14: Performance testing data flow for the Transport Component.

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 14 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[51], an asynchronous networking library, and the network port management of the platform. For the source, we use a message ingestion rate[52] 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.

Figure 15 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 15. Moreover, there are noticeable differences in settling time for the average throughput (not shown in Figure 15): 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.

[51] https://netty.io/

[52] 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.

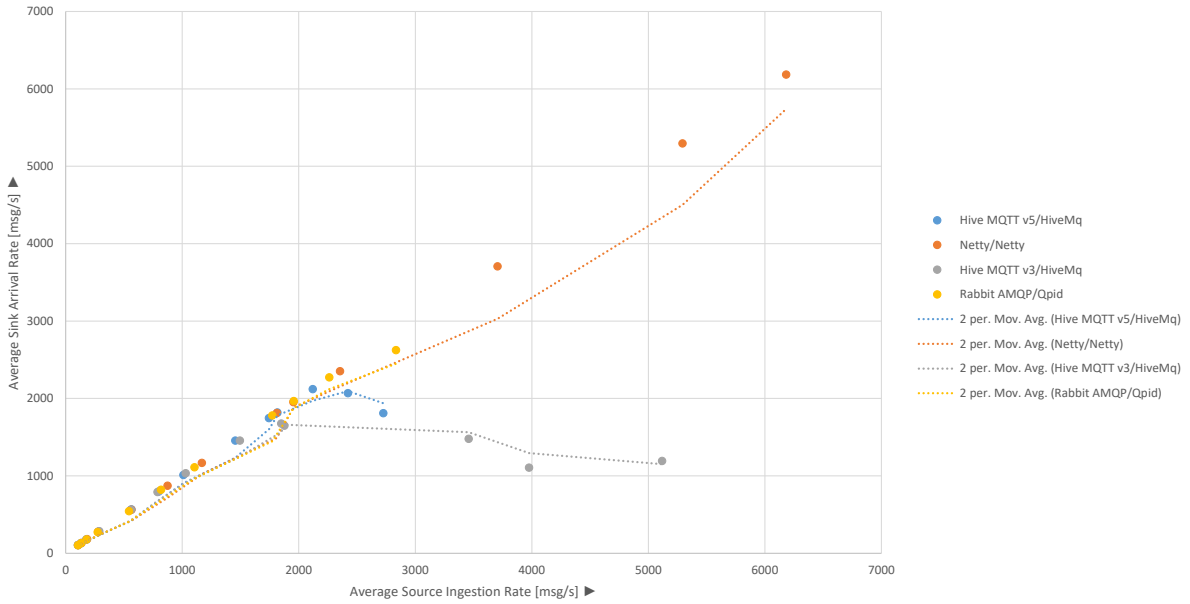

Figure 15: Average stream throughput measures for the four utilized alternative binders with trend lines.

As Figure 15 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 5 details these numbers for the measured protocol-client-server combinations. In particular, our HiveMq readings amount to similar ranges as reported in [KGR20], where two server machines with up to 16 CPU cores but no stream processing approach were used.

Table 5: 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[53]. 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

[53] Based on the transferred messages in Table 5, this leads to 13.5 GBytes up to 66 GBytes per hour.

parts of the platform are available that potentially impact the data size or the performance (meta-information, security, etc.), further experiments shall be performed.

3.4.2 Connectors Component

The Connectors Component is responsible for the communication with external data sources and sinks, e.g., machines (potentially connected via some form of edge devices) or already installed platforms (the virtual platform aspect). The aim here is to allow for a bi-directional, typed communication.

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 platform. For the outbound direction, (AI-)services or humans may send instruction or configuration changes to 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 turn the application-specific data types received from the platform side into information suitable for the external side. As stated in Section 3.4.1, application-specific type translators are realized by code generation to ease the development of applications.

We differentiate between:

- **Connector type:** Generic implementation of the platform connector interfaces for a certain protocol or information model, usually by wrapping an existing implementation (usually an external library or a framework). Connector types are implemented manually based on the connector component of the platform as an optional platform plugin.
- **Connector instance:** A connector type is created at runtime by an application after it was adapted for the specific application context, e.g., by adding type transformers for data types used in the application. These adaptations are realized through code generation, while some specific adaptations may be hooked in manually. To be considered, new connector types must be added appropriately to the configuration meta-model.

Usually, when we talk about an implementation of a connector, we implicitly refer to the respective connector type. Similarly, when we talk about the use of a connector in an application or it's generation, we man a respective connector instance.

Regarding related approaches, please refer to one of the previous versions of this handbook.

3.4.2.1 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.

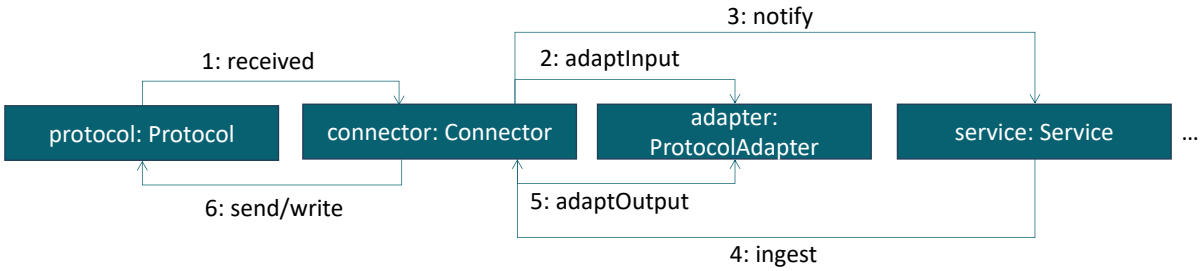

Figure 16: Event-based connector and push-based protocol-adaptation.

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 16, 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 16). When the data is translated, the respective instance is passed on to a registered streaming `Service` in asynchronous manner. For the outbound direction (not shown in Figure 16), the `Service` ultimately receives the data as a stream and passes that on to 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`.

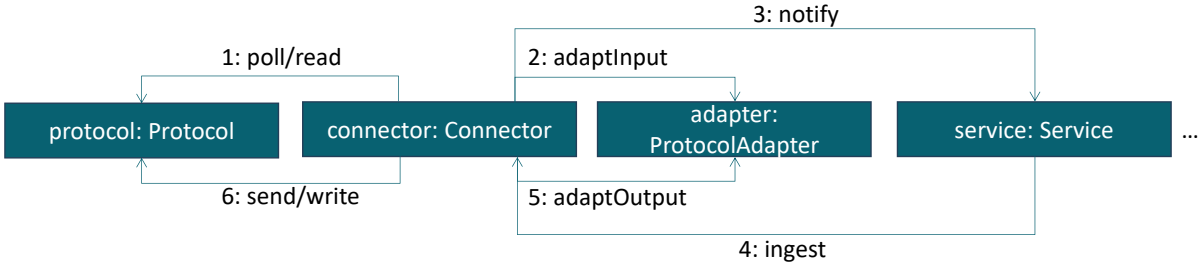

Figure 17: Poll-based connector and subsequent protocol adaptation.

Protocol implementations not offering such notifications are subject to **polling**. One example here is the current OPC UA implementation that we use, were OPC change events just indicate a change of the model but not of which element. In its basic form, polling repeatedly obtains information from a source and ingests the information regardless whether the information changed in the meantime. This may be intended, e.g., to realize equidistant input. However, if not desired, it can also unnecessarily allocate transport resources. To avoid this, a caching mode can be defined for a connector instance. The mode indicates whether there shall be **no** caching, i.e., ingestion of all received data, ingestion only if **hash** codes are different or ingestion if the contents of the data is not **equal**. While a comparison based on hash codes is typically faster than a comparison for equality, it may also fail if hash codes accidentally are the same for equal data items (hashing collision).

As illustrated in Figure 17, the Connector then actively (based on connector settings) polls information from the "Protocol". As before, the Connector consults the ProtocolAdapter and notifies the registered Service about the data to be ingested. The outbound direction works as discussed for event-based ingestion. Realizing the polling cycle in the Connector rather than the Service allows for connector-specific polling strategies as well as for a uniform interface towards the stream-based data processing in the platform.

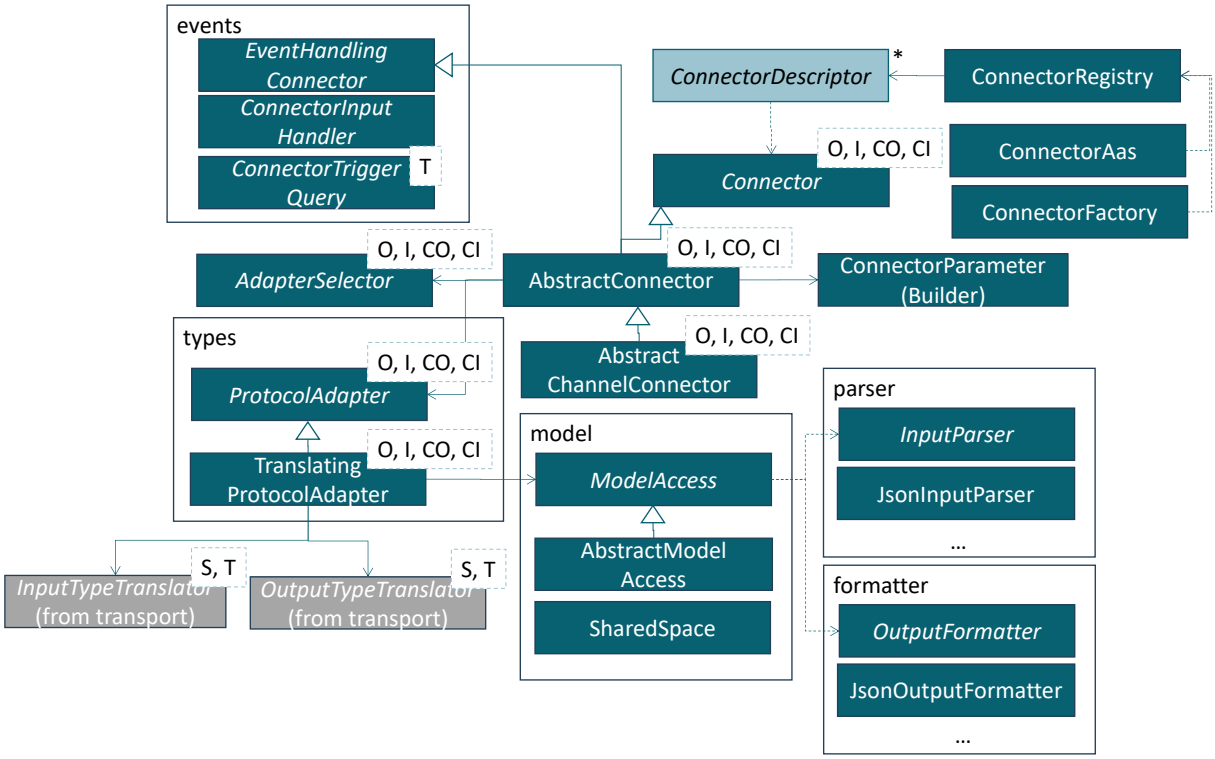

Figure 18: Overview of the Connectors Component.

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 18 presents an overview of the main classes in the Connectors Component of the platform. The component consists of:

- The Connector interface represents an external data source/sink. Connectors based on an information model shall exhibit a ModelAccess instance to interact with the information model rather than the payload. 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 data sink and O for output issued by the data source. A Connector can be connected to the data source/sink 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.
  All types used in CI, CO must be types used for data transfer among the services, preferably their interfaces. These types are generated by oktoflow based on the application model. CI,

CO must not be any internal types used by the connector implementations. In contrast, `I, O` may be connector-specific types as they just represent the external/machine side and are not used further in the hosting application.

- The `TranslatingProtocolAdapter` is a default implementation of the `ProtocolAdapter` and relies on type translators, i.e., `InputTypeTranslator` and `OutputTypeTranslator` defined in 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 underlying data model. The input type translator is responsible for turning generic values or data transport types to connector internal types, e.g., needed for implementing certain protocols and vice versa the output type translator is reposnible for the oppsite direction. In most cases, generic type translators for objects can be used[54].
- 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 `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. Further, connector instances provide access to their input/output data types. Ultimately, connector instances link to their descriptors in the `installedConnectors` sub-model to indicate their origin and capabilities.
- The `ConnectorFactory` is a proxy to dynamically create the most appropriate connector instance if there are alternatives, e.g., MQTT v3 and MQTT v5. Such a `ConnectorFactory` takes the `ConnectorParameters` and may decide on the supplied device service information (if present) on the actual connector type. An implementing connector may rely on existing connector implementations (as we do for MQTT).

As we will see in the next section, a connector instance can be wrapped into a service to be executed by the service management of the platform. Thereby, multiple connector instances of the same connector type (handling different types of input/output, specified in the configuration model as quadruples of CI, CO, I and O), can be wrapped into a single service so that the four template parameters of a connector type are actually no limitation for data flow modeling. If, e.g., for resource consumption reasons, underlying instances of the implementation shall be shared, a connector can create a `SharedSpace` which is passed among the connector instances in the same wrapping service.

Currently, nine specific connector types are realized in terms of individual oktoflow plugins. These are the

- generic **`AasConnector`** for integrating external AAS into the platform (based on the `AasFactory` from the Support Layer),

[54] In progress: Integration of direct reading/wirning typed access bypassing the type translators for performance reasons.

- **`OpcUaConnector`** for OPC UA 1.04 (based on Eclipse Milo)
- protocol-specific **MQTT connectors**, one for MQTT v3 and one for MQTT v5, also based on Eclipse Paho akin to the Transport Component.
- **generic MQTT connector** which selects dynamically from the MQTT v3/v5 connectors based on device information.
- **serial** connector, e.g., for connecting to EAN or QR code scanners. The actual format is provided through serializers, may be based on predefined `InputParser` and `OutputFormatter` classes.
- **MODBUS/TCP** connector for connecting, e.g., to energy meters. Supports reading/writing to MODBUS/TCP devices, translates usual 1-4 byte types and allows for configuring the device id as well as the vendor-defined byte order (little/big endian).
- **REST** connector for reading from and writing to REST resources. The rest connector implementation is abstract and must be completmented with implementation-specific class representations of the data to be handled. These class representations are created by the oktoflow code generation when the connector is used in an app.
- **InfluxDB** connector for writing to and streaming from Influx databases with Influx v2 authentication support via issued tokens and Influx v1 support for username/password authentication. Result streams are requested by simple timeseries or string queries (both requiring monotonic ascending timestamps), multiple entries per datapoint are joined into the data transport format of the platform (optional fields may be helpful) and ingested based on the timestaps of the data points in the database or, if given, overridden by a fixed data point delay given by the query.
- **file**-based connector for streaming from/writing to files in given formats. Multiple files can be read in sequence, a single file can be written. Files can be stored in the file system or may be app resources. The format is defined by the serializers to be attached, which may, in case of generated app integrations, be based on the `InputParser` and `OutputFormatter` classes of the Connectors component, i.e., a file-based connector could be used to stream CSV data while writing back JSON data. Akin to the InfluxDB connector, data read from files is streamed into apps either based on a) polling using a fixed data time difference or, through a `ConnectorInputHandler` or a `DataTimeDiffProvider` plugin b) triggering using arbitrary connector trigger queries.

The internal behavior of the connector types as well as the used interfaces differ depending on whether the connector is based on a typed (often hierarchical) information model or on payload transport, where usually the structure of the payload is not known to the transport protocol. This affects the role of the `ModelAccess`, the (payload) formatter/parser and, in turn, the `ProtocolAdapter`.

- Some protocol implementation libraries like OPC UA or Asset Administration Shells (AAS) are based on a **structured information model**. Accessing this model in a uniform manner is key for the uniform generation of application-specific connector code. We represent the access to the information 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. Typically, the generated connector code passes an initial path name to the `ModelAccess` instance and then, due to performance reasons, incrementally, indicates substructures to be accessed (`stepIn`/`stepOut` operations of `ModelAccess`) forming an incremental context within the underlying information model. For accessing a property or for calling an operation, the connector code passes a qualified name or, usually, a relative name within the actual context and requests the value of a property, writes the value of a property or calls an operation

defined on the model with respective parameters. The specific `ModelAccess` instance of a `Connector` can perform translations between value instances of the model and the (generated) types used in the platform/application. `ModelAccess` provides also opportunities to establish monitors on the underlying information model, i.e., to be notified on specific changes, as well as to register programming language counterparts of custom types defined in the model. For payload-based protocols such as MQTT, implementing the `ModelAccess` interface is not needed.

- For **payload-based transport protocols**, the payload is often binary, i.e., the structure of the (wire) format is not defined by the transport protocol. To handle binary input or output of channel connectors in a generic and open manner, introduce (payload) `parser` and `formatter`, for which wire-format specific implementations are provided. A parser allows generic access to a binary payload. In opposite direction, the formatter emits a given data type instance for a specific wire format. Currently, oktoflow provides formatters and parsers for JSON and separator-formatted data, e.g., by tabulators or commas as we found in some application cases. Parsed data is supposed to be mapped to types defined by the user in the respective app configuration. After parsing, data is either accessible via (hierarchical) names, i.e., in terms of nested data types, or in positional manner along a depth-first traversal of the defining data type. In the opposite direction, data is handed in depth-first traversal sequence to the data formatter, which produces the respective format in terms of binary data to be passed to a channel connector. Individual data values can be read in typed fashion from an input parser via an associated input converter. In the opposite direction, the output converter takes typed data and converts it into the output format.
- The role of the `ProtocolAdapter` and the involved type translators varies depending on whether the underlying protocol is payload-based or based on an underlying information model. For payload transport, the used communication channels may be relevant to the data parser-based type translation. This is taken into account by the `ChannelProtocolAdapter` and its default implementation, an extension of the `TranslatingProtocolAdapter`. For an information model-based protocol, the `ModelAccess` instance must be made available to the type translators as well as further initialization such as defining the polling mode must be carried out. This is enabled by two refined type translator interfaces, namely `ConnectorInputTypeTranslator` and `ConnectorOutputTypeTranslator`, both with a corresponding basic implementation.

3.4.2.2 Validation

The functional validation of the Connectors Component and the specific connector types happens through regression tests. Therefore, we follow the same basic idea as explained for the Transport Component in Section 3.4.1.2, i.e., we set up a corresponding protocol server/broker in a way that data sent to the server is (potentially after modification) echoed back to the connector. 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.

Further functional tests have been performed, e.g., in the context of VDW/UMATI based on externally implemented OPC UA structures. In these validations, typically first a manually instantiated connector based on handcrafted serialization or type translation is created to test the expected intake of specific formats (e.g., OPC UA or JSON via MQTT). In addition, connectors generated via the configuration model and the platform instantiation are employed (cf. Section 6). Further, we conducted performance analyses of the serialization mechanisms and the generated connectors [EW25, NE25]. The comparison

of serialization mechanisms indicated performance peaks up to factor 10 for different JSON libraries in the context of the use case studies. The generated connectors are (nearly) as fast as the handcrafted ones, sometimes sacrificing a bit time for the generic, open approach as well as a more generated schematic model-based integration.

3.5 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 3.5.1. In two further sub-sections, we turn then to the two major components of this layer.

While the service management component is generic and can be realized in the same way for all services, service environments are typically specific for the programming language used for realizing individual services. We support this in terms of a language-specific service executin environments (due to R113, at least for Python and Java) supporting a unified integration and easing the development of services. In Section 3.5.2, we discuss the Service Execution Environments.

The Control and Management component (Section 3.5.3) is closely related to the “ECS runtime” and acts as a mechanism to take control over services running on distributed 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.

3.5.1 Terminology and Background

In this section, we briefly introduce our notion of the term service and discuss the bigger picture.

Several notions for services are used, ranging from web services to microservices. In oktoflow, a **service** may receive input data, transform data, or produce output data in continuous stream-based manner. Typically, source services produce data, transformer services receive data and emit modified data, and sink services receive data. Technically, a service is (a thread in) a process implemented in any programming language, while oktoflow starts/stops/manages these threads/processes, i.e., keeps them alive and runs them continuously. Input and out data are defined in terms of types and their correct composition is controlled by the configuration model (cf. Section 6). In a service, data processing can happen synchronously, i.e., an input data item is turned directly into zero or multiple output items, or asynchronously, i.e., the service receives data and produces output data at any time later if at all. A service indicates its state (R4c), meta-information (R4b), name, identification, version, kind/category (source, transformer, sink) as well as the typed input- and output relations/data paths (R4a, [SSE21]). Moreover, it allows for certain runtime operations such as passivation, migration, runtime switch to an equivalent service, reconfiguration or (re-)activation.

We distinguish between **platform-provided services** and **application-specific services.** Application-specific services are designed and implemented for a specific application at hands. The application configuration determines the input/output types as well as relevant service properties and the platform/application instantiation generates interface, integration and template code for that service. The user implements the template code and, finally, the application instantiation assembles a complete application automatically integrating the user-supplied application-specific services. In contrast, a platform-provided service (for short platform service) is shipped with the platform and, thus, shall be applicable for more than one specific application, i.e., it shall be generic and parameterizable. Often, the service implementation is even more generic, e.g., an executable tool that

shall be integrated as oktoflow service. Such services are customized into an application, i.e., the application instantiation generates a specific service stub for the application including sufficient glue code to turn the reusable generic service into an application-specific service, e.g., by using specific input/output types or by nailing down data transport from/to the underlying service implementation as well as the passing of parameters. Moreover, **hybrid services** may occur, typically generic platform services that can operate without application specific code, but which can be customized through add-ons or plugins that turn an instance of the hybrid service into an application-specific service. Thus, hybrid services can ease the realization of application-specific services and still act as platform-provided services.

Further, it is important to answer the question "**Where do services come from?**". Details of the mechanisms will be introduced later, in particular in Sections 3.9 and 6. However, a coarse picture may already be helpful here. Services are specified in the app/platform configuration, in particular through their technical information, 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 app configuration. The platform instantiation/code generation either integrates platform-provided services into the realization of such a mesh or, in case of application-specific services, turns this information into service interfaces contained in application code templates to be edited by the user. Moreover, the code generation creates support artifacts such as data classes, data serializers or basic service implementations (for all relevant programming languages, e.g., Java or Python). Further, the instantiation process binds the service (interfaces) with service/glue code to the selected service execution/streaming engine. The binding happens through dynamic class loading. Dependent on the service configuration, data may be handled synchronously or asynchronously. As part of the code generation, also service descriptors required by the Service Control and Management component are created. Further, information for AAS nameplates is taken from the configuration and turned into respective implementation artifacts to build up the related AAS and submodels, e.g., one vendor AAS per service type known to the platform.

The notion of a service is cross-cutting, i.e., it occurs in many topic areas in [ESA+21] and, thus, a summary of all relevant requirements is important for the design and realization. Besides these functional requirements, we must also consider 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 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 [ESA+21]) or dispose (R134c) services on demand. To be integrated in a flexible manner, monitoring and service management must be realized based on explicit interfaces and the oktoflow plugin approach, so that an exchange of the implementations becomes possible. If feasible, existing interfaces shall be utilized.

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.

3.5.2 Service Environments

In oktoflow, 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 processes and receive their control commands and data via inter-process communication/network.

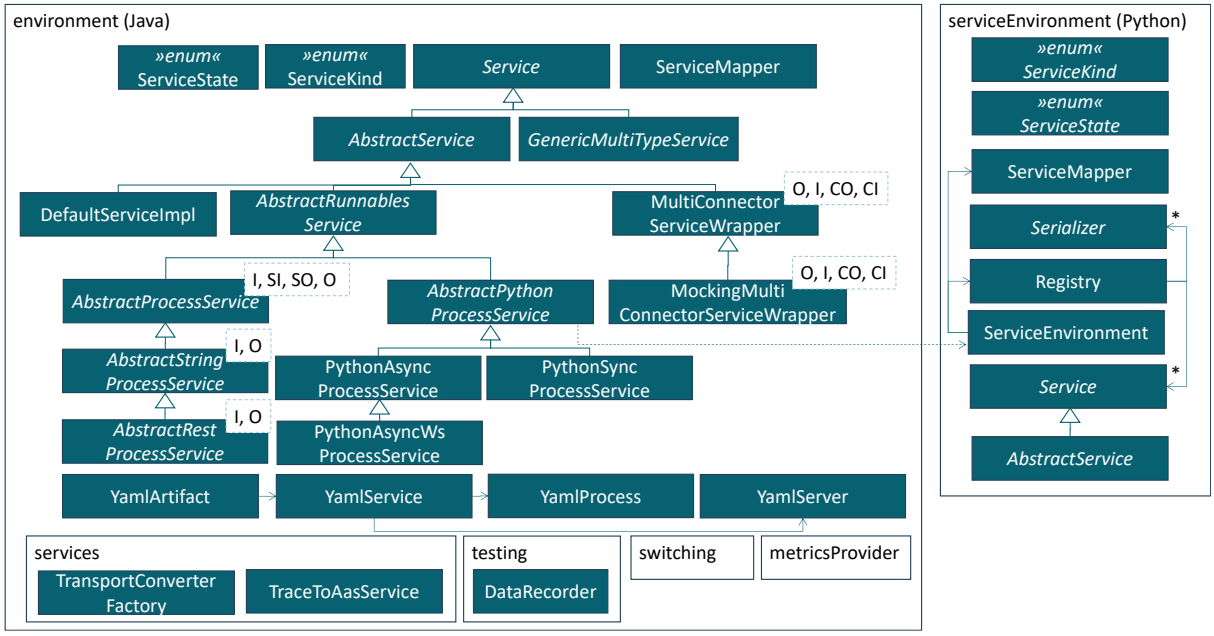

Figure 19: Design of the Service Environments.

The service environments provide the basic types for service implementation. The Java service environment also contains a variety of base classes to ease the integration of native services or services realized in different programming languages such as Python. We will first detail the Java service environment, then the related Python service environment.

3.5.2.1 The Java Service Environment

Figure 19 illustrates the concepts and relations of the service environments, in particular the **Java Service environment**. From the Java point of view, the central package (`environment`) represents both, the basic service environment for Java and Python. This package (on the left side of Figure 19) defines the `Service` interface with all operations discussed for a Service in Section 3.5.2, the `ServiceState` enumeration and the main service kinds (source, transformer, sink) in terms of the `ServiceKind` enumeration. The service environment also provides support for service parameters, i.e., the customization of generic services through values that are determined upon service start or may be changed at runtime.

Several abstract classes provide basic mechanisms for realizing services, e.g., through operating system processes or using the REST protocol (on device-local networks). We just mention some examples here. `AbstractProcessService` provides the abilities to create an operating system process through a standardized naming scheme, to manage the process instance via the service status, including activation and passivation, and to customize the console input/output streams. The `AbstractProcessService` defines four template parameters, namely the received input data type (`I`) from the perspective of the platform/application, the input data type (`SI`) of the implementing process, the output data type (`SO`) of the implementing process and the output data type of the service (`O`) from the perspective of the platform/application. An `AbstractProcessService` requires two `TypeTranslator` instances, so that incoming data can be translated into a format that can be handled by the implementing process and, further, that the output of the implementing process can be re-ingested into the platform data streams. Here, we rely on the `TypeTranslator` interface from the Transport Component (cf. Section 3.4.1). Moreover, the `AbstractProcessService` requires a `ReceptionCallback` (the interface from the Connectors Component in Section 3.4.2), so that data can be processed asynchronously.

As one path of refinements, the `AbstractStringProcessService` forms the basis for process-based services that communicate through a string format via console streams with the implementing process, e.g., JSON. This fixes two template parameters, namely `SI` and `SO` to `String`, also allowing to provide a basic realization of some inherited abstract methods, e.g., how to receive data from the implementing process. The `AbstractRestProcessService` is a further refinement which expects REST based communication represented as `String`.

Another path of refinements targets Python services, i.e., specialize the `AbstractProcessService` for executing Python and the Python Service Environment. This includes synchronous processing via command line streams (`PythonSyncProcessService`), asynchronous processing via command line streams (`PythonAsyncProcessService`) and asynchronous processing via websockets as alternative forms of integration.

Further, the `DefaultServiceImpl` is a base class that implements all methods not implemented so far empty. As "adapter" classes in Java, this class shall ease defining own services without the need of providing empty methods that are technically required but not seem to be needed by the service.

All service basis classes discussed so far, partially like connectors due to type parameters, handle exactly one input and one output type. All "Multi" classes wrap such instances in a way that the resulting service can handle arbitrary types. For services, this is the `GenericMultiTypeService`, for Connectors, which can be wrapped into a service in uniform manner, the MultiConnectorServiceWrapper, and for testing apps, the `MockingMultiConnectorServiceWrapper`, which injects testing data provided in resource files.

Besides the basic classes, the Java service environment offers the following:

- The `Starter` registers all services given in a YAML service descriptor and starts the application AAS command server on a given port.
- The `ServiceMapper`, a helper class that binds a service against a given AAS command server, usually the one created by the `Starter`. Further, the `ServiceMapper` registers the available metrics (see below) in the AAS command server.
- Generic `services`: One generic service that is provided by the Java Service environment is the `TraceToAasService`. This (sink) service provides/contributes to an AAS used as application endpoint. The main purpose is to provide access to trace messages explicitly emitted by the services of an application as well as to act as a uniform frame for operations that steer or reconfigure the respective application, e.g., through backward data flows. However, although potentially desirable, the service currently does not turn received data rather than optional trace messages sent by the individual services in an application. Trace messages can be enabled for debugging or for demonstration. These messages carry their origin, the action as well as an action-specific payload, e.g., the received data. The service collects all trace messages and displays them for a given time frame in its trace submodel[55] or in a channel of the websocket status server of the platform so that they can easily be taken up by the management UI. As some data tends to be rather large, e.g., the drive oscilloscope or the magnetic identification data of the EMO'23 demonstrator, the underlying `TransportConverter` can be set up to filter or modify/clean certain payload types. Further, the `TraceToAasService` is intended to act as a hybrid service, i.e., it can be used as basis to implement an application-specific service, which then may display processed data in an application manner or which may provide operations to be called by a device connected to the endpoint AAS. The `TraceToAasService` (as well as mocking connectors) include a generic

[55] ´Not recommended for BaSyx1, as the insertion and cleanup changes to the submodel cause cause memory overflows and increasing CPU load.

`DataRecorder`, which is only active in application testing mode (`--Diip.test=true`). The recorder stores all received data in terms of JSON in order to ease the realization of concrete services, e.g., to pass mocked data sent out or received at a certain service as example input for data analysts or AI developers.

- Runtime service switching (`switching`): Additional mechanisms and base classes to enable a runtime switch-over between compatible, alternative services. The application configuration represents the alternative services in terms of a service family.
- Runtime monitoring (`metricsProvider`): Customized mechanisms to ease applying oktoflows metrics plugin to services based on the work of Miguel Gómez Casado [Cas21, CE21]. All information to be monitored is represented in terms of gauges, counters or timers as defined in oktoflow's monitoring plugin interface. This customization also allows to represent and query distributed meters in a uniform manner, in particular to map them into AAS (R7, R14). The `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. Services can be explicitly marked as `MonitoredService` to receive an instance of the metrics provider in order to define and measure application-specific metrics. For example, the `AbstractProcessService` discussed above is a `MonitoredService` to provide access to the runtime metrices of the underlying process through oktoflow's process monitoring plugin interface. Moreover, services can be marked as `UpdatingMonitoredService` if regular updates of the measurements are needed.
- Test support (`testing`): Additional support classes for testing services in their environment, in particular the `DataRecorder`.

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 default Service Management and Control component relies on service deployment descriptors, one obvious approach is to represent the relevant information in terms of that (extended) descriptor. As these descriptors are given in YAML format, the classes `YamlArtifact`, `YamlService`, `YamlProcess` and `YamlServer` are part of the service environment to represent that information. It is important to recall that we need here only a part of the potential information in the deployment descriptor, e.g., the technical information on how to transfer network ports or how to start a Python process is not required. Further, these classes can be used as a basis to realize the parsing of the deployment descriptor of the service management and control operations in Section 3.5.2. For this purpose, parts of the (Java) service environment are imported into the Service Management and Control component and used there.

Besides the generic Java service environment, there is also a one for the current default stream processing approach (Spring Cloud Stream), the `environment.spring` in Figure 19. To provide a better integration with the logging of a Spring environment and also to reuse the already customized metering probes, the Spring service environment integrates the default logging and meter plugin implementations of oktoflow as direct dependencies and replaces the underlying libraries with the libraries that are used in the actual Spring version (also running the plugin test suites to ensure functionality and compliance). Further, the Spring service environment handles Spring-specific integration topics, e.g., the startup code in the refined `Starter` class hooks into the Spring startup process, i.e., it re-configures the Spring Rest server port and attaches that port to the `MetricsExtractorRestClient` used by the upcoming services. Further, this `Starter` version fires up the AAS command server of the parent class at a point in time when this is permissible for Spring. The `Starter` class is then executed by a specialized Spring loader (cf. Section 3.5.3).

3.5.2.2 The Python Service Environment

So far, we exclusively discussed the Java side of the service environment. Except for the generic service classes, the helper/support classes, the monitoring and the Spring-specific implementation, the **Python service environment** is a mirror of the Java service environment. Differences are:

- The Python environment is accessed through the Java representation of Python services in the streaming engine, i.e., the Python environment realizes some form of command server as well as the intra-device soft-realtime data transport, possibly with a fixed (local) wire format.
- We apply a reduced monitoring to the non-Java service environments, because stream measures can be taken on the Java side. In contrast, resource measures such as execution time or memory consumption can be combined with the related Java process, i.e., the non-Java service environment delives the information for the own process, which is then combined to a unified measurement on the Java side.

As illustrated at the bottom of Figure 19, the Python service environment implements similar service concepts as the Java environment. However, the Python service environment does not need to be a complete mirror as only the parts to execute services and to enable the communication between the Java side and the Python service implementation are required. Therefore, the Python service environment provides the `ServiceEnvironment` class, which imports service implementations dynamically from four modules named `datatypes`, `serializers`, `interfaces` and `services`. The first three modules are generated from the configuration model and provide datatype implementations, related serializer/type translator implementations and service interfaces specified in the configuration model (implemented based on `Service`/`AbstractService` from the Python service environment). The `services` module contains the (manual) implementations of the services.

For the data transport between the Java side and a service environment in Python, we can imagine the following alternatives:

1. Use of the platform Transport Component using a local transport server/broker. However, as the Transport Component is open regarding the transport protocol and the serialization, this would imply that all service environments (except for the Java service environment) must implement all transport protocol variants and all serialization mechanisms. If there is not just “the Python environment” but further language-specific environments, this leads to a plethora of required protocols. In particular, if the user organization decides to implement own protocols, these protocols must be mirrored into each environment. We do not think that this is a feasible solution and opted for restricting the transport layer to Java code. Thus, the communication with Python may be based on a (local) protocol here, e.g., HTTP/REST as well as the serialization mechanism may even be fixed (we decided for JSON). While this approach is feasible as only a few variants are needed, it requires server processes for the communication on Java and Python side.
2. Extend the AAS communication protocol to transfer data (in BaSyx1 VAB, in BaSyx2 HTTP/REST). This is a specific decision of fixing protocol and serialization as mentioned in alternative 1. Also here, the backwards channel would require further server processes on the Java side as well as compliance with potentially changing BaSyx protocols. However, extending an external protocol also imposes compatibility and sustainability risks if decisions for the underlying implementation are made, that conflict with our decisions.
3. As the non-Java service implementations are executed as local processes, also command line input- and output streams provided by the operating system may be a low-risk option (as some service candidates and many Unix command line programs do). Here, in particular the Java side must carefully parse the output of the executed services/service environment not to

confuse “normal” or logging output with data output. However, command line streams are said to be a performance issue on Windows-based systems.

4. Apply a local inter-process communication approach as alternative to the command line integration suggested in the last alternative. An obvious option that also shares synergies with other platform-provided service integrations is REST[56] (representational state transfer). However, REST is designed for synchronous communication and does not allow for asynchronous service execution. While this may be adequate for service implementations that offer a REST API, it is an unnecessary limitation for the Python service environment. As a full-duplex enabled alternative that allows for synchronous and asynchronous service execution, we apply WebSockets[57] for local communication between Java and Python. Anoter alternative that could be integrated similarly is some form of RPC[58] (Remote Procedure Call), e.g., gRPC[59] with Protobuf.

Currently, the Python `ServiceEnvironment` implements both, the third (command line streams) and, as alternative, the fourth alternative (WebSockets) using generated object-to-string serializers with JSON as default wire format. We also use the command line streams for the command protocol. On the Java side, specific classes are bound against the Python service environment and the service deployment descriptor specifies the required service-specific Python artifacts as well as the Python command line parameters. More specifically, as already indicated above, `PythonAsyncProcessService` (command line streams) and `PythonAsyncWasProcessService` (WebSockets) are responsible for continuously running the Python `ServiceEnvironment` and the `PythonSyncProcessService` is an experimental call-and-return implementation of a Python service integration. While the two continuous classes communicate data and commands with the `ServiceEnvironment`, the `PythonSyncProcessService` transfers only data items and calls Python upon each data item. Besides the different communication styles, all Python integration classes support synchronous and asynchronous services and their call styles.

As Python is a system-level program, it is relevant to understand how the platform determines which Python version/installation to use. This is of particular interest, as operating systems tend to provide a more recent version, but sometimes the most recent version may not be compatible with your requirements. Moreover, certain installations may have multiple Python versions installed, so it is important to provide a mechanism, which Python to execute actually per service.

- For a (containerized) application, the platform and the platform supplied (generic) services take the information on installed dependencies into account (cf. Section 3.3.3.1).
- For direct execution of Python, e.g., in tests or in the build process, the platform (and the build processes) considers the system environment variable `IIP_PYTHON` as location of the python binary. If `IIP_PYTHON` is not set, the platform tries to utilize the first Python in the system path, as fallbacks `/usr/bin/python3` or, ultimately, `python`.

3.5.2.3 Validation

The service environment is subject to automated regression and integration testing. In particular the monitoring classes are tested extensively [Cas21]. 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

[56] https://de.wikipedia.org/wiki/Representational_State_Transfer
[57] https://de.wikipedia.org/wiki/WebSocket
[58] https://de.wikipedia.org/wiki/Remote_Procedure_Call
[59] https://grpc.io/

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 3.4.2.2 could be used as baseline.

Besides service-level tests, performance experiments for BaSyx1 have been performed in [Cas21]. Retrieving a meter via an AAS on a Lenovo Z50-70 laptop requires 4-5 ms after a settling period 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 more extensive performance experiment is presented in [CE21], showing that an integration of the transport layer with a monitoring values cache attached to a remote AAS can be as fast as a local AAS on the monitored device. In other words, the AAS that is updated in parallel is not impacting the data paths to other components, which are informed via the transport layer (i.e., pub-sub with fixed JSON wire format). However, AAS properties realized through Java functions are only (partially) supported in BaSyx1; for BaSyx2 we currently write the metrics values directly into the AAS at higher overhead. In particular, the pub-sub approach decouples the startup of the AAS implementation server as only the platform transport broker/server is used, which is started as part of the platform.

3.5.3 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 service 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.

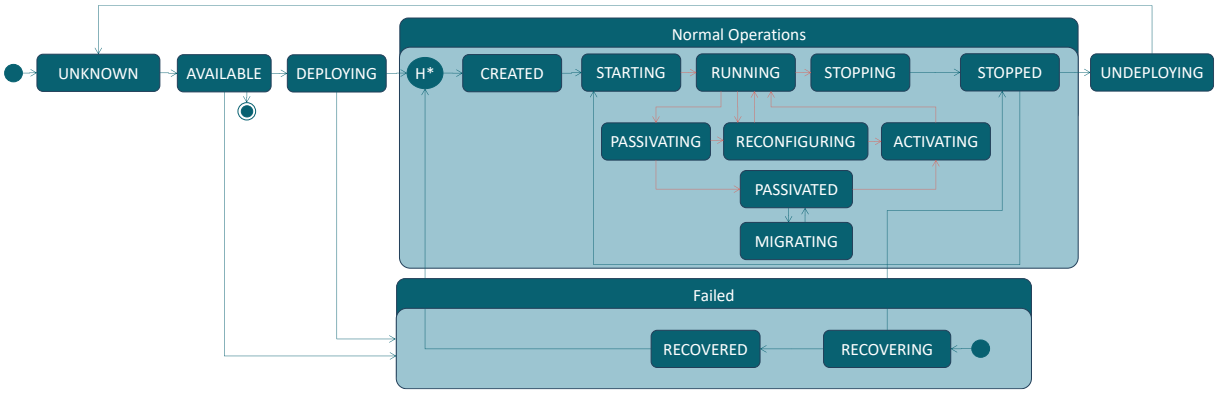

Figure 20: Service states.

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 20. Services (in their artifacts) can be downloaded and transitions via `UNKOWN` to `AVAILABLE` on the hosting resource. When triggered

through 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 started again or removed from the resource (UNDEPLOYING, afterwards UNKNOWN and potentially again AVAILABLE). 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 20). If the service becomes operational again, it continues in the upper sub-state machine and there into the last "normal operation" state (via the UML H* deep history state).

3.5.4 Design and (Plugin-)Interfaces

Figure 21 illustrates the design of the Service Control and Management component (services as plugin interface for concrete implementations). At the core of this layer is the ServiceManager, which performs operations such as starting and stopping individual services. While the transitions displayed in dark green in Figure 20 are controlled by the ServiceManager, the transitions in red are performed by the service and monitored (via AAS property value polling) by the ServiceManager.

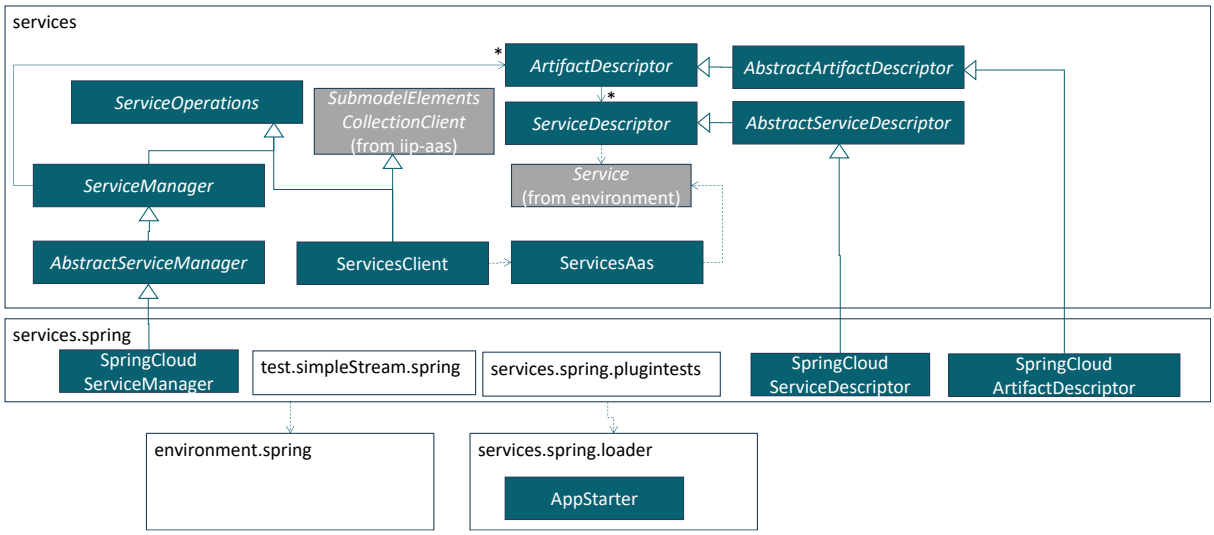

Figure 21: Service interfaces and management

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, whereby several operations and information accesses are delegated by the service manager to the descriptors or through the respective AAS to the (device-local distributed) application AAS implementation server directly approaching the service instance.

Typically**, services can operate as a single process** and do not require further resources, e.g., a central server process to communicate with. However, there are services with this requirement, e.g., the anonymizer and pseudonymizer KODEX (cf. Section 3.7.1), which optionally may cooperate with a server instance, or a federated learning approach, which usually requires a central model management and exchange server. There are settings, where such server processes may be installed and controlled by the user, but we support also application-specific server processes, which are integrated into the lifecycle of an application, and, thus, are started and stopped along with the application by the platform. Moreover, certain applications may require some form of interaction with the server, e.g., to re-configure KODEX at runtime or to request the replay of a model snapshot of a federated learning server. It is desirable that such interactions happen in a uniform manner so that application components can rely on and reuse the approach. For the platform, it would, thus, be desirable to

represent such server processes within the platform AAS and to allow, e.g., for runtime re-configuration of service parameters (which may also trigger a model exchange). Furthermore, it would be desirable that the resource which hosts the server process can, as for services, be decided freely within the available resources of a platform instance. Moreover, a uniform, standard-compliant communication between client and server would be desirable, most preferably via the Transport Component (cf. Section 3.4.1) of the platform to also exploit the flexible exchange of protocols. However, directing an externally determined communication via the Transport Component may require changes to the server/service to be integrated, which may not be possible in all cases, for which we prioritize this requirement as optional, i.e., to be decided based on the server/service at hands.

While we believe that in most cases the open service concept of oktoflow is also applicable to services in a server/service setup, the server side requires specialized capabilities. In general, we consider a server as a realization of an (internal) service, which does not exchange data via the service input/output interfaces. This allows for utilizing AAS publishing, distribution, re-configuration and execution mechanisms of usual services, in particular the service environments for Java and Python, while not requiring an input/output modeling for the internal service/server communication in the configuration model (cf. Section 6). For this purpose, some server-specific capabilities had to be added to the service environments, e.g., that services know the (optional) service/server communication channel they are relying on. It further requires specialized capabilities such as assigning server processes to an application in the configuration model, registering and announcing the actual server network information via the platform network management (cf. Section 3.3.4.2), starting/stopping server processes along with applications (while maintaining the number of service instances using the ports in the platform network management) or provisioning of specialized transport streams for (private) service/server communication. The required capabilities are partially realized in the service manager and partially in the service environment as generic or specific functionality of basic services.

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 3.6.1 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 4 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.

The `ServicesAasClient` provides access to the properties and operations of the AAS of the service layer. Actually, both implement the same interface called `ServiceOperations`, which defines 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.

Applications consisting of service meshes can conveniently be managed through **deployment plans** as opposed to manually starting and stopping individual services. One application can have multiple deployment plans, i.e., different deployments. Moreover, a deployment plan may be enabled to be started multiple times. Upon each start, a new application instance with individual data paths/streams and individual service instances is created. During startup, a deployment plan can directly re-defined service parameters so that application instances may, e.g., target different devices. We utilized this capability in one of our public demonstrators, where the same application in two distinct instances with different service parameter settings was used to execute a federated learning setting between two individual cobots. Starting multiple instances and re-configuring them at startup prevents specifying the same application multiple times in the configuration, which may easily lead to inconsistencies. Similar to startup, application instances can individually be shut down using their application instance id (more conveniently on the platform management user interface where you just select the application instance to be stopped instead of recording the instance id).

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 [ESA+21] 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 3.5.2). 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.

3.5.5 Spring-based Service Control and Management

Different technologies can be used to realize and execute service meshes, 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 3.4.1, 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 3.4.1. As also stated there, we foresee that the 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 21 as interfaces (in the package `services`), while the actual implementation is realized as an oktoflow plugin.

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 (as discussed below). The Spring-based `ServiceManager` refines the descriptor classes so that the information can be loaded directly from the Spring application description. Further, it 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 some cases so that the deployment descriptor allows for specifying “ensemble” services, i.e., Spring-services that must be executed within the same process.

Figure 22 illustrates the structure of a generated application artifact for execution with Spring Cloud Stream. Here, an artifact consists of combined binaries provided by the service execution framework (the startup code) and binaries that make up the application (below `BOOT-INF`, requiring a specialized `ResourceResolver`, cf. Section 3.3.2.2). Within the application parts, such an artifact contains all application dependencies (in `lib`) including, e.g., the service manager of the platform and its transitive dependencies, but also generated parts such as the application interfaces (folder `iip`, also containing the startup class for the Java service environment `Starter.class`), the Spring Cloud Stream application specification (`application.yml`), the service deployment descriptor (`deployment.yml`) and the logging configuration (`logback.xml`). Depending on the integrated services, more artifacts may be included, e.g., reusable service binaries (here `kodex.zip`), customized service artifacts (`kodex_pseudonymizer.zip`) or the Python code for executing a Python service in the Python service environment (`python_kodexPythonService.zip`). Such specific binaries are referenced in the deployment descriptor and unpacked by the service manager upon start. However, this structure does not serve for isolated loading using oktoflow plugins. For this purpose, the contents of `classes`, `libraries` and the `classpath.idx` file (serving for the root classloader then) is separated into application-related files for isolated loading, named with suffix -app, i.e., `lib-app`, `classes-app` and `classpath-app.idx`. As the original Spring loader shown in Figure 22 is not able to cope with this separation, we realized the component `services.spring.loader`, which can handle both isolated and non-isolated app loading depending on the actual structure of the artifact replaces the Spring loader. Unfortunately, the base classes of Spring are not designed for reusability so that several classes have to be duplicated.

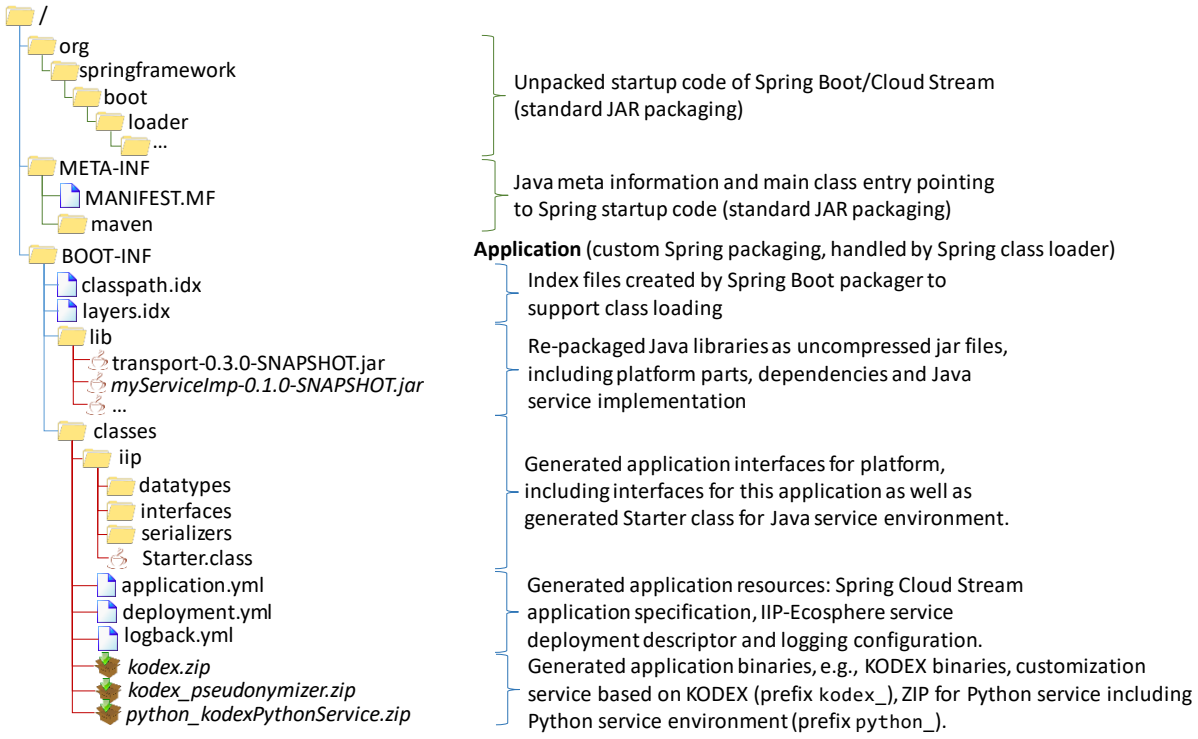

Figure 22: Structure of a JAR application artefact for the Spring Cloud Stream engine (non-isolated loading).

The packaging shown in Figure 22 represents an executable Java ARchive (JAR) and may even be executed without the platform (provided that a respective setup, e.g., communication ports are given). In principle, this is also one major functionality of the service management - besides passing environment settings such as (dynamic) ports to the services stored in the service artifact. However, as an executable JAR, the packaged Spring application prevents shared libraries, i.e., besides plugins, libraries that must not be packaged into the artifact to reduce the footprint of the artifact. To allow for shared artifacts, the service management supports a secondary format, which is independent of the Spring packaging approach (although based on `services.spring.loader`). This packaging structure is shown in Figure 23.

In contrast to Figure 22, the ZIP-based artifact is not an executable JAR and contains packaged rather than unpacked JARs. The application JAR including the generated class `iip.Starter` must be in the top-level directory of the ZIP. There must also be the service deployment descriptor (`deployment.yml`) so that the service manager can read the information about the contained services. Moreover, also the "binary" artifacts, e.g., for KODEX or python must be on top-level, so that the service manager can extract the artifacts for executing them in terms of operating system processes. In contrast, the Spring application definition (`application.yml`) and the logging configuration shall reside in the generated application artifacts as they will be loaded by the respective Java libraries on demand via class loading. The dependencies of the application are located in the `jars` folder (or optionally on top-level). Here the difference to Figure 22 is that any shared jar can easily be removed from that folder (during the packaging process or manually for experiments) and provided through a shared libraries folder[60] known to the service manager. The ZIP shall may contain in the file `classpath` a listing[61] of Java libraries in their intended class loading sequence. For isolated loading, we also employ the file `classpath-app`, i.e., as for Spring classes and libraries for the root classloader are named in `classpath` and for isolated app class loading in `classpath-app`.

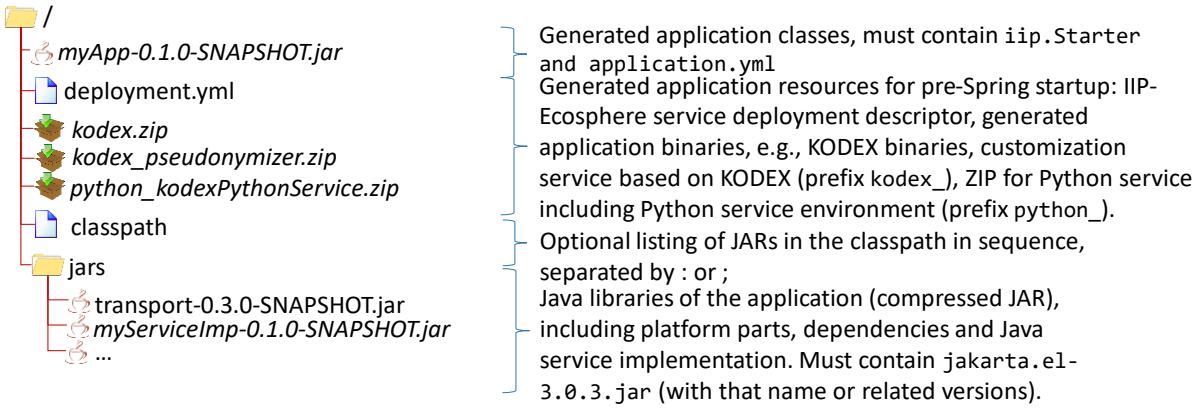

Figure 23: Structure of a ZIP application artefact allowing for shared libraries (as variant of Figure 22).

3.5.6 Validation

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

[60] Currently, this folder is specified in the setup information of the service manager. This information could be relocated into the service deployment descriptor in future versions of the platform.

[61] Relative file names in Windows or Linux notation, separated by : or ; depending on the operating system. Such a file can be created by Maven. Before execution, the file is rewritten to comply with the @ argument file format of Java 9. Thereby, the JARs in the root folder of the archive are added by the service manager to the start of the classpath. If no such file is present, a wildcard classpath is constructed, which may cause accidental class loading conflicts.

handcrafted service artifact with simple contained services and multiple deployment descriptor targeting different artifacts, e.g., with or without process ensembles. This artifact (`test.simpleStream.spring`) is based on the (Spring) Java service environment (cf. Section 3.5.2). However, just running the tests may not be possible as in particular at the service control and execution varying, potentially conflicting dependencies may be needed, e.g., the Spring service execution together with the Spring-based BaSyx2. In other words, we must set up a dependency-free environment for the tests (`services.spring.plugintests`), load the spring-based service execution as one of multiple plugins and, through the service manager interfaces, execute the tests that are defined in `services.spring`, which, in turn, stat the processes of the test application (`test.simpleStream.spring`). 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 3.5.2) 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.

Furthermore, the Spring Cloud based Service Manager was validated in a Linux VM-based server setting as well as on a Phoenix Contact AXC 3152[62] PLC-Edge with 2 GByte RAM and 8 GByte memory card providing additional hard disk space. On the Linux server, the Service Manager was executed directly on the operating system as well as in a Docker Container, on the AXC we focused only on the Docker setup. In both cases, we were able to manage a simple demonstration application (adding the artifact, starting, stopping, removing the artifact) and to verify that the expected input/output behavior of the services can be observed. As starting and stopping individual services involves powering up a JVM, the service manager takes a certain operation timeout (with a default length of 30 seconds) into account. This is sufficient for the Linux server (and the regression tests mentioned above), but does not work on the AXC 3152, where a longer timeout is needed.

3.6 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 3.6.1, the device management in Section 3.6.2 and the monitoring in Section 3.6.3.

3.6.1 ECS runtime

Flexible and heterogeneous deployment to edge, server and cloud devices is a central capability of the platform. [ESA+21] defines several requirements for the envisioned deployment approach. R25c and R25d target the (central) management of resources and, thus, are addressed by the device management in Section 3.6.2.

As described in [SSE21], 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.

[62] https://www.phoenixcontact.com/online/portal/de?uri=pxc-oc-itemdetail:pid=1069208&library=dede&tab=1 provided by Phoenix Contact to the SSE group of the University of Hildesheim https://sse.uni-hildesheim.de/aktuelles/detailansicht/weltweiter-marktfuehrer-unterstuetzt-universitaet-hildesheim-im-bereich-industrie-40/

Figure 24 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 3.5.2 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.

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 oktoflow 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 3.6.2).

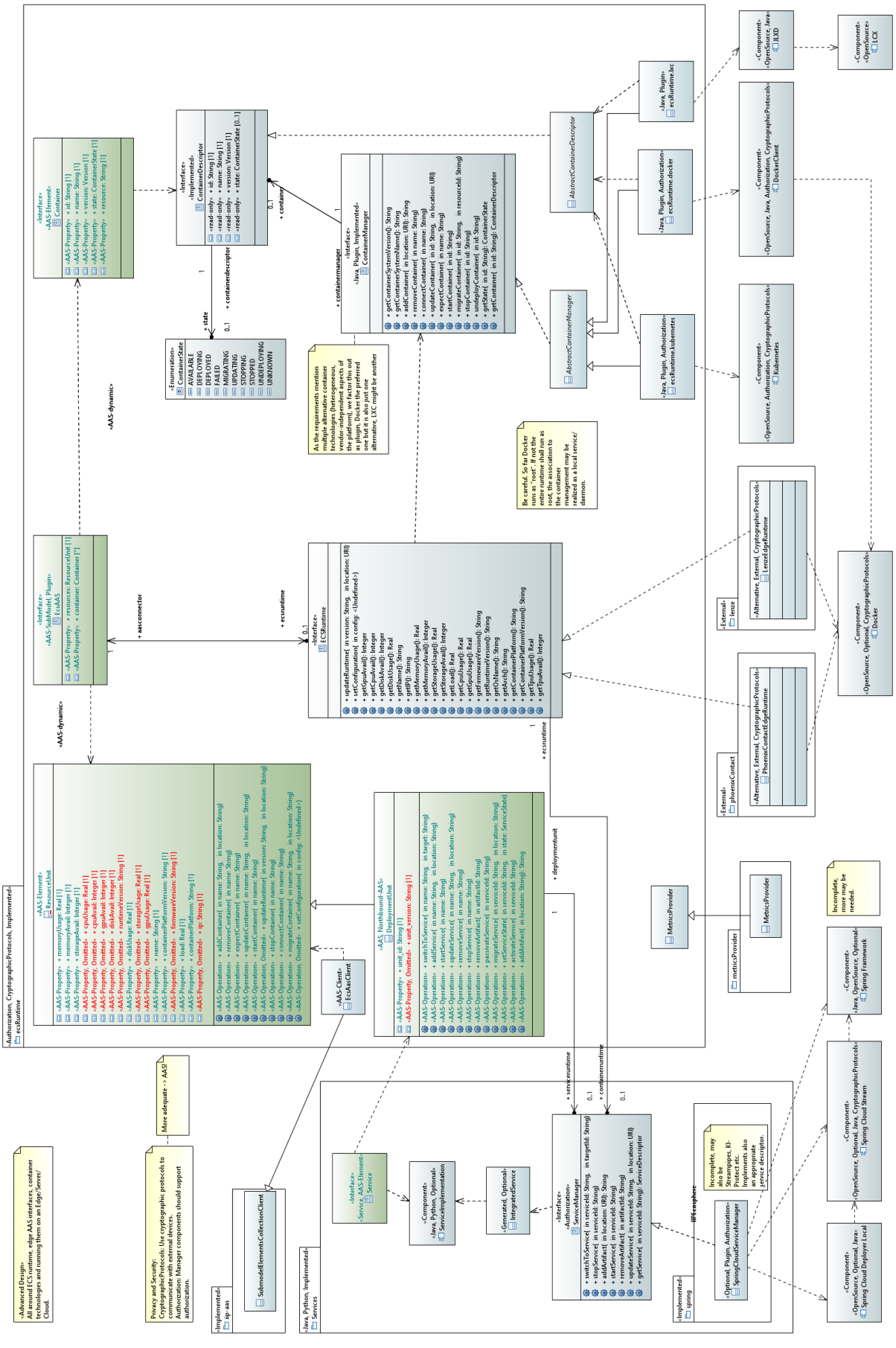

Figure 24: ECS runtime for Service Deployment (comments partially cropped)

As already emphasized in [SSE21, ESA+21], one fundamental basic work for the resource abstraction runtime is the LNI 4.0 edge configuration usage view [LNI40]. [SSE21] subsumes and extends [LNI40] and [ESA+21] integrates relevant requirements from [SSE21]. As the need for managing resources and containers on resources, in particular edge devices, is known in Industry 4.0, platforms [SEA+20] and also other work address this topic in various ways. In addition to the 21 platforms analyzed in [SEA+20], also approaches like OpenHorizon[63], the IBM Edge Application Manager[64] or the ICP4Life platform [MBB+18] have been researched or are available. In recent time, also container orchestrators such as Kubernetes[65] became popular. Although there are significant overlaps, there are also important differences between these approaches and the ECS runtime. One main difference is the general requirement R7 that all interfaces in the platform shall be based on AAS aiming at an interoperable integration of heterogeneous devices (based on an agreed structure, at least within the platform). Moreover, it is important to point out that we aim 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 [SSE21], we aim at supporting more sophisticated management operations on the edge, in particular for data paths and relations as discussed in Section 3.5. Also platforms have been created as part of research projects, e.g., ICP4Life [MBB+18]. However, not all of these platforms are publicly available and suffer from similar overlaps and differences, e.g., the strong focus of oktoflow on AAS and AI.

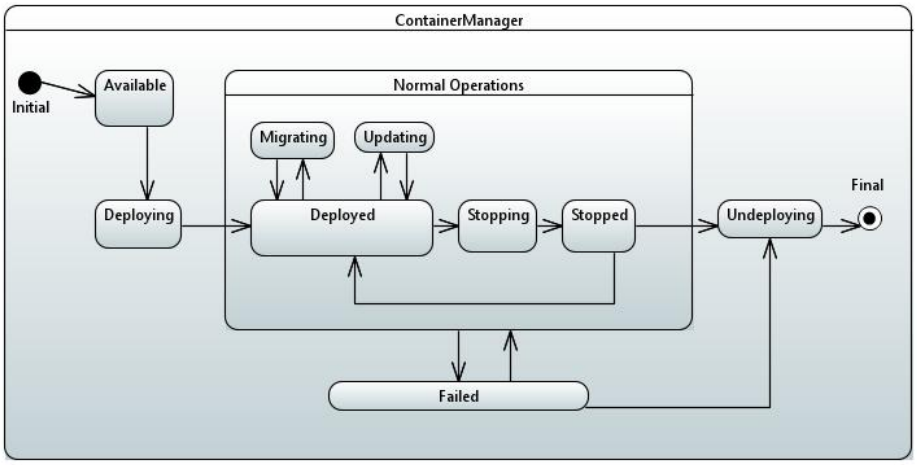

Figure 25: Container states

Figure 24 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[Error! Bookmark not defined.] for the ECS r untime 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

[63] https://www.lfedge.org/projects/openhorizon/
[64] https://www.ibm.com/docs/en/edge-computing/4.1
[65] https://kubernetes.io/de/

Figure 24, but also a `GenericJavaRuntime`, which relies on an abstract `ContainerManager` (along with a `ContainerDescriptor`, akin to the service descriptors in Figure 21). Akin to the service manager, the container manager provides AAS operations, e.g., to download, start or stop a container. The respective container states are depicted in Figure 25.

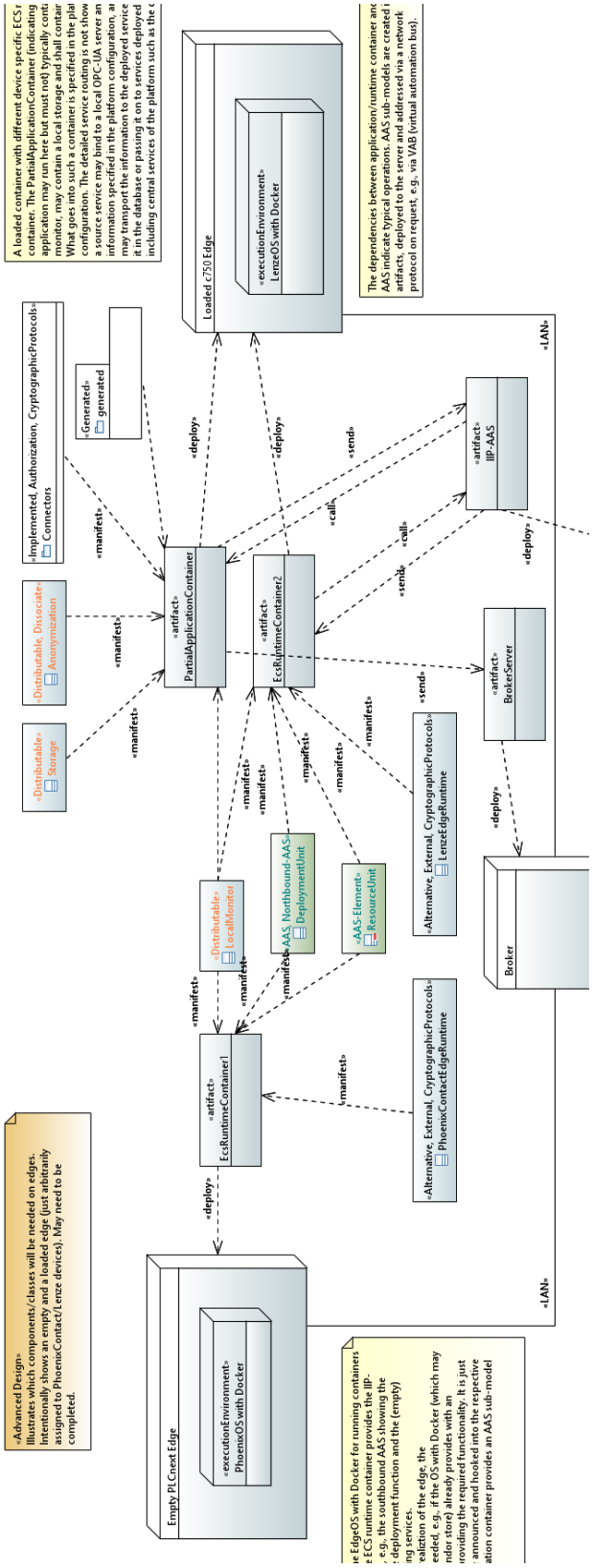

Figure 26: Example deployments to empty (left) and loaded (right) edge device (comments and further deployment nodes representing the platform partially cropped)

Partially, device-specific functionality is supposed to be realized via other plugins, e.g., target-system specific implementation of the `SystemMetrics` (cf. Section 3.5.4) or how to provide and access to the device AAS. It may be that the device is already older and does not provide an AAS. For this purpose, the ECS runtime allows to customize the AAS origin via the

`DeviceAasProviderDescriptor`, which determines the component that returns the address of the respective device AAS (the component may also create the AAS if needed). Currently, three implementations of the related `DeviceAasProvider` are shipped with the platform, a pragmatic one reading the AAS from a simple Yaml file (reading manufacturer/product images from the same location), an implementation reading the AAS from an AASX package file and a multi-provider that selects the first provider (by default Yaml or AASX) that returns an AAS address. For the Yaml/AASX providers, the underlying information is retrieved as classpath resource, either as `nameplate.yml`/*deviceId*`.yml` or `device.aasx`/*deviceId*`.aasx`, respectively whereby *deviceId* is taken from the `IdProvider` of the support layer. This is currently the most easy and compliant way to specify device-supplied services and their properties, e.g., port or protocol version.

The platform provides a plain Docker[66] 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. In addition, the platform offers an LXC container manager. We selected LXC [Sch23] due to the use in other industrial platforms [SEA+20] and as Docker changes its (commercial) licensing. It is important to mention that LXC is licensed under GPL and, thus, must be an optional component of the platform due to platform licensing rules. LXC is integrated via the Java LXD library JLXD[67] and requires specific installation steps, which are detailed in [Sch23]. In the current state, it is not supporting all container creation strategies and may be limited regarding a container registry. 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 the figures, we did not indicate the relation 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, the diagrams do not indicate much monitoring support for the ECS runtime except for some AAS properties in ResourceUnit. As the Java service environment (cf. Section 3.5.2) 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. 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 26 illustrates two potential deployments to aforementioned example devices (including AAS server components, deployment interactions, a broker server and stereotypes from the UMLsec/security profile).

[66] https://www.docker.com/

[67] The official repository is at https://github.com/digitalspider/jlxd. However, due to required bugfixes and the need for a deployment to Maven central, which was not provided by the original authors, we rely on a fork of JLXD, which is part of EASy-Producer https://github.com/SSEHUB/EASyProducer.

The AAS of this component is represented by `EcsAAS`, actually the resources sub-model already mentioned in Section 3.5. 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 3.5.

As for the generic platform 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.

Experiments with containers and AAS indicate 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 3.5.2). 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 a non-virtualized operating system was not necessarily better in this regard. However, this experience strongly depends on the AAS and protocol implementation and, thus, is not representative.

We also validated starting Docker containers via the ECS runtime and the container manager, running the ECS runtime directly on the underlying operating system as well as running the ECS runtime in a Docker container. For the latter, a Docker-out-of-Docker (DooD)[68] setup is required. Moreover, to achieve a certain genericity of the ECS runtime container, it is advisable to mount the containers via a folder of the host operating system into the ECS runtime container. The functions of the container management were validated on a Linux virtual machine running on a VMWare ESXi server as well as on the Phoenix Contact ACX 3152 mentioned in Section 3.5. As the container operations require a certain execution time, the minimum overhead created by an AAS-based management operation is not relevant here. The service capabilities are validated in several examples, most under regression testing, as well as some public demonstrators (cf. Section 6.7). The service/server functionality is currently subject to regression testing as well as the preparation of upcoming public demonstrators.

It is important to mention, that the sizes of the Docker container depend on the platform and application services that are installed. An ECS runtime with a DooD setup requires a container of around 1.1 GByte size (packed image of 444 MBytes), a service manager demands 509 MBytes (336 MBytes packed image) and a combined installation of ECS runtime and service manager into one container 600 MBytes (286 MBytes packed image). All containers can be installed and executed successfully even on an AXC 3152, typically with the platform server and the central broker installed on a server, e.g., the Linux virtual machine mentioned above. The running containers in idle mode allocate roughly 200 MBytes main memory (1.4 GBytes remain free on the AXC 3152), although at least 3 JVMs (ECS runtime, Service Manager and a local broker for the services) are running. If a simple service chain with 2 services is started, further 400 MBytes are allocated by one JVM per service, i.e., roughly 800 MBytes to 1 GByte memory remains free on the AXC 3152. Here, dependent on the actual load and service demands, we allow for some optimizations, e.g., to combine services with process backends, e.g., Python into the same JVM (ensemble services) or to limit the maximum memory allocation of the involved JVMs adequately. For the latter, the platform configuration model allows settings for the platform services as well as for individual application services, which are turned into executable artifacts by code generation (cf. Section 6).

[68] http://tdongsi.github.io/blog/2017/04/23/docker-out-of-docker/

3.6.2 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 3.6.1, the notion of devices in oktoflow 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 [SSE21]) and/or a Service Manager. Also, different forms of installation for an ECS runtime as discussed in Section 3.6.1 are subject to the device management. It is important to recall that following [SSE21], Industry 4.0 field devices such as machines are out of scope for the platform.

From [ESA+21] 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 [SSE21]. This includes the abstraction of vendor dependencies (R25.a), on/offboarding (R25a) and device management (R25b). Common management functions which are neither listed in [ESA+21] nor [SSE21], e.g., mechanisms for human interactions (acknowledgements), management techniques such as device templates or import functions for "asset data providers" [SSE21] are desirable, but also well covered by existing platforms [SEA+20]. Thus, in [ESA+21, SSE21] it was intentionally left open, whether the platform just focuses on the essential capabilities mentioned in [ESA+21, SSE21] or provides also additional useful capabilities. Please note that quality requirements regarding data processing time limits, e.g., soft realtime, do not apply to management operations of the device management.

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 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 3.6.1. 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 3.6.1). The functionality of the device management is influenced by given information (through AAS events[69] 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 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 platform. This allows taking other 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[70].

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.

[69] 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 3.3. Thus, the component design shall foresee event-based change notifications as well as (potentially less efficient) polling/scanning of the respective AAS structures.

[70] Currently, security and access restriction mechanisms are not (fully) in place in BaSyx and, thus, not reflected in the AAS abstraction introduced in Section 3.3.

Regarding security (R38-R44) or data privacy (R45-R68), this may include the exchange/installation of encryption keys or certificates during onboarding.

For the device management, further existing components could be helpful, e.g., an IoT device management approach (to be frontended by AAS), a secure console (R37) or a storage for binary images (R36a, R36b, R136a). A discussion of potential components in the scope of the requirements for the resource management is provided by Pidun in [Pid21]. There, for the IoT device management, two components are discussed, namely DeviceHive[71] and ThingsBoard[72]. The specific capabilities of ThingsBoard, such as software-over-the-air, firmware-over-the-air, the broad range of supported protocols, an Angular user interface that could help realizing an optional user interface for the platform as well as a higher development activity in the recent time led to the suggestion of integrating ThingsBoard as one alternative technology into the platform. For the binary storage, MinIO[73] and OpenStack Object Store Swift[74] were compared. Here the support for the de facto standard S3 made the difference and MinIO was suggested in [Pid21].

The architecture of this component follows the architectural suggestions in [Pid21]. An overview is depicted in Figure 27. The component offers two AAS interfaces, a southbound interface in `DeviceRegistryAas`, and a northbound interface in `DeviceManagementAas`. The southbound interface is intended to enable a self-registration of devices and to notify the platform that they are available (heartbeat). This involves so-called `ManagedDevice` instances, which bridge between the `ResourceUnit` from Section 3.6.1 and specific information required by the underlying management approach, e.g., a different secondary device id. The northbound interface provides device information to higher-level components in the platform.

[71] https://github.com/devicehive
[72] https://github.com/thingsboard/thingsboard
[73] https://github.com/minio/minio
[74] https://github.com/openstack/swift

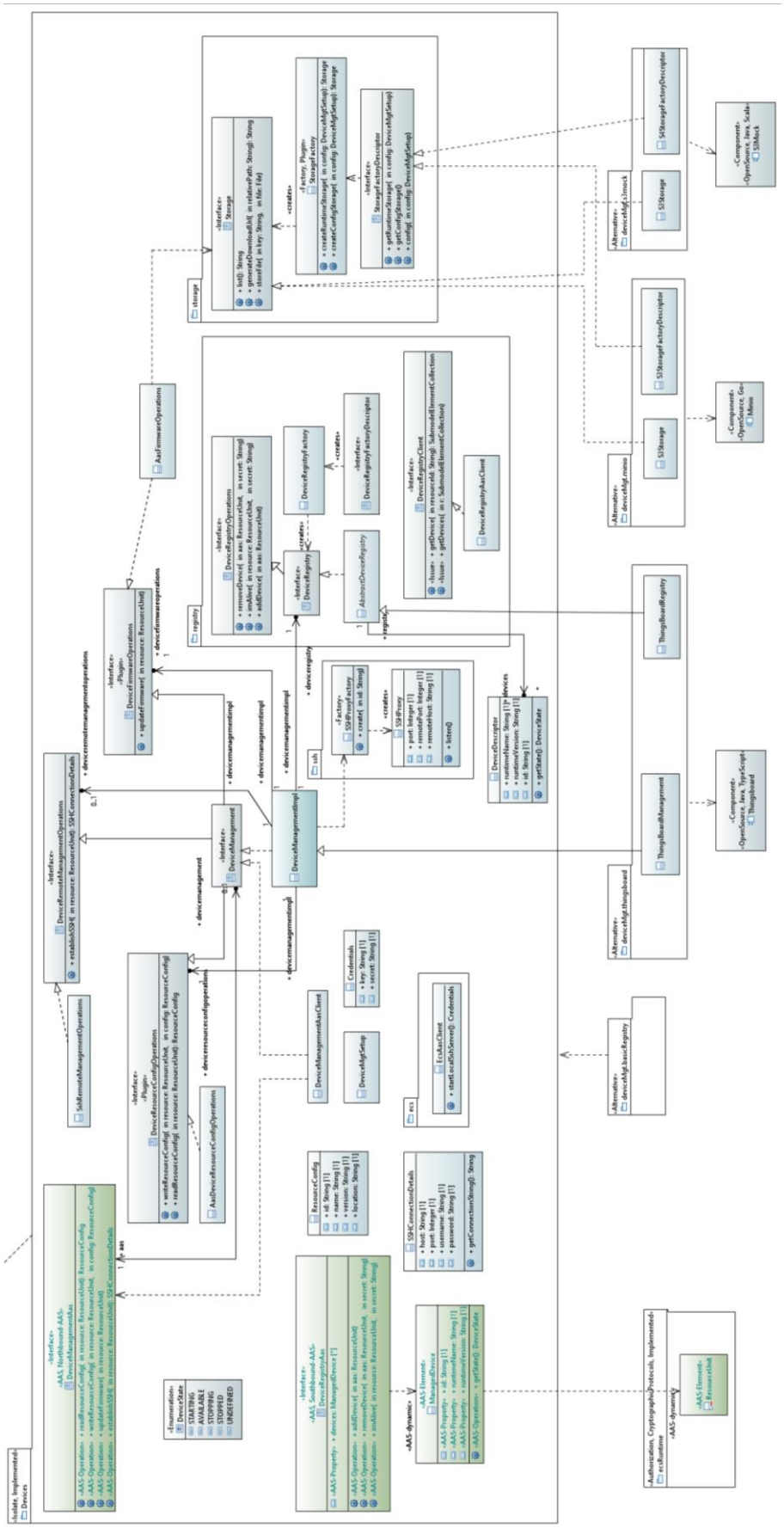

Figure 27: Device management (comments cropped)

At the core is the DeviceManagement interface, which is composed of operation interfaces covering different aspects indicated by the requirements, such as resource configuration, remote management or firmware operations. The separation into different interfaces allows for a unified handling of implementation, AAS (client) and testing. The DeviceManagementImpl class unifies these interfaces (by delegation) and implements a default remote management approach via Secure Shell (SSH) based on temporary sessions created by request on trusted, registered devices. To rely on an existing, mature implementation, the communication is performed here via SSH streams rather than AAS (due to performance issues) or the transport layer (a pair of channels might be used for this purpose), although SSH may impose issues to Windows devices. Further parts of this component are the device registry abstraction and the storage abstraction. For both parts, the implementation is left open here and can be realized by alternative components, e.g., ThingsBoard, MinIO etc. As usual in oktoflow, these parts define specific interfaces to the abstracted functionality, a JLS descriptor to create a concrete instance for the interface as well as supporting classes like AAS client implementations. A specific AAS client class (EcsAasClient) offers access to an extension of the ECS runtime from Section 3.8.1 to create a remote SSH endpoint on demand.

As indicated in Figure 27, for a specific setup, the device management offers the following alternatives:

- **ThingsBoard** as central management component. ThingsBoard ships with load balancing mechanisms, an own selection of internal protocol and frequently is setup via Docker containers. It must be installed separately in addition to server components.
- **basicRegistry** as a simple, in-memory implementation of the device registry interface. Can be used instead of ThingsBoard, but does not provide a user interface or persistency mechanisms, i.e., can be used for test setups.
- **MinIO** calls itself as the world's fastest object storage server. MinIo requires adequate setup on the server side. However, MinIO is licensed since April 2021 under AGPL, i.e., only an optional integration is permissible and it might not be the primary choice for installations.
- **S3Mock** is a Java/Scala-based object storage server for testing S3 implementations. In contrast to MinIO, it can be used without license limitations (MIT license), provides access via the Amazon S3 interface, but accepts any authentication, i.e., can be used for test setups. In contrast to MinIO, the S3Mock integration contains a lifecycle descriptor and, if the setup includes a storage server section, starts also a local storage installation (on the central IT side).

It shall be noted that due to usage of the well-known S3 protocol/interface for the object storage (although the individual technical interfaces of S3Mock and MinIO differ), the object storage integrations can act as storage connectors to storages located in a cloud and accesses can be directed to a cloud if stated in the component setup.

The device management component supports a simple on/offboarding process, *currently without manual approval of the operations*. If explicit on/offboarding is enabled (by default, this is currently not the case to ease development), a device must be explicitly on-boarded or off-boarded. This may lead to the exchange/removal of security certificates or encryption keys. On devices, that were not on-boarded, the platform may not execute operations. *Neither exchange of security information nor denial of operations are currently implemented.*

The functionality of the device management has been validated through many fine-grained test cases, see also [Pid21]. There, the performance of the direct execution of individual device management operations using the ThingsBoard device registry and the MinIO S3 connector have been measured and take in average 8-170 ms. If the operations are executed via the device management AAS sub-models, the operations take in average 11-204 ms.

3.6.3 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 platform foresees a set of generic built-in monitoring probes (cf. Section 3.5) 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 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 3.5 or Section 3.6.1, respectively), the main task of this component is to aggregate the information on IT infrastructure level (see also [SSE21]). The aggregation of the received values shall follow existing guidelines, approaches, relevant standards or norms in Industry 4.0. As the platform shall operate across a plethora of resources (and connected or underlying platforms and their resources, if available), the monitoring component shall foresee (optional) hierarchical aggregation to distribute the input load and to increase the efficiency.

The basic requirements for the monitoring component in [ESA+21] focus on devices/resources, services and alarming/alerts, in generic or application-specific fashion (e.g., through specific monitoring services hooked into the data processing chain). Akin for the device management, one important general requirement is R7 which requests the use of AAS for the interfaces of all layers/components in the platform. On the one side, the monitoring must take the information in the platform AAS on available resources into account and use the information provided by services and resources through their local monitoring. This component may require further properties than those, e.g., described in Section 3.6.1. On the other side, the device management shall provide relevant (aggregated) information and own operations to upper layers such as the user interface of the platform. The functionality of the monitoring component shall rely on underlying information (through AAS events[69] and polling, R11) in the services/resource sub-model elements collections. However, due to potential performance issues of the AAS approach, for urgent alarms/alerts also a second path via the Transport component is more adequate.

As described for the device management, the monitoring component must take the virtual character of the 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 platform. This allows taking other 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[70].

Primarily, for the monitoring component Java 1.8 compatible libraries shall be used, this constraint can be relaxed for the central monitoring components. As first (alternative) monitoring component we decided for an integration of the Prometheus service and resource monitoring approach (open source, Apache License). Prometheus is based on gathering data from HTTP/REST servers exposing monitoring endpoints, allows for configuring evaluation rules on the gathered data, stores the data in a time series data base and exposes the aggregated information again as HTTP/REST service endpoints (including an alert manager). However, direct HTTP access across all resources in a production system may not be permitted, i.e., some intermediary representation might be required.

One approach could rely on directly reading out the platform AAS, the devices and services AAS or submodels. Initial experiments using BaSyx as backend for a web-based UI (cf. Section 3.12) are promising. Here, standardized submodels (as started by the IDTA for a resources submodel) could lead

in the future to existing, reusable components for AAS based scraping of monitoring information, and, thus, could ease the effort of integrating such monitoring systems.

An alternative approach, as illustrated in Figure 28, is to provide the monitoring information through the transport layer as envisioned in [CE21]. For Prometheus, this is then similar to approaches allowing Prometheus to monitor resources over network borders, where a firewall or a gateway provides a proxying service[75], which collects all relevant information in the subnet on behalf of Prometheus and offers the collected information on individual endpoints to the scraping process of Prometheus. Although this requires an additional server process in the Prometheus case, it also allows for the flexible integration of other monitoring systems, as the data is provided independently and just must be translated into a format that can be understood by the respective monitoring system, e.g., a transport-to-MQTT translation that feeds information into the monitoring component.

In our case, the integration of Prometheus into the platform receives the monitoring data of individual resources via transport communication, exposes this information in an own (local) web server and adjusts the Prometheus configuration so that new devices are considered for scraping via the (local) web server. The implementation of such a metrics exporter is prepared in the generic monitoring component, while the Prometheus-specific integration is done in the alternative Prometheus integration component. The Prometheus integration component also contains the Prometheus binaries for Windows and Linux as well as an own lifecycle descriptor that starts and shuts down Prometheus. Within this lifecycle, a bridging metrics exporter as well as an alert monitor are started. We use the Prometheus alertmonitor[76] (Apache License) as the Prometheus client library does not provide support for alerts. The alertmonitor scrapes the alert manager HTTP API of Prometheus in regular fashion and turns alerts into alert instances of the transport layer (alert stream, see **Error! Reference source not found.** in Section 7.1). The setup of Prometheus is defined in the configuration model and the setup information is generated during the platform instantiation process. Part of this setup is also the information, whether we rely on an installed Prometheus server or whether we have to start the included binaries.

The monitoring component defines an own AAS submodel, which currently consists of a list of recent alerts. Individual, aggregated monitoring values as well as changes of monitoring rules will be subject of the next releases.

Ultimately, the Prometheus monitoring component is configured and integrated through the configuration model and the platform instantiation. However, realizing the bridging approach requires careful handling of the embedded Tomcat webserver, as otherwise in particular our default test broker Apache Qpid may throw `NullPointerExceptions` when the Tomcat instance creates server contexts. This would significantly limit the flexibility of choosing transport protocols in the platform. The current approach works with a single root context, into which a default metrics servlet and one servlet per device in an platform installation are added. The default metrics servlet is added statically when the server instance is created, the device servlets upon status messages of the platform. Further evaluation also of the performance will be subject to further work / releases.

Some components conflict regarding their dependency versions, e.g., in the past BaSyx and Spring regarding the required versions of the Servlet container Tomcat, partially running both in the same JVM was not possible. For this purpose, the platform instantiation creates an own monitoring component that can be executed besides the platform services in an own process. Moreover, the Prometheus monitoring component defines three lifecycle profiles, the complete Prometheus

[75] https://github.com/pambrose/prometheus-proxy
[76] https://github.com/matjaz99/alertmonitor

integration, only Prometheus or only the integration. In one of the next releases, we may abandon the creation of an own component and rely only on the lifecycle profiles.

The monitoring of system-level meters and application level meters (data items received/sent) has been validated through the Prometheus UI. Added resources (ECS-Runtime/Service Manager) when they occur are taken up, system- and application-level meters are categorized according to device id (and service id for application-level) and displayed individually. Aggregated values or rates can be calculated from this information on Prometheus level. Currently, the underlying approach based on micrometer automatically adds several technical system-level meters. Moreover, Spring also adds additional meters. Most of these meters are not relevant on platform level and could be filtered out. Also per monitoring diagram, currently all meter information is transmitted, while updates could focus on changing values and omit already known static values like descriptions or monitoring units. Both improvements are out of scope for this release, but planned for the next release of the platform.

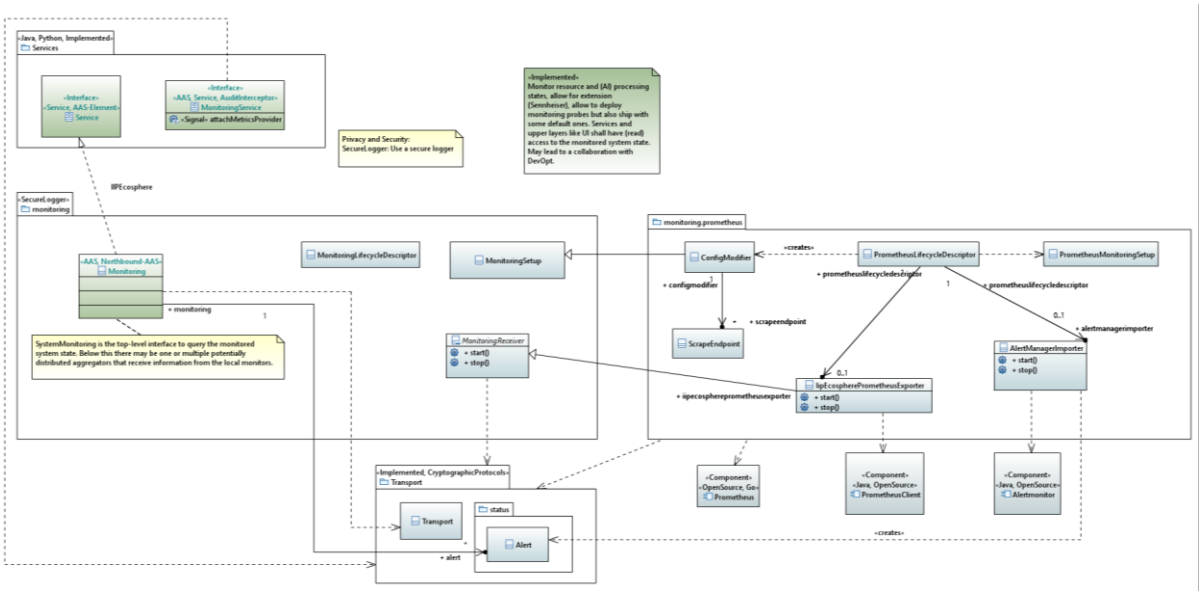

Figure 28: Monitoring (comments cropped)

3.7 Storage, Security and Data Protection Layer

The Storage, Security and Data Protection Layer is responsible for managing security aspects of the platform based on the platform configuration, for offering security-enhancing services (such as anonymization or pseudonymization), but also for secure integration of (encrypted) data lakes or clouds. As discussed in Section 3.1, the purpose of this layer is not to realize typical cross-cutting security mechanisms. We do not focus on the configuration aspects (R40a, R40b, R41a, R42, R44, R64a, R65a) here, as we do so later in the discussion of the Configuration Layer in Section 3.9.

3.7.1 KODEX platform service

The privacy enhancing service in this layer integrates the KODEX privacy and security engineering toolkit[77] by KIPROTECT into the platform. Currently, we focus on two alternative integration forms, one via command line streams (`AbstractStringProcessService` from the Java service environment) as well as a REST-based alternative. It is important to mention that KODEX is a generic tool that requires some form of setup to operate on the incoming data in the intended manner. Thus, KODEX acts here

[77] https://heykodex.com/, https://github.com/kiprotect/kodex

as a blueprint for rather generic data services in the platform. It is interesting to mention that KODEX is realized in GO, i.e., not in Java, and it was the first external data processing service integrated into the platform. To cope with the genericity of KODEX, some design decisions were made for KODEX that apply analogously to other external services:

- The `KodexService` is parameterized over the incoming and outgoing data types. To transfer data instances correctly to KODEX, respective type translators (the more generic form of a Serializer) are required. These type translators shall be provided by the utilizing code, e.g., a Spring Cloud Service node generated from the configuration model, which collects the knowledge about incoming and outgoing types of all service chains of all applications on a certain platform instance.
- The customization of KODEX happens in terms of certain files that specify the data model. Akin to the type translators, the contents of these files are determined upon integration into a service processing chain and shall be generated from the configuration model. These files shall be packaged into a ZIP archive (named according to the using node in the service chain) and stored in the service implementation artifact as specified in the respective process part of the service deployment descriptor, which is also contained in the service implementation artifact. When starting the service node, the deployment descriptor is consulted, the artifacts are extracted and the customization files are placed into the home directory of the process implementing the service, here KODEX.
- In the (extracted) home directory of the process, also the service implementation must be located, i.e., in the KODEX case the operating-system specific binaries. Such implementation files shall be packaged into a “binary” Maven ZIP artifact and deployed along with the service integration code, here `KodexService`. When integrating the generic service code into a service chain, the Maven identification of the service implementation, here KODEX, is known, and so is the deployed implementation (here binaries for different operating systems). During automated instantiation/integration, the “binary” ZIP is packaged into the service implementation artifact and named respectively so that it can be extracted upon service start along with the customization files as described above.
- Upon code generation of the Spring nodes, further customizations may happen, e.g., service-specific customization files could be created.

Further integrations, e.g., using the REST API of KODEX as well as a performance comparison among different forms of the integration are currently being conducted. Initial results confirm that REST outperforms command line streams on Windows. For example, processing a batch of 1000 tuples, windows with command line streams takes 15 ms per tuple in average, REST on Windows 0.22 ms, command line streams on Linux 1.4 ms and REST 2 ms on Linux.

Regarding licenses, it is important to mention that at the point in time of writing this document, KODEX is licensed under AGPL. However, viral AGPL rules do apply to binary code, i.e., using the KODEX binaries with respective credits does not taint the license limitations of oktoflow. Moreover, KODEX is only integrated, if it is explicitly used as a service in an application and only becomes active when the respective service chain is started.

3.7.2 Influx DB connector

Although technically belonging to the connectors layer (Section 3.4.2), the Influx DB connector is logically part of the data storage and protection layer. Basically, this connector is intended to write data points transported as part of oktoflow streams to an Influx DB. Therefore, incoming data tuples are split into individual Influx entries of the same measurement id and time point. Receving a data stream from an Influx DB, e.g., for AI training, requires a connector trigger query, in this case a simple timeseries (start/end time without further filter conditions) or string queries (in flux query format).

While the simple timeseries query is turned into a stream of monotonic ascending timestamps, this must be explicitly ensured for the more open string-based queries. When receiving the individual data points, the connector joins them back to data tuples and uses the generated data transformers to ingest corresponding application-specific types (optional fields may be helpful if the data in Influx is partially incomplete). The periods between two successive datapoints is obeyed as far as possible, if not overridden by a fixed delay per tuple given by the trigger query. The connector supports Influx v2 authentication support via issued tokens and Influx v1 support for username/password authentication.

The connector is subject to regression testing based on mocking the Influx client (the usual approach) and further was manually/in internal examples tested with InfluxDb2 version 2.7.6.

3.8 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. In this section, we briefly discuss the specific requirements (Section 0), the integration of RapidMiner RTSA as AI platform service (Section 3.8.2), further service candidates ahead (Section 3.8.3) as well as the deferred AI service concept.

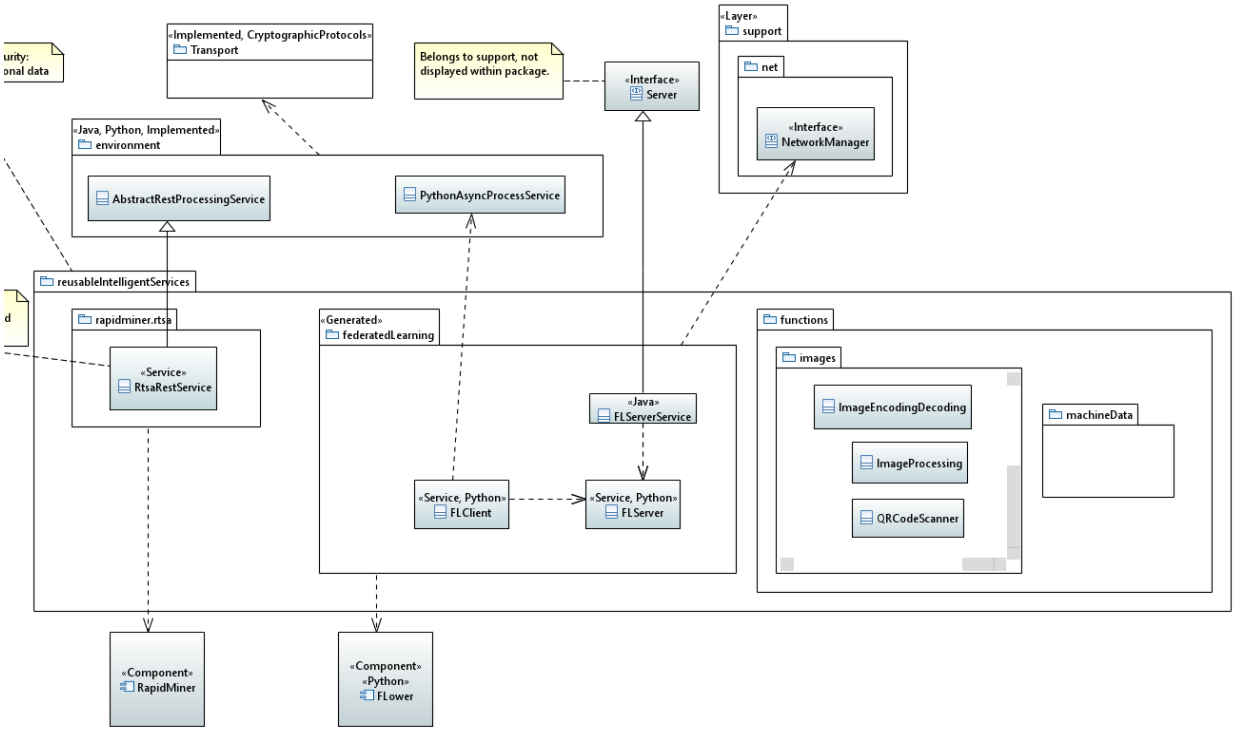

Figure 29: Reusable Intelligent Services and data processing function library (cropped)

3.8.1 Data Processing Function Library

Application of AI methods encompasses more than just the AI methods. Usually, also functionalities for data pre-processing etc. are required. As a basis to meet the requirements regarding pre-processing and data transformation functions (in [SSE21], e.g., frequency analysis), we equipped the platform with a library of functions for data processing. In this version of the platform, first functions were integrated, namely

- Image transcoding from/to base64 strings.
- Image processing such as grayscaling, rescaling or thresholding.

- Barcode/QR-code detection based on the Java library zxing[78] and, as optional fallback, the Python library pyzbar[79]. For the Python fallback, respective packages must be installed.
- Generic channel-based time-series aggregation. The need for this functionality arised during the realization of the EMO'23 demonstrator where a condition monitoring AI shall be fed with channeld data from a linear drive and, from a different sensor, related energy measurements. The energy measurements were delivered as individual telegrams representing a time-series, while the condition monitoring AI shall be fed with summarized data over different measurement channels for the time span the drive is operating. As two kinds of data, individual data points for all energy channels per point in time and one data point for all energy channels per point in time were available, we realized two generic, re-usable aggregators, which were then instantiated in the respective aggregator service of the application.

3.8.2 RapidMiner RTSA service

RapidMiner is a pioneering company in the fields of data analysis and AI. Their work on the RapidMiner Platform (in Java) and the RapidMiner Studio shows that AI composable from building blocks is not only a vision. One fundamental component in the RapidMiner ecosystem is the Real Time Scoring Agent (RTSA), a REST-based execution environment for deployments created by RapidMiner Studio.

Following the ideas in [SSE21], a separation of data science exploration and design processes from the actual execution/deployment is desirable. Thus, RapidMiner is an excellent example for such an approach integrated into the platform. While the DataAnalyst can first create a data science process for given data in his/her own environment, the created process (as "deployment") can later be deployed by the platform and executed under the control of the Service Manager on top of RTSA. For the integration, mainly the data input/output formats must match, i.e., the data provided by the platform (output of a connector/service) becomes the input for the RTSA deployment and the output of the RTSA deployment becomes an input for upstream services.

Along these lines, the platform-supplied `RtsaRestService` (see Figure) integrates the RTSA and links the platform data streams to the input/output of the RTSA. The realization is similar to the REST-based integration of KODEX discussed in Section 3.7.1. In the configuration, a deployment file for the RTSA is specified, which is packaged by the application/platform instantiation along with the RTSA binary into the respective service artifact. Here it must be considered that RTSA is a commercial service, i.e., it requires a license file and cannot be distributed openly. For testing the service integration and the platform instantiation the test part of the component ships with an RTSA mockup (`FakeRtsa`), which acts as a REST server pretending to be an RTSA instance with a deployment. This fake RTSA can be configured in limited form to transform the input data, e.g., by changing fields or adding fields having a random number value. While this is not needed for the plain RTSA testing, it can help testing data flows for an application if a real RTSA is not available. For the platform instantiation, either the real or the mocking RTSA and its deployment are given in a specific folder according to the platform naming for binary files from which the instantiation process can take up the binaries.

3.8.3 Flower-based Federated Learning

One AI method that is of particular interest for the industrial production and for the integration within the platform is federated learning. Federated learning services act as services as they consume data and produce predictions, but they also share information about their AI model with other federated learning services (usually of the same application) so that the other services can learn new knowledge faster, e.g., acceptable anomalies. So far, also due to the client-server approach, federated learning is often viewed as a complex approach, which might be used more frequently in the industrial practice if

[78] https://zxing.org/w/decode.jspx
[79] https://pypi.org/project/pyzbar/

an easy-to-use form of federated learning would be available. We aim at demonstrating one approach to easy-to-use federated learning in terms of the upcoming Hannover Messe 2023 IIP-Ecosphere demonstrator. Currently, the integration is still in development, but technically it is based on the service/server communication introduced in Section 3.5.3.

In particular, we integrated the Flower[80] framework written in Python. Services (Federated learning client) and the server (usually assigned to one or multiple applications) are specified in the configuration model. Both, client services and server service are generated into the Python part of the application code templates based on the configurated settings (including the basic technology code for, e.g., tensorflow and numpy), must be completed during service realization and are executed through the Python Service Environment and the respective Java integration counterpart (`PythonAsyncProcessService`). As explained in Section 3.5.3, in contrast to the client services, the server does not consume/produce regular service data streams. To be manageable within the platform, the server needs a Java counterpart which is also a `Server`, which manages the (hidden) service application lifecycle and relies in most of its functionality directly on `PythonAsyncProcessService`. Akin to the service and server code, respective test code for the Flower services is generated into the Python test part of the application code templates.

3.9 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.

Figure 30 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.

The configuration model follows the layered architecture of the platform, i.e., each platform layer is represented by a configuration module. Figure 30 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 30 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.

[80] https://flower.dev/

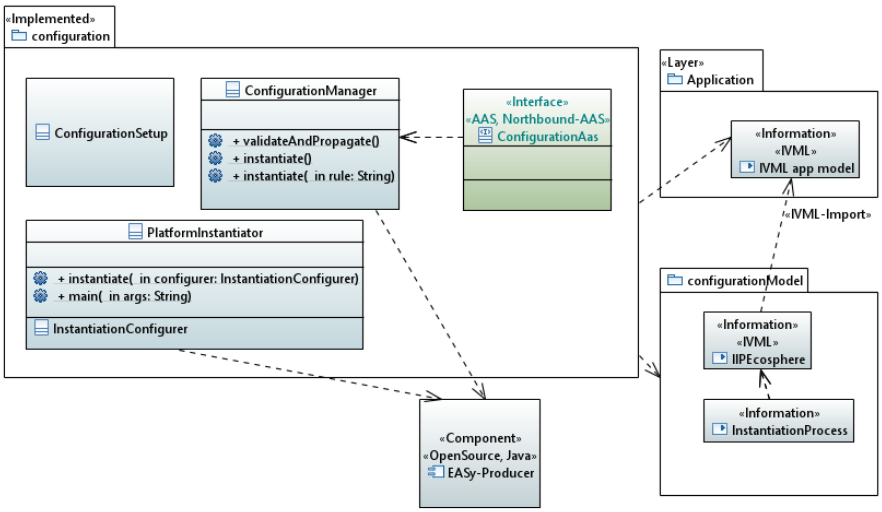

Figure 30: Configuration and instantiation of Definition of applications and orchestration of services (comments cropped)

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. The configuration model, the actual settings and in particular the application data flows are reflected in the `ConfigurationAas`. To ease UI integration, the `ConfigurationAas` offers operations to read out the data flow graphs in various formats, to write back/delete data flow graphs, to create and delete configuration variables, to change configuration variables and to initiate the application instantiation. For these operations, it is essential that no arbitrary modifications are permitted. The result must always be a valid configuration that can be instantiated. For this purpose, all operations perform a validation through the IVML reasoner and persist the configuration model only if the model is valid. Further, 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. The `PlatformInstantiator` offers various modes, ranging over the instantiation of interfaces for applications, the full instantiation of applications, the instantiation of platform components, etc.

The classes of the Configuration component are separated into three projects, a) `configuration.interface` declaring the basic classes and the interfaces to realize configuration technology plugins, b) `configuration.easy` realizing the interface in terms of the EASy-producer configuration technology (which itself ships with a complex, potentially conflicting tree of dependencies including Eclipse components, xText etc.) and c) `configuration.configuration`, the actual home of the configuration model, which shall be packaged and distributed using a technology-independent component name. Further configuration technologies may implement the configuration interface in terms of an own plugin and share their model in respective folders in `configuration.configuration`.

The configuration meta-model is extensible, i.e., it consists of a core model as well as extensions for devices, services and connectors. Already mentioned extensions integrate, e.g., KODEX (Section 3.7.1), Flower (Section 3.8.3) or RapidMiner (Section 3.8.2). One further, particular extension represents an integration of the MIP technology[81] magnetic identification sensor, one of the IIP-Ecosphere dynamic

[81] https://mip-technology.de/

demonstrators. The extension specializes the platform-supplied MQTT connector through specific data types for the MIP sensor technology and, thus, eases the use of this technology in user-defined applications. Besides a connector, also some code to control the sensor is helpful for application projects. This code is supplied by the **configuration default library**, a platform component based on data types defined by the MIP extension of the configuration model. Thus, the configuration default library showcases how to supply code and related dependencies with the configuration model and provides a blueprint for own extension libraries. For short, the build process of the default library just instantiates the interfaces of a minimal platform configuration without any IoT application, i.e., only defined types are generated. These Java and Python classes can then be used in code, which in turn, as Maven artifact, used in user-defined applications. It is important to mention that such libraries intentionally ship without the generated types, which are then generated as part of the respective IoT build process.

The configuration meta-model and the platform instantiation are subject to regression testing in the continuous integration. While the validation within the configuration model is currently not as extensive as it could be, partial results of the platform instantiation have been validated for functionality, e.g., the instantiation of executable platform components, the creation of several stream-based IoT example applications as well as the automated low-code generation of connectors. In particular, the connectors were validated in the platform use case studies with partners (cf. Section 3.4.2.2), i.e., the respective input/output formats have been modeled in configuration, the connector service integrations have been generated and the intake has been validated. To demonstrate the setup of the platform, the platform instantiation as well as the creation of example service artifacts is part of the Docker platform containers provided on DockerHub.

3.10 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 31.

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.

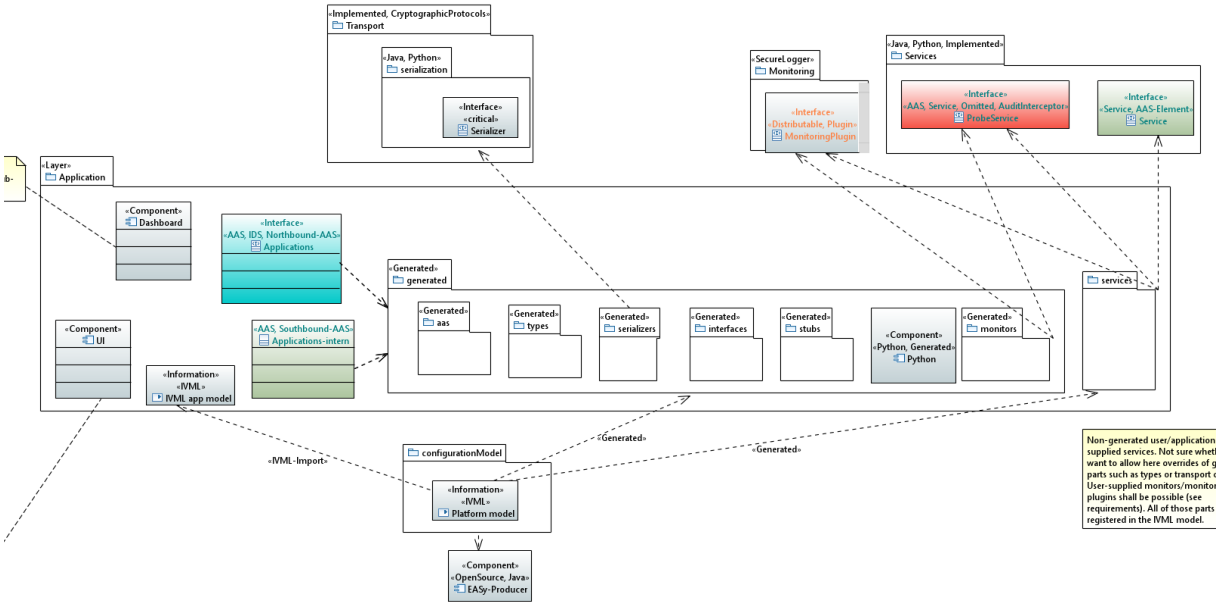

Figure 31: Application Layer (comments cropped)

3.11 Platform Server(s)

As discussed above, the 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 platform, e.g., to start containers or services.

Figure 32 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 three purposes:

1. Powering up the servers to run the platform. Therefore, the component defines a lifecycle descriptor (`PlatformLifecycleDescriptor`), which reads information from the `PlatformSetup` representing the YAML setup file. The lifecycle descriptor is loaded via JSL into the `LifecycleHandler`, which, in turn, is called by the `platform` 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 contains the platform "nameplate" (`TechnicalInformation` sub-model), further software-specific information (`Platform` sub-model) as well as a listing of all available application `Artifacts` (service artifacts, containers, deployment plans).
2. Observing a heartbeat of the distributed components via their regular monitoring messages, in particular ECS-runtimes and service managers, to detect whether a component instance dropped out accidentally and whether the respective AAS submodel still exists and shall be removed.
3. Providing a simple command line interface (`Cli`) to experience the operations of the platform. For the command line interface, simple means that we (currently) provide access to some, particularly rather low-level functionality of the platform, which incrementally shall be taken over into the platform, e.g., which service shall be deployed where shall be part of the deployment component in the configuration layer. In addition, first higher-level commands such as executing a deployment/undeployment plan are provided, which adds/removes services implementation artifacts and starts/stops contained services that shall be distributed across multiple devices. The command line interface does not rely on the lifecycle mechanism, but on the `PlatformSetup` and, in particular, on the AAS clients of the service and the resources layer to ease executing the operations defined there. Figure 33 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 33, 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.

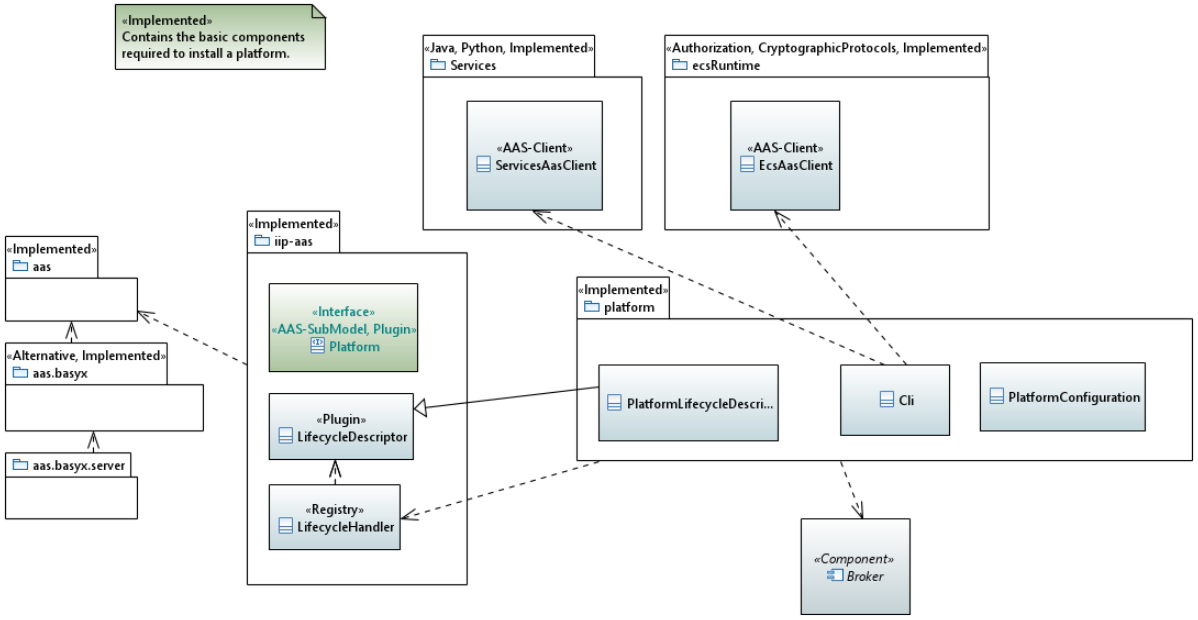

Figure 32: Platform server(s) component

```
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 33: 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:

- BaSyx issues exceptions when checking whether a non-existing AAS exists through trying to access 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.

The platform server processes write a simple YAML file (`oktoflow.yaml`) with the actual URLs of AAS registry and AAS server into the same folder as the process identification files (PID). Since version 0.7.0, the platform can be configured to use ephemeral ports for AAS registry and server, in particular for testing purposes. This file can then be used to setup testing processes in the right manner.

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 7.4 on how to install, instantiate and containerize the platform, i.e., to perform the steps that we also executed for validating the command line interface and the instantiated platform components. The platform CLI also supports creating snapshots of the platform AAS[82] that can be explored with the AASX Package Explorer[83].

As the platform layer shall also be used as basis for a Management User Interface (cf. Section 3.12), additional information that is usable on that layer may be required. For this purpose, we added a listing of available artifacts (containers, service artifacts, deployment plans) to the Platform AAS, so that they can be selected, inspected or executed on UI level. Akin, further information and interfaces regarding the configuration will be required.

3.12 Platform Management User Interface

As already stated in [SSE21, ESA+21], no real user interface was scheduled for the platform in the grant agreement. However, the value of an accessible and usable user interface for a platform (over a simple CLI as discussed in Section 3.11) is evident. For this reason, the platform team aims at providing a management user interface, i.e., a web user interface that allows for managing the platform operations such as starting or composing an application.

Currently, the user interface focuses on displaying the information that is accessible through the CLI as well as basic management operations provided by the CLI as well as on editing the configuration model / support service implementation. Here, the Platform AAS forms the information model the Management UI can rely on and one interesting question is whether it is possible / to which degree it is possible to realize an efficient Management UI based on an AAS. Challenging tasks include reading the nested AAS REST structures or calling AAS operations, as BaSyx does so far not provide TypeScript support. While for some operations on a command line we can assume that the user can look them up, e.g., the URI to a service artifact, this may not be the right approach for a Web UI. Thus, as part of creating a management User Interface, we also have to provide additional information, e.g., on the available artifacts, containers or deployment plans and to make them “executable” via the UI.

The management user interface is separated into three main parts / views, the available and on-boarded resources (e.g., edge devices) known through their AAS, the platform and application configuration as well as runtime operations such as starting or stopping applications. The configuration part is structured according to the logical flow of setting up a platform and specifying applications. Below we present some screenshots of the user interface.

Figure 34 illustrates the overview of the available resources allowing for more details from the respective AAS on demand. Each on-boarded resource is displayed with information from its device

[82] Currently it seems that BaSyx allows only a single snapshot per run. This may change in future versions.
[83] https://www.plattform-i40.de/IP/Redaktion/DE/Newsletter/2019/Ausgabe21/2019-21-Praxisbeispiel2.html tested with version 2021-08-17.alpha.

AAS, e.g., the vendor and the product image. Pressing the button “resource details” leads to an overview of technical information, a combination of the static device AI and the runtime information contributed by the platform monitoring.

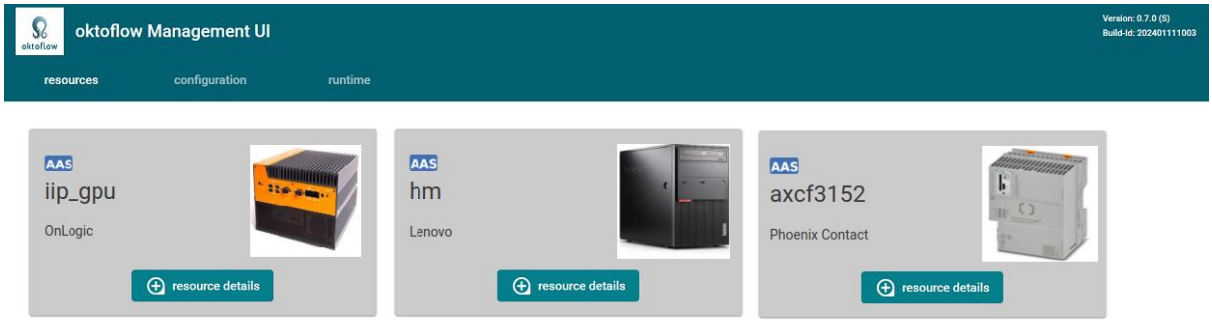

Figure 34: Management user interface, available resources.

The next screenshots illustrate the configuration part. The sub-items of the configuration menu indicate the sequence of steps to define an application: Constants, types, software dependencies, AAS nameplates, servers, services, meshes, applications. Most of these steps are displayed as lists with a dialog-based editor for creating or editing respective elements. Figure 35 shows the configuration of AAS nameplates to be used in the configuration of individual services. Figure 36 depicts the graphical viewer/editor for service meshes and Figure 37 the configuration of applications, here with actions to obtain application code templates and to integrate service implementations to an application.

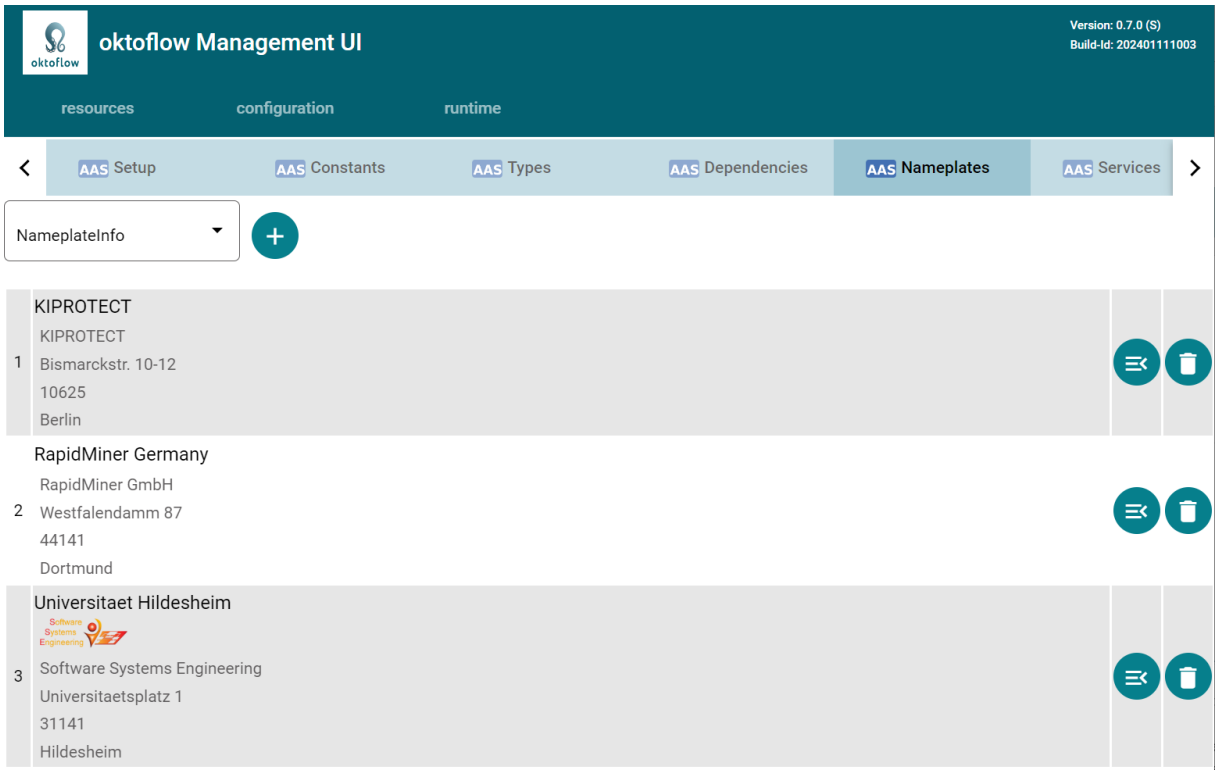

Figure 35: Management user interface, configuration of AAS nameplates for service vendors.

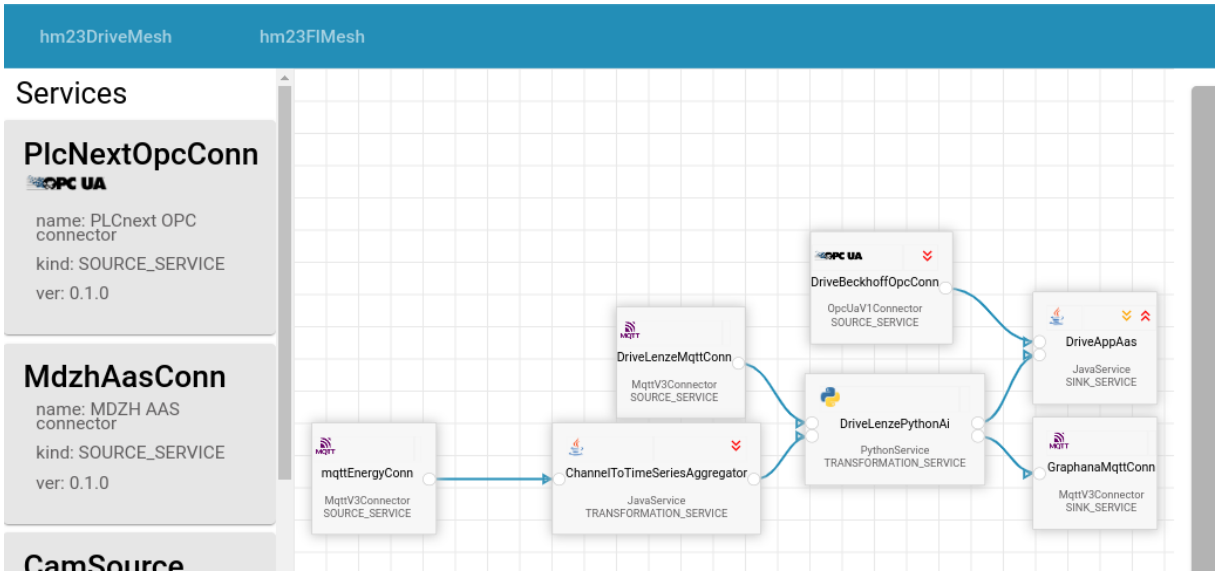

Figure 36: Management user interface, configuration of service meshes.

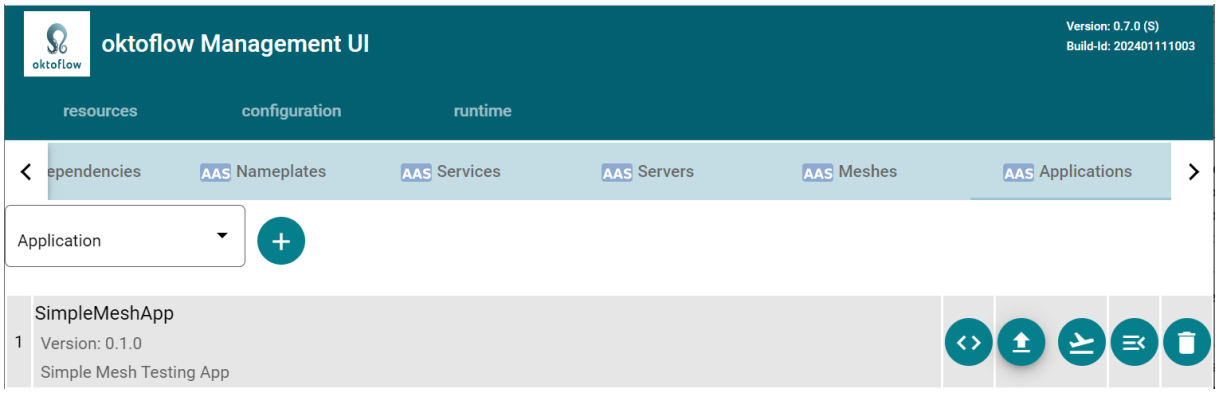

Figure 37: Management user interface, application configuration allowing to obtain implementation templates and integrating implemented templates.

In the last step illustrated in Figure 37, the user can generate implementation templates supporting the realization of services. When pressing the "generate template" or the "integrate application" button, a specialized form of the instatiation is executed. If successful, the result of "generate template" is a download offer for interface and template artifacts. Without specific Maven repository, the interface artifact shall be unpacked and installed (`mvn install`), the implementation template shall be imported into Eclipse and utilized as discussed in the Tutorial Videos. Finally, the service implementation can be uploaded and integrated using the "integrate application" button. If a maven repository is specified in the configuration (e.g., SCP or FTP upload to a maven repository, a Sonatype Nexus[84] or a JFrog Artifactory[85] server), all created application-specific build processes will try to use that for deployment so that except for the download of the implementation template, no further up/download is needed. As stated above, we assume in the simplest form we assume that you are developing/running Maven on the same computer as the platform. In addition, the platform also allows for a setup with a (local) maven repository, where all deployment operations go to.

In the runtime part, the user can access the available deployment plans (Figure 38), upload new deployment plans or start enabled plans. Starting a plan multiple times may (if not prevented by the plan) lead to the execution of multiple instances of the same application. In the instances view, all running application instances are listed and can be undeployed individually. Figure 39 illustrates the currently running services with interactive access to the service logs (Figure 40).

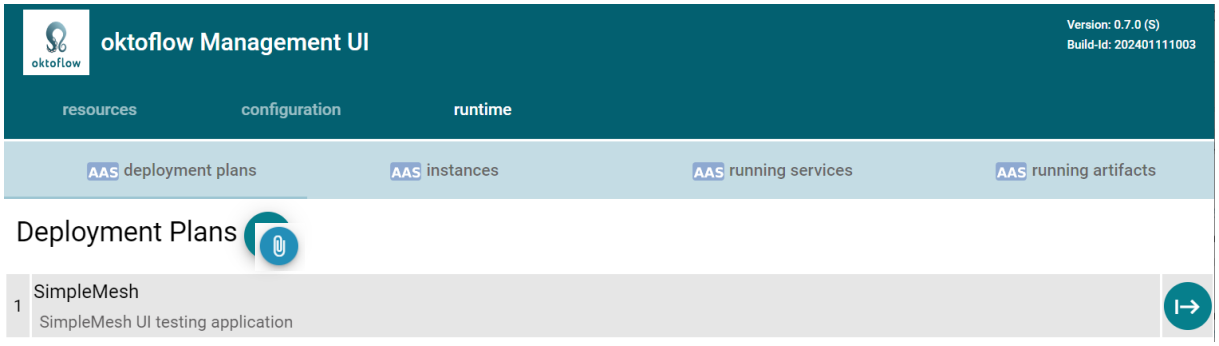

Figure 38: Management user interface, overview of deployment plans and actions to start application instances.

[84] https://help.sonatype.com/repomanager3/product-information/download
[85] https://jfrog.com/artifactory

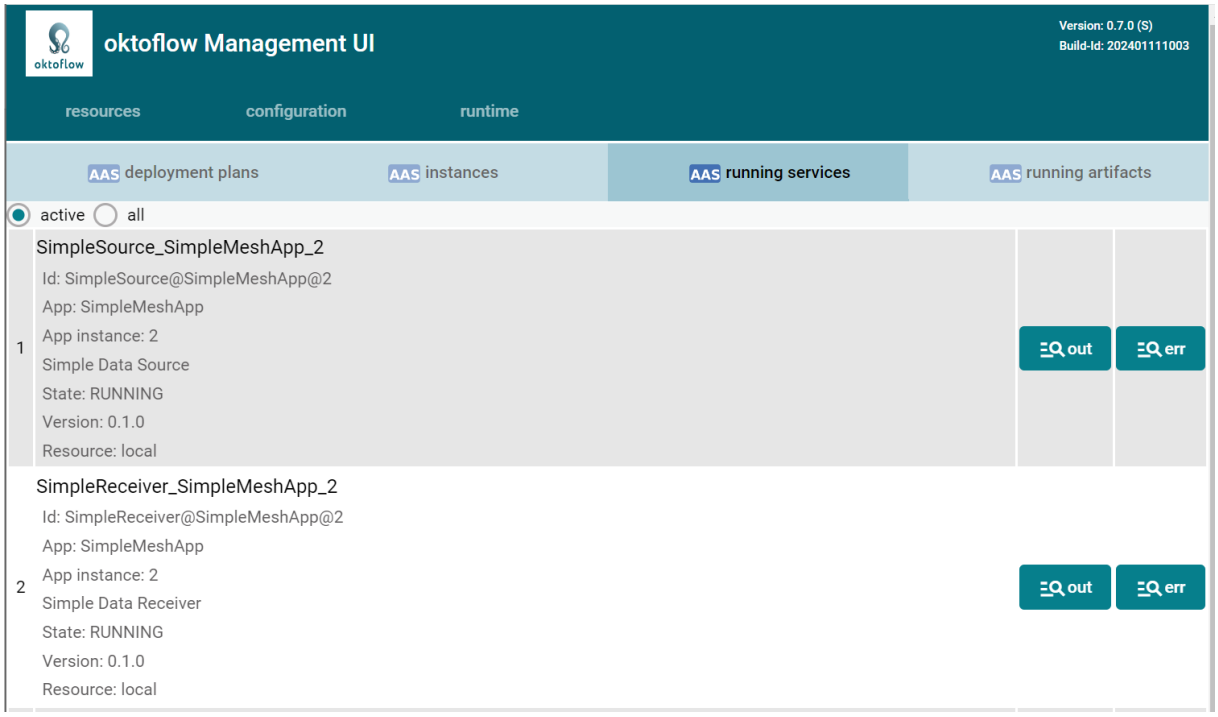

Figure 39: Management user interface, overview of running services actions to access the runtime service logs.

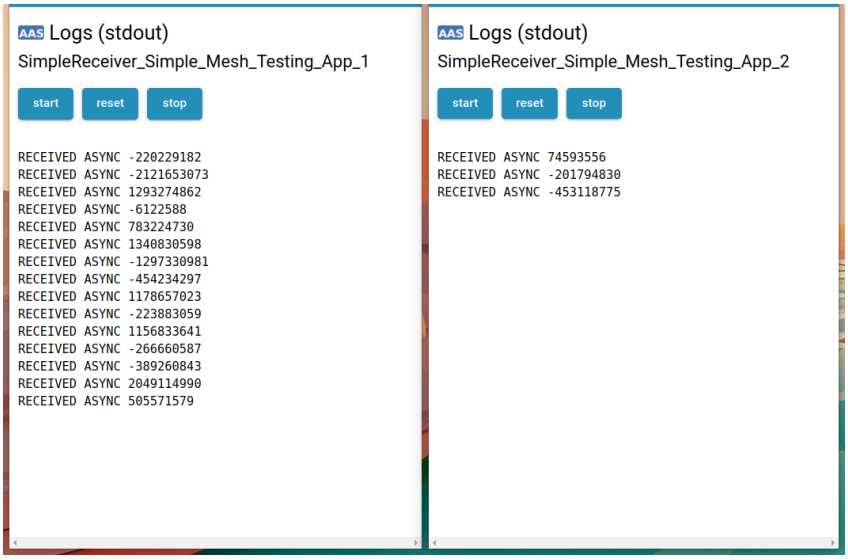

Figure 40: Management user interface, active runtime service logs.

The platform management user interface is able to configure most aspects of an application including the service meshes, i.e., the data flow graphs. However, still aspects are known to be missing and will be added in future releases. On this level, configuring means adding new typed configuration variables (a whole mesh is represented as variable), editing or deleting existing ones. Each operation requires a validation of the underlying configuration model so that consistency issues or problems can be identified before instantiating the results. Currently, also the platform and application instantiation functions of the management interface are not fully integrated.

The user interface requires some form of setup, in particular knowledge about the installation location of the platform AAS servers. To resemble UI release versions and integration with the platform instantiation and installation approach, the management UI allows for compiling the TypeScript code

for Angular while allowing for an external setup through a JSON file[86]. The compilation happens as part of the Continuous Integration of the platform, the customization during the instantiation based on information in the configuration model. The platform instantiation turns the compiled Management UI into an instantiated version, where in particular the settings in the Angular environment are adjusted or respective start scripts, e.g., for an Express webserver are generated.

As the management UI is based on the platform AAS, which is typically running on a different network port and may even run on a different server, access may need to be granted, in particular in terms of Cross-Origin Resource Sharing (CORS)[87]. By default, CORS is disabled in the configuration meta-model, but it is enabled for all accesses in the install package. CORS can be set up through the configuration variable `aasAccessControlAllowOrigin`, e.g., by setting the value to `"*"` (typically in `TechnicalSettings.ivml`, requires freezing as stated in section **Error! Reference source not found.**). If CORS is not explicitly enabled, usually a browser plugin is required.

Figure 41 depicts the structure of the implementing TypeScript classes, mainly in terms of Angular components (UI elements of different granularity with code, display and style) and Angular Services (re-usable functionality). From top-to-bottom and left-to-right, the implementation consists of the top-level Application module and the Application component that bootstrap the functionality and the dependencies. The `FlowchartComponent` and its subordinate feedback component integrate drawflow as flowchart editor and customize it for oktoflow service meshes. The `ServicesComponent` displays running services and via its associated `LogsComponent` displays runtime logs of individual services in own windows. The `DeploymentPlansComponent` displays the available deployment plans and allows executing them. Subordinate components allow for file uploads, status updates as the execution of a deployment may take some minutes and a subordinate `StatusBoxDetailsComponent` to display a sequence of related status messages. The `ListComponent` is rather generic and responsible for all configuration lists, e.g., for constants, datatypes, nameplates, servers, services, or applications. The most complex component is the `EditorComponent`, which represents an editor for complex configuration types that are selectable in the aforementioned configuration list.

The `EditorComponent` interprets the configuration AAS to dynamically derive editors for the complex configuration types, including mandatory and initially hidden optional settings. The `EditorComponent` further delegates to individual editor components representing individual configuration settings within a complex type, in particular for Boolean values, enumerations, AAS/OPC-UA language strings, references among configuration elements (e.g., applications link to meshes, services to servers) and representations of nested complex values (e.g., datatypes declaring service in/outputs), which, in turn, use a (filtering) instance of the `EditorComponent`. Similarly, when editing a refernce to another configuration element, the `EditorComponent` for the reference target is opened.

Furthermore, the InstancesComponent displays the list of running application instances and allows terminating individual instances. Finally, the `ResourcesComponent` displays the overview of the available resources, delegating a detailed view on a single component to the `ResourceDetailsComponent`.

[86] https://mokkapps.de/blog/how-to-build-an-angular-app-once-and-deploy-it-to-multiple-environments/
[87] https://de.wikipedia.org/wiki/Cross-Origin_Resource_Sharing

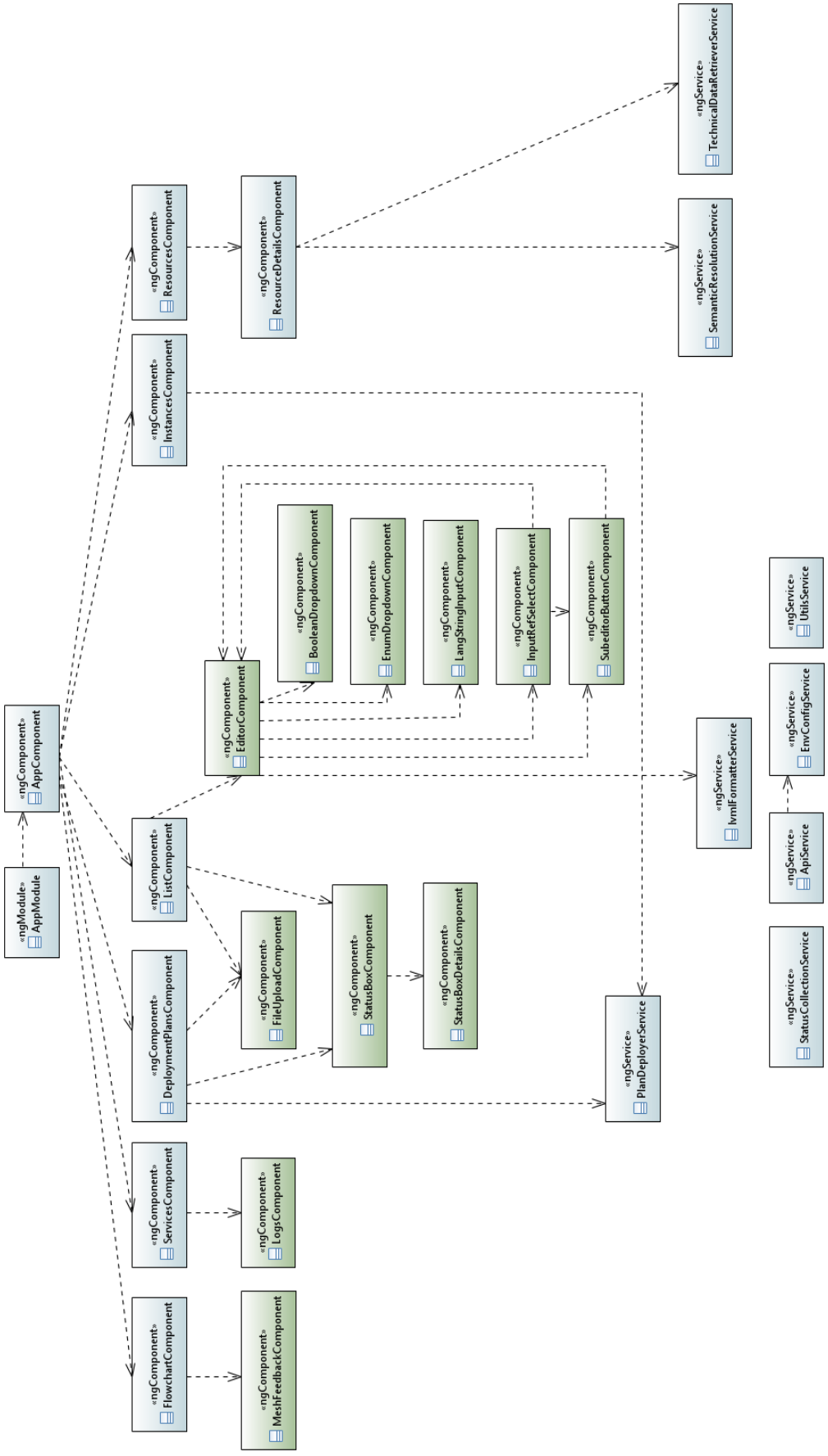

Figure 41: Management user interface, implementing components and services.

The briefly explained components utilize sharable services for realization. As services are made available through dependency injection, in principle every service can be used in any other service or component. However, in practice also the services form a usage hierarchy, for which we display in Figure 41 the dependencies from the main using components, e.g., `DeploymentPlansComponent` and `InstanceComponent` rely on the `PlanDeployerService` for executing/stopping individual deployment plans, the `ResourceDetailsComponent` in particular on the `SemanticResolutionService` for resolving AAS semantic IDs as well as on the `TechnicalDataRetrieverService` for the actual technical data of an individual resource. Moreover, the EditorComponent utilizes the `IvmlFormatterService` which, at its core, turns user input into IVML values that can be handed over to the platform for modifying the configuration.

In turn, these services depend on even more basic services, the `StatusCollectionService` which receives status updates from the platform, the `ApiService` which realizes communication primitives, in particular, to retrieve data structures from the configuration AAS or to call AAS operations there. The `ApiService` is based on the environment configuration service (`EnvConfigService`), which reads a customizable JSON file written during the platform instantiation. Finally, the `UtilsService` provides helpful utility functions, separated into those that are useful for components (they inherit from there) and static functions for data access and manipulation.

Since version 0.7.0, we are equipping the Management UI with regression tests (more than 100 tests). Although we might mock the platform services, in particular the AAS, we decided to test the Management UI against a real platform instance for better consistency. Therefore, the Maven plugins of the configuration component are used to instantiate a simple application including platform (AAS registry and server on ephemeral ports), start the platform and an ECS-Runtime-Servicemanager, if required also a simple application, run the Angular regression tests and shut down this temporary testing platform instance.

3.13 Test support

So far, we focused on the elements to construct the platform, the services as well as the applications. In this section, we provide a cross-cutting overview on the testing support, in particular to answer the question, how the platform supports the user in testing his/her own services and applications.

Besides internal component and service testing, in particular of services supplied with the platform, it is essential to test user-developed services as well as their interactions in an application. At a glance, the answer might be to construct a unit test and to test the supplied code. However, reality is not so trivial, as, e.g., connectors may be based on external devices or their server instances, e.g., MQTT broker or OPC UA server, and these devices may not be available in certain testing situations. Moreover, setting up a test for a single spring-based service involving a Java or, in the more complex case, a Python service (with involved Java integration and Python service environment) requires much internal knowledge and may even lead to problems if the service layer is equipped with alternative service execution engines that are not considered by the test.

As a general advice, we recommend to apply testing to all levels of an application, ranging from tests of the code that you supply up to entire applications. In particular, testing of services before running the application instantiation and packaging process can usually save much time, i.e., as usual, getting rid of failures in individual services shall be performed before integration tests of the application.

Moreover, we recomment to base the implementation of individual services or applications on implementation templates generated by the application instantiation process (cf. Section 6.8). Such implementation templates provide a structure of implementation projects where only explicitly

marked parts must be filled with code, e.g., glue code to own classes added to or imported into that project. Such templates also contain a customized build process for Java and, if required by the application configuration, Python.

Thus, the platform offers different forms of testing support that we will summarize here:

- **Testing the connectivity of an individual connector:** To validate that a certain connector correctly connects to its counterpart, e.g., a device or a server, the platform generates per connector a simple (template) connectivity test program that uses the specified connection information, creates a connector instance and requests data. These template programs are meant to be starting points for own, connector/device-specific tests and made available for execution through the generated (template) build process.
- **Testing an individual connector as a service:** Typically, a connector is a kind of platform-supported service. Thus, the connector itself, either a model-based or channel-based connector is already tested sufficiently by platform tests. The instantiation process wraps the connector into a service and adds generated input/output data translators or user-supplied data translators or event handlers based on the data specification of the configuration model to turn the generic connector type into an application-specific connector. Although the user-supplied parts shall be tested individually, it always remains unclear whether they will work correctly in the interaction context of a connector. For this purpose, the generated code for the configured connector sets up an environment that the connector can be executed individually. By default, the connector runs against the configured device, e.g., the OPC UA connector against the OPC UA server of the configured device. As this requires the device at hands, the test may fail in a CI environment. Thus, it is possible to execute the test as a stand-alone program (for the target environment) and to mock the connector (as set up in the configuration model). When the connector is mocked, the code generator wraps the connector differently into a service, i.e., it creates a connector instance, but detangles it from the execution while pretending to feed the connector output with data from a JSON file[88] (via functionality provided by the service environment). Such mocking data files wrap the entire input for a single point in time into a JSON structure, indicating the type of data to be used (in case of multiple outputs), delays between data points as well as repetitions of data points. The field names are the same as defined in the configuration model irrespective, e.g., of Java naming conventions. Such a mocked connector allows you to experiment with different data settings even in a CI environment.
- **Testing individual services:** For a service it is more likely to integrate self-supplied code. This code shall be tested individually. However, writing tests against internals of an infrastructure like the platform is not trivial. Thus, the implementation templates mentioned above contain template classes for production and testing code for both, Java and Python, as demanded by the respective application configuration. The customized build process of the generated templates performs compilation (Java), syntax testing (Python) and testing of the generated test code (Java, Python). In particular, Python services are tested by default by reading a JSON data file, feeding the data points through the generated serializers and data classes into the service and asserting the output.
- **Testing services in the service environment:** As for a connector, testing a service in its real execution environment lacks without further preparation the service execution environment, e.g., unpacking Python and AI models in the right folter, executing the surrounding Python environment or the service lifecycle through spring cloud stream. Here, further generated test

[88] For legacy reasons, the file name endings in some examples or generated implementation temples may look like YAML files. The mocking implementation of a connector actually looks for ".yml" and ".json".

cases set up an environment that fits to the actual service execution environment, per default Spring Cloud stream. Akin to mocked connectors, a JSON file determines the input data (which may consist of multiple data type instances, and may define a timed ingestion behavior), which is fed through the `DataWrapper` into the service through the actual mechanisms of the service environment or the platform, e.g., the data transfer. Resulting data, synchronous or asynchronous, is received by the test and basically emitted. In the generated version, all data received from the service under testing are asserted as true as we do not employ a data correctness specification at the moment. Basically, the test only ensures that data shall come out of a test (assuming that data was fed into it). Statistics about received data types are collected by default and can be used for simple asserts. Further, you may extend the test to determine more complex assertion behavior and to turn it into a real regression test.

- **Mocking of applications:** Although components may be working after applying the test opportunities discussed above, there is no guarantee that the integrated application will be working. Here, again, mocking may be required as not all devices or even software environments are available, that you would need for mocking tests. The basis here is that applications can be executed even without the service execution environment provided by an installed platform. For this purpose, the generated build process of an implementation template allows for starting an application through a mockup of the service environment, starting the services in an arbitrary ordes on the same machine (in contrast to distributed execution and controlled order as provided by the service manager of the platform). To support this, so far, as shown in the platform examples, we configure the applications with a Boolean switch, e.g., to change between local host addresses for testing and real internet addresses for production. Connectors can be mocked as above, e.g., if certain devices or their OPC UA servers are not available. Services can also be mocked, by replacing the service classes defined in the model, e.g., by extensions of the services that disable some functionality. One example is to replace a TensorFlow-based Python AI script (with one reason that you may not have required GPU capabilities at hands) by a simple mock script and to tell the generation to use the mock script instead of the original script.
- **Testing the application:** Despite all tests, ultimately also the application with all services and all required devices in place must be tested. This is supported by the Platform Evaluation and Testing Environment (cf. Section 7.5).

It is important to mention that only testing on application-level may include all resources and service implementations in the final form as it will be deployed. Thus, accidental overlaps of resources, e.g., identity stores may only be detected when running an integrated application.

The requirements documents [SSE21, ESA+21] even demand in-place pre-deployment tests. Currently, the platform does not offer functionality for these optional (but important) requirements.

4 Architectural Decisions and 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 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 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. **Asset Administration Shells** are the primary mechanism for the **communication among platform components**, in particular for remote method calls. AAS shall also be used as unified backend for the platform management user interface where ever possible. Notable exceptions are monitoring / tracing streams (via the transport protocol/component) [CE21], soft-realtime transport streams (via the transport protocol/component) or operation status updates to be displayed on the management user interface (currently not possible due to performance and resource problems of the BaSyx AAS implementation).

C4. 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.

C5. **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.

C6. **Protocol brokers/servers** are internal and shall not be used for communication with external components. On the one side, these broker/server instances host the internal data traffic, i.e., additional channels (accidentally) using the same names may disturb platform operations or even overload transport capacities. On the other side, the internal protocol/broker/server may change due to some configuration decision, also as part of system evolution, and, thus, may break assumed communication with external components. With the advent of new security mechanisms, e.g., an internal transparent overlay network provided by KIPROTECT Hyper[89], the communication may be internally authenticated and encrypted and, thus, not also be available for external communications.

C7. **Protocol brokers/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). These testing servers may be used during platform instantiation to provide a broker/server for a selected protocol.

[89] https://github.com/kiprotect/hyper

C8. Production code must **not have dependencies to alternative or optional components**. As a rule of thumb, generic components without "suffix" names (representing the generic part of a component) shall not directly access related optional/alternative components indicated by "suffix" names, e.g., `transport` is the generic transport layer while `transport.amp` the alternative for the AMPQ protocol. This applies to the support layer (no access to BaSyx/Server), the transport binders (e.g., `transport.spring` vs. `transport.spring.amqp`), the connectors, the service environment (`services.environment` vs. `services.environment.spring`), the services (service manager `services` vs. `services.spring`) the ECS-Runtime (e.g., `ecsRuntime` vs. `ecsRuntime.docker`) etc. In contrast to production code, test code (Maven scope "test") may declare dependencies to specific alternatives to allow for functional testing, e.g., to rely explicitly on the AMPQ transport protocol. Although alternatives shall be tested equally on their level, it is also clear that component testing with specific alternatives just shows the functionality for the assumed/selected alternatives.

C9. **Generated artifacts** shall be separated from manual code (usually an own top-level folder such as `gen`) and generated artifacts shall not be modified as they may/will be re-generated upon request.

C10. **Implementation of services** shall be separated per service, so that services can be composed/integrated free of other dependencies. For convenience, in testing code, we may intentionally invalidate this rule, e.g., `test.configuration.configuration` implements all service artifacts for all tests in `configuration.easy`.

C11. **Exception handling** is often not considered a topic for architectural constraints. However, the basic decisions on how and where to use/handle exceptions are important as they enforce certain responsibilities. Moreover, some architecture modeling languages like UML allow for the specification of exceptions. Exceptions indicate abnormal situations in the program execution that shall not be handled by normal program code rather than by stopping the execution at the point of occurrence and tracing back the method calls until the exception at hands is handled (or on top-level it terminates the program or the actual thread). While often programmers try to handle an exception at the point where it obviously occurs (in Java, where a checked exception could be thrown that must be handled), we believe that in most cases the caller, i.e., the cause of executing the code that throws the exception shall be informed, which does not mean that each exception must be transported and handled in top-level code. For example, consider some complex data format processing code, e.g., reading an AASX file for an asset administration shell. If we handle an Input-Output exception in that code, the caller does not know that and why the format processing fails. Let us now assume, that reading the AASX file was triggered by the ECS runtime when building the AAS of the ECS runtime, e.g., to link device vendor and ECS AAS. Here, the lifecycle handler of the ECS runtime (more or less top-level code) that starts the creation of the ECS AAS is not interested in why an AASX file processing fails. However, the code creating the AAS trying to establish the AAS link is better suited to handle the exception, e.g., to insert an empty link or to log the problem. Thereby, logging (cf. Sections 2 and 7.1) is often not the right answer to an exception, in particular not emitting an exception stack trace to the console (which may not be logged properly). In contrast, the programmer shall think about handling the exception in a manner that processing can succeed, e.g., inserting an empty link into the AAS rather than no AAS property at all, which may cause failures in other parts of the system relying on the assumption that such a property exists. In particular the type of the used exceptions shall be selected carefully (cf. Sections 7.1).

C12. Apply **defensive logging**, i.e., carefully think about what is an "error", a "warning", an "information". Errors shall only be emitted if a component will fail to operate. If the

component can compensate this, e.g., by falling back to some strategy or default plugin, then a warning is more adequate.

C13. **Logging setup/filtering is decided during integration, not before.** As some "bigger" components like BaSyx, Spring or even Apache QPID-J ship with their own ideas how to set up and configure logging, hiding the actual platform logging through the logging support interface and implementing plugins as well as deferring the logging decision (SLF4J, LogBack, Apache, etc.) and the setup what to log as long as possible. Thus, all components must use the platform logging support and not define a concrete logging implementation in their production code dependencies, only in their test dependencies, preferably using the platform logging implementation plugin. Ultimately, the code generation that is performing the integration automatically must know whether further dependencies are needed, existing dependencies can be used and how the logging of the components at hands shall be set up.

C14. Basic libraries like **YAML and JSON shall not be used directly**, only via the platform support interfaces, akin to logging.

C15. **Interfaces, signatures and behaviors**, in particular of services and connectors, must comply with the design of the respective components so that the code generation can properly take them up. In many cases, arbitrary changes or customizations cannot be easily realized as they must conform with all existing components and may require changes to the code generation. E.g., constructor signatures shall declare the same parameters in the same sequence as the (abstract) superclasses. Likewise, template must comply with the given template parameters. However, in individual cases, template parameters of connectors may be extended (new parameters added to the end) or an abstract connector class can be instantiated by generating additional, required methods. However, typically constructor calls cannot easily be modified, if at all, also by adding parameters, for which the values must be obtainable from the platform configuration model. Akin, the behavior of new connectors and services must comply with the expected behavior of implanting or overriding methods as documented, e.g., in code.

C16. **Dependencies** of all kind including Java, Python or artifacts like AI models or resources are managed through Maven and, thus, must either be available through official or own/local/private Maven repositories or packaged during the platform/application build processes into Maven artifacts, e.g., using Maven assembly descriptors. Application templates include generated maven build processes with respective assembly descriptors, e.g., for Python services or specified (AI) artifacts. Due to the plugin concept and the isolated class loading introduced in version 0.8.0, managed Maven dependencies are not allowed anymore as they pollute the effective dependencies (thus, the basic parent POM `platformDependencies` does not contain any dependencies). Thus, platform core layer components like support or `transport` are not allowed to declare any external dependency. All implementing components/plugins (e.g., `support.aas.basyx` or `transport.amqp`) may declare dependencies locally, but are encouraged to rely on the managed dependencies of the platform's bill-of-material POM (`platformDependenciesBOM`). In particular, `platformDependenciesBOM` aims for commonly used versions of dependencies as well as for reducing the installation footprint.

C17. **Spring dependencies** are considered optional so that implementation components can rely independently on individual spring versions (although this may require packaging such components as platform plugin, i.e., loading it into an own classloader). The common spring dependencies for platform implementation components/plugins are in `platformDependenciesSpring`, an extension of `platformDependenciesBOM`. However, plugins may rely on more specific/recent versions of Spring, e.g., BaSyx2, and, thus are allowed to declare own dependencies or to override managed 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 oktoflow 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. However, it is not feasible for the work on the platform to wait until such standards are defined. Primarly, we define pragmatic AAS submodels as need and adjust them to standardized formats as soon as possible and feasible, e.g., until version 0.7.0 we integrated up

- Generic Frame for Technical Data for Industrial Equipment in Manufacturing [IDTA 02003-1-2]
- Handover Documentation [IDTA 02004-1-2]
- Hierarchical Structures enabling Bills of Material [IDTA 02011-1-0]
- Product Carbon Footprint [IDTA 2023-01-24]
- Time Series Data [IDTA 02008-1-1]
- Submodel for Contact Information [IDTA 02002-1-0]
- Nameplate for Software in Manufacturing [IDTA 02007-1-0]

through automated generation of an IVML specification per submodel format and subsequent code generation. Based on this approach, oktoflow can support out-of-the-box in total 26 IDTA submodel specifications and one specification draft. There will be, for sure, further specifications oktoflow can benefit from. However, upgrading the device representation or the technical data submodel requires a synchronization among the work of several components, in particular also the management UI. With this approach in mind, we designed and partially realized the prototypical platform AAS structure shown in Figure 42. As already explained in Section 3.1.2, we separate between AAS describing an (external) artifact and internal information (usually in sub-models). AAS do exist for

- The platform AAS with its various sub-models like (legacy) equipment nameplate, software nameplate, dynamic network port assignment, transport setup, (S3) storage access, (available) artifacts such as containers or deployment plans, installed connector/service types and their utilized data types, the device management and available devices (with installed/running containers, installed service artifacts, running services).
- Further assets represented in their own AAS like devices, service or composed applications (with vendor information). Device and service AAS are linked from the respective platform submodels to make the information in the AAS available. For each application running on top of the platform, an AAS shall be provided (currently via the TraceToAasService discussed in Section 3.5.2.1), which states the creator of the App but lists also the utilized services and may provide application specific operations.

For the platform AAS and its sub-models, 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.

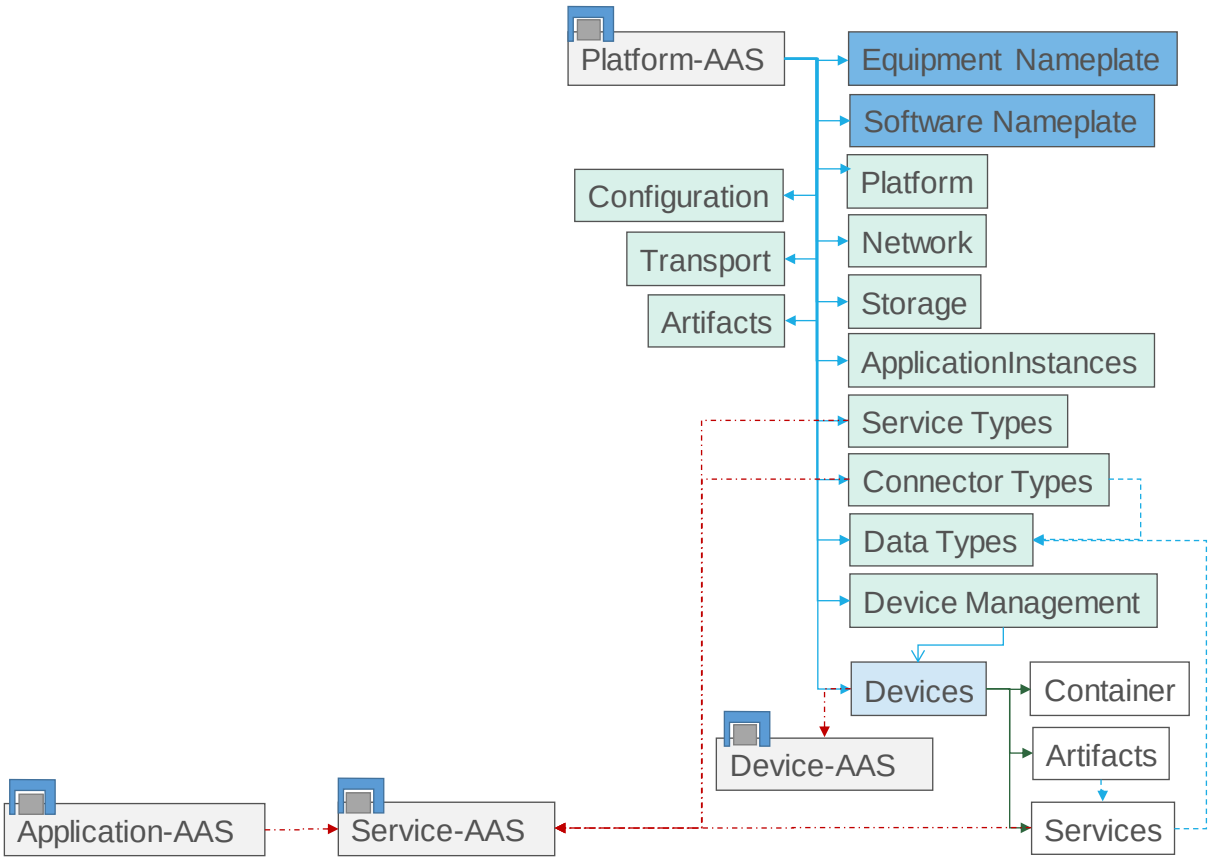

Figure 42: AAS structure of the platform (preliminary, incomplete)

Figure 43 depicts a screenshot illustrating a fragment of the platform AAS in the AASX Package Explorer, i.e., an excerpt of the full AAS shown in Figure 42.

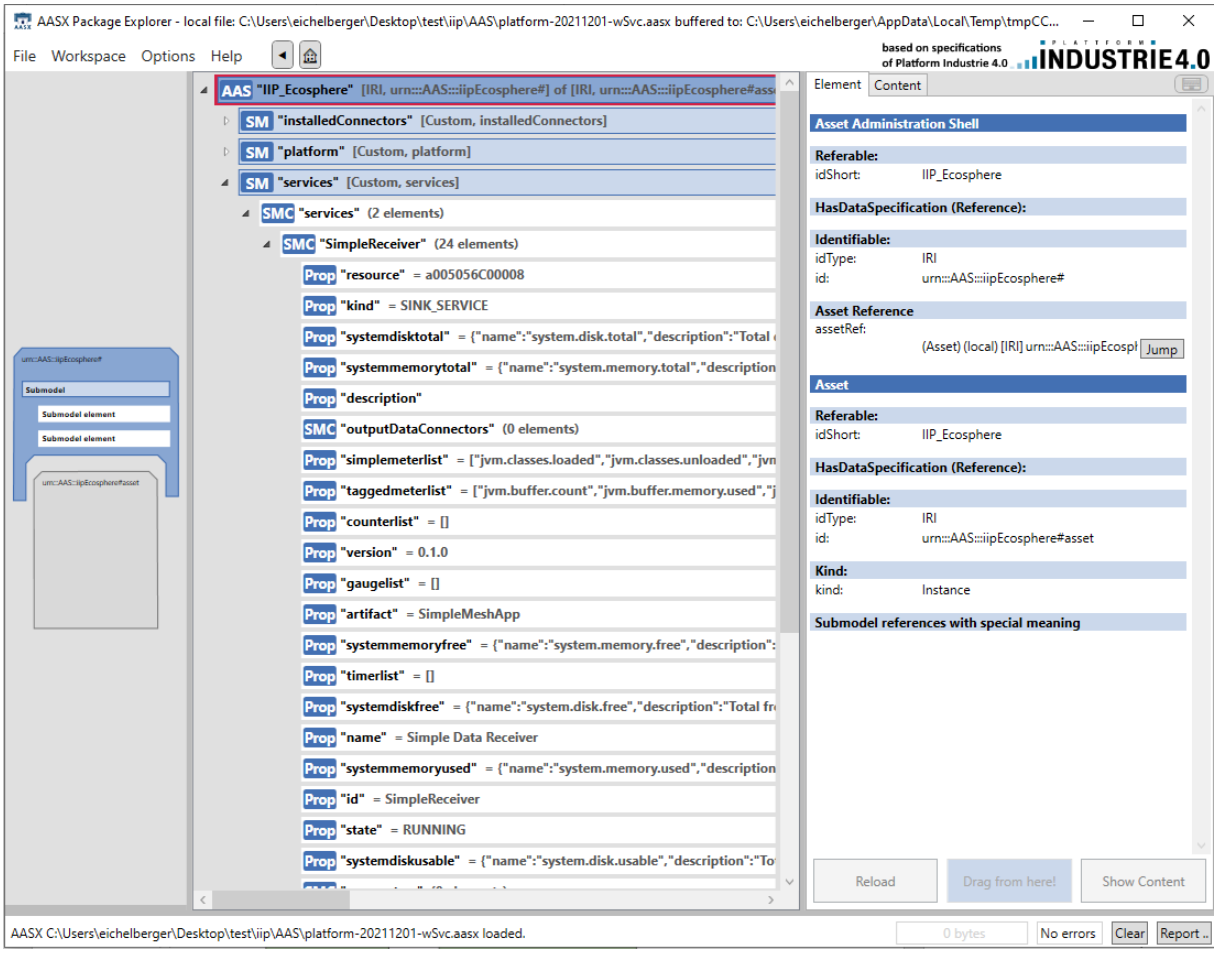

Figure 43: oktoflow AAS in the AASX Package Explorer showing a running service (`SimpleReceiver`).

As the platform 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 3.5.2, 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 3.5.2, 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 (similar to the contents of File Data Elements).
- The platform abstraction appears to be easier to use and requires less code than plain BaSyx [Cas21], but this was a design goal. Moreover, the AAS implementation can be replaced seamlessly, also by a non-AAS interface realization.
- So far (as far as we know), BaSyx does not provide support for resolving references to the referenced element. While this may not be a serious problem when following such links is not crucial, it is an obstacle for platform submodels such as services where we need to reference to a related service, resolve that and access the actual state. This absent functionality drove some of the structure decisions for our sub-models. Similarly, we use URIs to link sub-models and external AAS, in particular that the CLI/Management UI can provide information stated in the AAS.
- When deciding about the concept to realize, in particular AAS vs. sub-model, take the industrial production viewpoint where the AAS concept originates from and try to identify the asset that is described. If the modeling is about an asset (potentially provided by a different organization), typically an AAS is required. When detailing (own) information, often an own AAS is more adequate.

6 Platform Configuration

This section provides an overview on the platform configuration model and the concepts used to model configuration options for the oktoflow platform. We now give an overview of the configuration model, then, from a more pragmatic point of view, an insight into a simple example configuration as this is required for running the platform. Section 6.1 dives deeper into the configuration model by discussing applied modeling patterns, Section 6.2 provides more details on the structure of the configuration meta-model as well as on adjustments for a model managed by the platform, Section 6.3 discusses additional support for standardized protocols or connectors such as OPC UA and Section 6.4 details selected configuration elements as reference for own models. Next, Section 6.4 details the instantiation process. Section 6.6 discusses the container instantiation. We then turn to the development of applications based on the configuration model. Section 6.7 outlines some example applications shipped with the platform. Then, Section 6.8 discusses the steps needed to create an application with the configuration and the instantiation process, Section 6.9 illustrates typical implementation project structures, and Section 6.10 illustrates default build sequences and their build commands. Finally, Section 6.11 summarizes service implementation considerations.

In essence, the configuration model mirrors the component hierarchy of the platform and describes per component the configurable elements, their dependencies and constraints. IVML is the Integrated Variability Modeling Language [IVML] as realized by the EASy-Producer toolset [SE15]. 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) [VIL]. In the platform, 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** [Eic16]. These capabilities will be used in the last project year to allow for self-adaptation of the platform.

The configuration model is taken up by the configuration component (Section 3.9) and used for platform instantiation and runtime adaptation. The configuration component allows for high-level model operations.

As illustrated in Figure 44, the configuration meta-model reflects the layers and components of the 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 specific concepts in the remaining models. The first 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[90] 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.

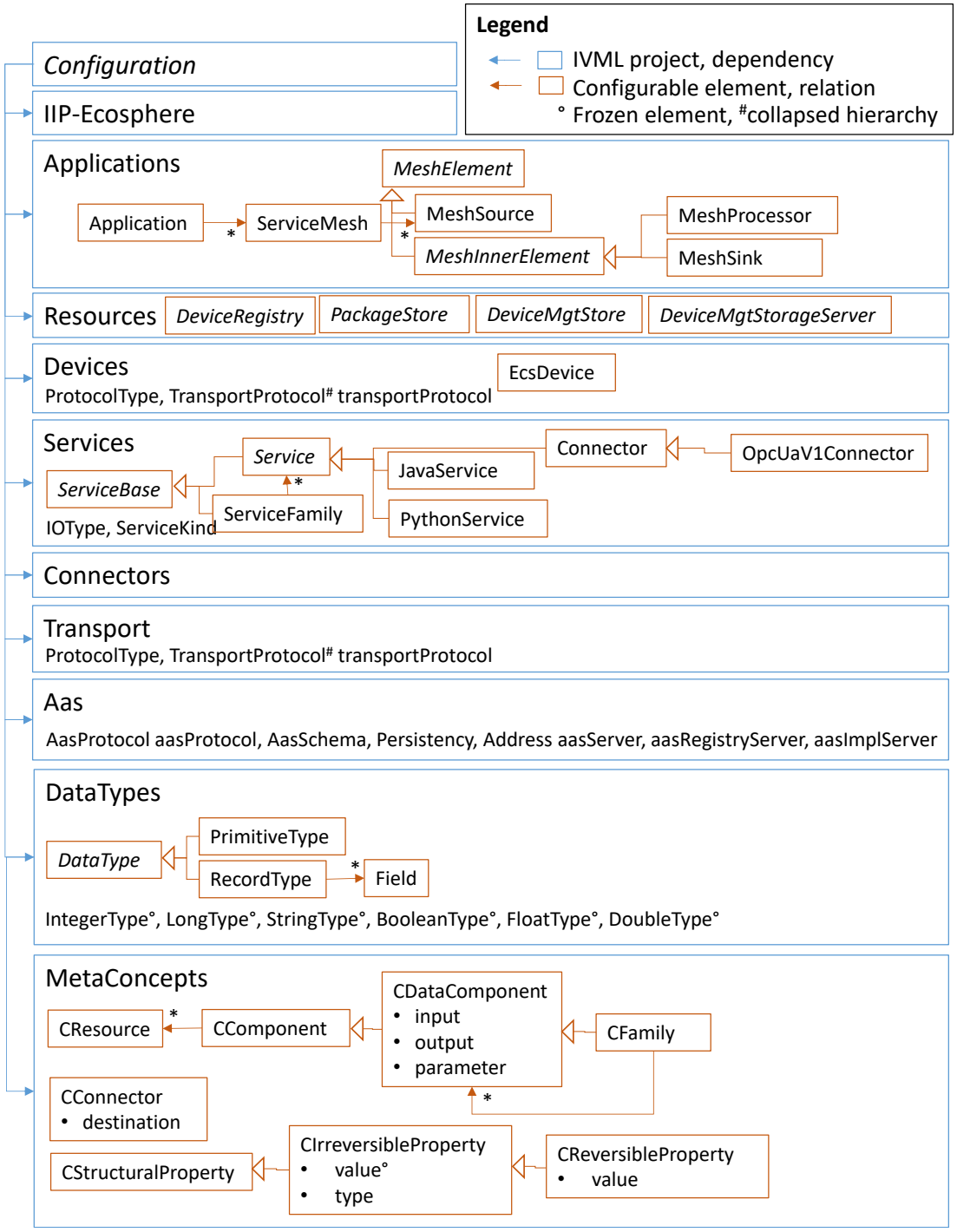

Figure 44: Illustrative structure of the IVML platform meta-model (simplified).

[90] 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.

On the service level, several refinements of the platform's 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 (as introduced in Section 3.1.2) 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`, which is refined to application-specific Java or services as well as platform-supplied services like connectors or pre-integrated services like the KODEX, the RapidMiner RTSA or the `Trace2AAS` service. Services declare their input and output data types, typically for forward or backward data flows (cf. Section 3.1.2). In the model, service properties are pulled up from service level to mesh level during model validation and allow for checking whether a flow 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.

For building up AAS, e.g., to trace platform operations or to represent services, the configuration model also reflects basic vendor information that is required to instantiate respective AAS. This information can optionally be attached to an application or a service. Moreover, nameplate information items can be reused to increase consistency, e.g., if a "vendor" created multiple services.

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 3.11, 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 45: Technical and service part of a simple platform configuration.

Figure 45 depicts the first part of a simple platform configuration used for testing[91]. 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 of the 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)
};
```

[91] We illustrate the example as a single IVML project file. A managed platform configuration separates the parts into technical setup, (all) types, (all services) and individual files per application and mesh, each with own freeze blocks, and ties parts and pieces into a single platform configuration through imports. Essentially, a managed platform configuration is semantically the same as a single file but easier to handle for the platform.

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

Figure 46: Application and service mesh part of a simple platform configuration.

The second part of the example in Figure 46 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 47: 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 47 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 47 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 configuration model creates more than 14 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, Linux/system service descriptors, README files, Broker setup specifications, Docker files, Type script files, Angular environment setups),

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.

Besides a brief explanation of a configuration model, it is probably relevant to the reader to have executable examples or tutorials at hands. We will cover this topic in Section 6.6.

6.1 Modeling Patterns

As shown in Figure 44, the platform configuration model consists of several layers reflecting the architectural layers of the platform. Each configuration model layer defines the decisions to be made, typically either a) using basic IVML types b) refined compound types if the alternatives have detailing properties or c) a more detailed structure of own types to model service and app decisions. This section dives a bit deeper into the IVML platform configuration model.

```
decision = X::Alternative2;
```

```
enum X {Alternative1, Alternative2, …};

X decision = X::Alternative1;
```

Figure 48: IVML model pattern for simple alternatives without detailing properties.

Figure 48 shows the IVML model pattern to represent simple alternatives that do not need to be detailed further, e.g., the transport layer serializer format. The lower box in Figure 48 illustrates the model layer, the upper box the specific platform configuration. The alternatives are modeled (in the lower box) as the enumeration type *X* listing all potential alternatives. The declaring model layer also defines a variable representing the respective *decision* and assigns a default value to ease creating a configuration. The configuration (upper box) overwrites the value to indicate that a different alternative shall actually be included into the platform instance. It is important to note that this pattern does not allow for openness as IVML enums are fixed and cannot be extended later, e.g. in importing IVML modules.

```
decision = Alternative2 {
  // assign properties as needed
};
```

```
abstract compound X {
  Type p = default;
}

compound Alternative1 refines X {
  //optional further properties, constraints
}

// further alternatives, constraints

X decision = Alternative1 {
  p = default1
  // assign further properties
};
```

Figure 49: IVML model pattern for alternatives with detailing properties.

Many alternatives demand further information when selected, e.g., the transport protocol, the S3 storage client/server or the AAS client/server settings. In this case, we model alternatives as IVML

compounds, an abstract base compound defining a common type for all alternatives and refining compounds for the individual alternatives. The base type, in Figure 49 the compound *X*, defines properties that are common for all alternatives, e.g., a server port, usually with default values, while the individual alternatives such as *Alternative1* may add further properties. Each type representing a specific alternative can define constraints that become active only if that specific alternative/type is used. The alternatives may override the default values by re-declaring the properties with the same name/type. As in the first pattern, the declaring model layer defines a variable representing the respective decision, assigns a default instance of *Alternative1* including specific values for the properties. The configuration in the upper box may then assign a more specific value, here *Alternative1* and properties. Please note that property values can be derived from, e.g., common global variables to increase consistency. Moreover, modeling alternatives via compound refinements allows for openness, as further refining alternatives can be defined on any upstream model level, i.e., this IVML model pattern is appropriate for alternative components and plugins contributed by the user. This form of openness must be adequately taken into account in the instantiation process.

In several situations, the configuration model must remain open for extensions by the user or by third parties (supporting the idea of a service store in [SSE21]). Typical extensions are service or application definitions as well as device types (acting as templates for device-specific container instantiations). Akin to the configuration model, also the instantiation process must be extensible, e.g., to perform service-specific generations when an externally provided generic service shall be integrated. The EASy-Producer languages IVML, VIL and VTL offer mechanisms to support such requirements. One basic mechanism is dynamic dispatch operations, for user-defined constraints as well as for generation functions. In dynamic dispatch, the actual operation to be executed is dynamically determined based on the actual types of all parameters (an extension of first-parameter polymorphic execution in object-oriented programming languages like Java) [EQS+16]. Basically, dynamic dispatch allows to consider later extensions of the actual model by refinement and, thus, provides a basic form of openness to refined, but yet unknown types. A second mechanism allows for a dynamic model structure through wildcard imports (similar to wildcard imports in Java), i.e., model parts that are not known at modeling time can dynamically be added to the model structure. However, imports just make model elements known to other model elements and do not influence the dynamic dispatch mechanism. Thus, the EASy-Producer languages provide a special import statement, that allows to extend the dynamic dispatch in the model at hand by dynamically loaded models. In essence, externally provided model parts are dynamically loaded into the model structure and can hook themselves into the constraint and generation operation through specialized types defined in these models.

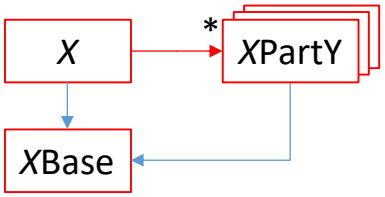

Figure 50: Model structure for openness and extensibility.

For the aforementioned situations, this requires a specific model import structure as shown in Figure 50. Let *X* be the part of the model for services, applications or devices, e.g., *X* could be “Devices”. To enable the kind of openness as explained above, one model module must define the basic types and operations the dynamic extensions shall hook into (module *X*Base, for the devices example the name would be DevicesBase). The extensions are modules that import the base module and add own types and operations. Their names are a composed from the respective model part, i.e., X, the infix “Part”

and the individual name of the extension, e.g., `DevicePartPhoenixContact`[92] for specific device types representing the AXC PLC/edge series of Phoenix Contact. Finally, some modules must use the extensions. Such a module must be an extension of the *X*`Base` and dynamically import the *X*`Part`*Y* in a way that they hook into the dynamic dispatch of *X* (and transitively extend *X*`Base`). In the model, this is the module *X*, in the example called "Devices", which usually is empty except for imports and the model extension statement. This module is then imported by further modules of the configuration model and implicitly introduces the dynamic extensions. At runtime, further extensions can be added, but currently only by re-loading the entire model.

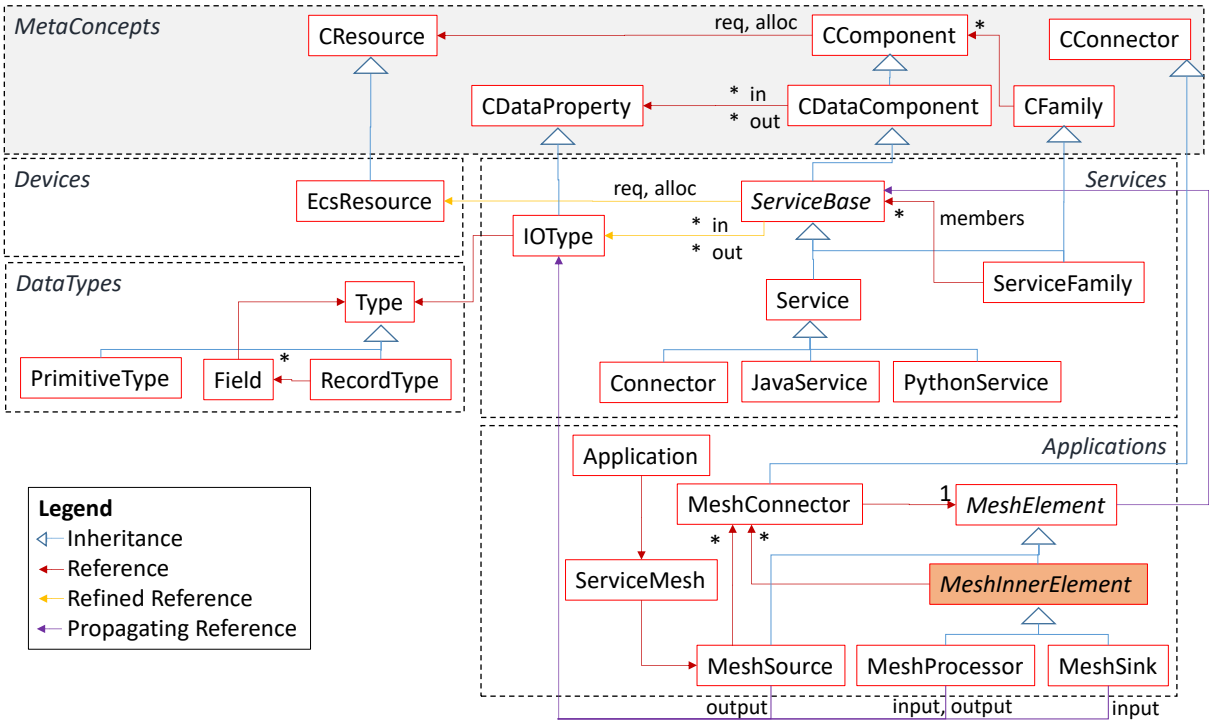

Figure 51: Meta-model concepts for defining services and alternatives.

The configuration of user-defined applications that are executed on top of the platform, the involved services, their data paths and the resources to execute the services on requires more information and, thus, is more complex than the two IVML model patterns discussed before. The most relevant configuration concepts for applications are illustrated in terms of the UML-like class diagram in Figure 51. We target the following aims:

1. Configure **re-usable services** (the *`Services`* module in Figure 51) and their properties, potentially families of semantically equivalent services that can be exchanged at runtime, e.g., alternative AI services. Services include those provided and integrated into the platform as well as user-supplied services.
2. Represent **data transformation and mappings** to reduce the effort of manual coding in standard situations (in the sense of "low code", R19f). Currently, we apply such data transformations in particular to integrate (machine) connectors. Figure 52 illustrates the applied data mapping approach. For a connector we specify a machine- and a platform-side I/O data format, usually a record of named/typed fields. Fields with same (nested) field names are mapped onto each other in both directions, machine-to-platform and platform-to-machine. Fields that cannot be mapped are either ignored, i.e., projected out, or left uninitialized. To fill individual fields, assignment expressions between both sides can be stated,

[92] Depending on the naming, the prefix may be adjusted deliberately for a more "speaking" name.

allowing for simple data transformations, e.g., unit calculations. For a model connector (cf. Section 3.4.2), the given data formats can be turned by the generator into model paths and data can be obtained and transformed automatically. For a channel connector (cf. Section 3.4.2), the input is always binary. Here, the input parsers from the Connectors component can be specified in the model to turn the data into named/indexed fields that are mapped by default to the machine formats and, thus, can further be used as for the model connectors. In the opposite direction, output formatters can be applied. If no platform-supplied parsers/formatters fit to the data at hands, own Java components supplied as Maven component can be specified. Similarly, the entire mapping process can be bypassed by own serializers or model adapters.

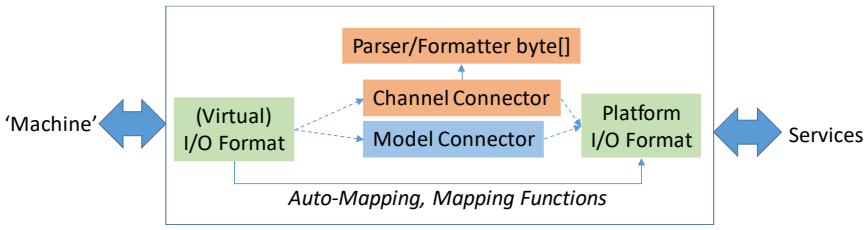

Figure 52: Overview of low-code data mapping for connectors.

3. Configure **physical and logical compute resources** the services are executed on (the *Devices* module in Figure 51), although it is important to emphasize that configuration instances of such resources shall be created and reflected into the configuration by the device management at runtime.
4. **Compose connectors and services to applications** (the *Applications* module in Figure 51) so that one service can occur in multiple applications and that the data paths within an application are defined and can be instantiated automatically.

In more details, the most basic module in Figure 51 is *MetaConcepts*. Although concepts and properties defined in this module could also be introduced in the dependend modules shown in Figure 51, the aim of *MetaConcepts* is to represent generic concepts of configurable runtime-adaptable systems. Thus, *MetaConcepts* introduces basic notions of resesources (`CResource`), components (`CComponent`), families of components (`CFamily`) and connectors among components (`CConnector`). As these concepts define properties using these types (and the connectors even of the top-most IVML type `Any`, therefore no associations in Figure 51), which must be re-defined in upstream modules, e.g., in the *Services* module.

From the generic *MetaConcepts* perspective, we now turn to the specific configuration concepts. The *Services* module currently just introduces the notion of a device, where additional properties will be added in the future. The *DataTypes* module introduces the ability to express types that are reflected in current programming languages, such as “primitive” types like String or Integer, but also more complex, composed types (called `RecordType`). These types are used in the *Services* module to specify the inputs and outputs of individual services. In this module, specific service types (`Connector` for the connectors in Section 3.4.2, `JavaService` for services implemented in Java and `PythonService` for services implemented in Python) are defined and also service implementations shipped with the platform (AAS, OPC UA and MQTT connectors from Section 3.4.2) are defined as configuration instances. Akin to *MetaConcepts* a refined type for user-defined service families that can act on behalf of a fixed, individual service is introduced.

On top of these configuration layers, the *Applications* module defines graphs of services, called service meshes. An `Application` consists of one or multiple `ServiceMesh` instances, and, in turn, a

service mesh starts at one or multiple sources (of type `MeshSource`). Sources are linked via `MeshConnector` instances to processor or, ultimately, sink nodes. In contrast to the IVML model used in the FP7 QualiMaster project [EQS+16], we cannot restrict inner nodes to processors and sinks, as processors may have backward flows to control machines via connectors.

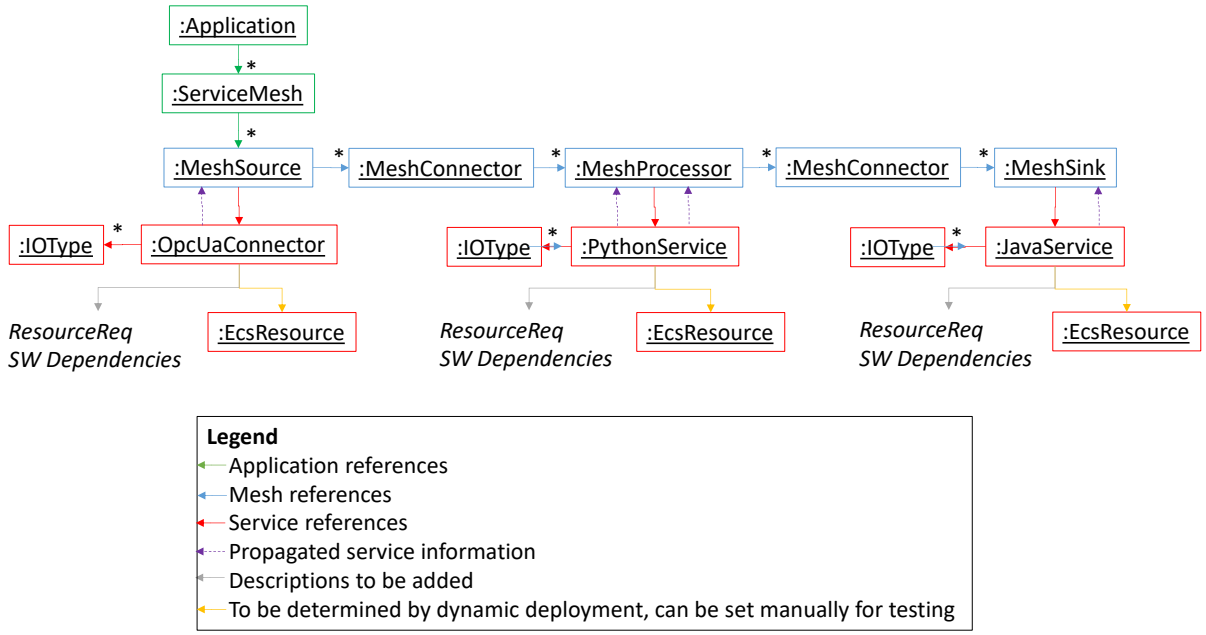

Figure 53: Instance view on a platform application.

As illustration of these concepts, Figure 53 shows how instances of the aforementioned types can be linked together (backward flows are not shown in Figure 53). The `Application` consists of one `ServiceMesh`, which, in turn, consists of a chain of three services, a source, a processor and a sink, all linked by instances of MeshConnector. The source is implemented by an OPC UA connector, the processor by some Python implementation, e.g., an AI algorithm, the sink by some Java implementation, e.g., a database. Each of these services has its own input/output types, which must comply with the predecessor/successor services along the graph constituted by the service mesh. Further, each service is (at latest at startup time of the application) deployed to a certain resource, e.g., an edge device. To determine adequate resource candidates, currently descriptions of resource requirements and software dependencies to be installed into hosting containers are developed.

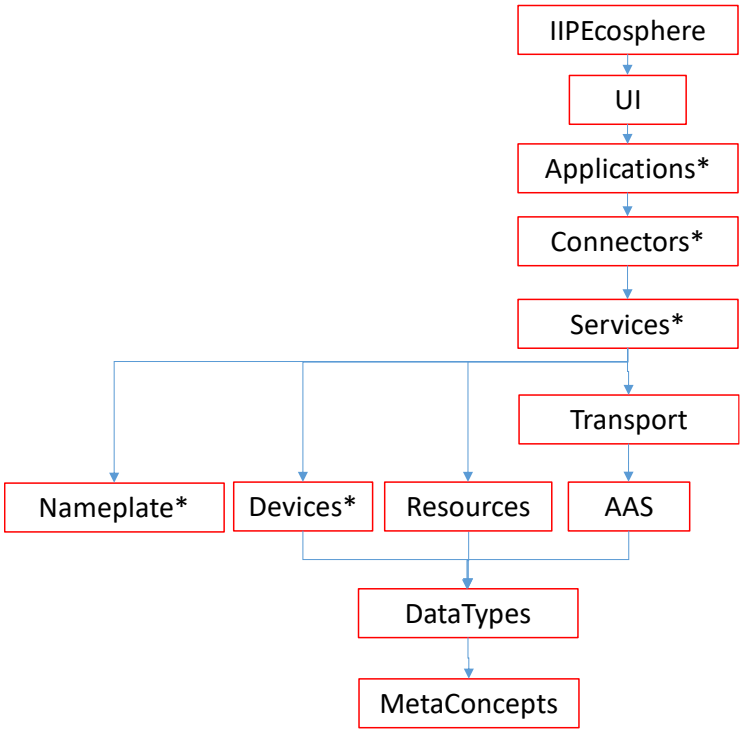

Figure 54: Structure of the IVML configuration meta-model (transitive dependencies omitted).

6.2 Configuration Model Structure

The structure of the configuration meta-model (Figure 54) follows the overall structure of the platform architecture with some additions and a different layering/dependency approach due to pragmatic reasons. The individual modules define the configurable properties and types for the respective architecture layer, component or aspect. The bottommost module `MetaConcepts` stems from an attempt to capture the basics of an adaptive software system and is included here for evaluation purposes. The `DataType` module defines primitive and extensible data types used for specifying input/output types of services and connectors. On the next level, AAS server/implementation properties, AAS `Nameplates` with default instances, `Devices` including capabilities and requirements, and `Resources` (including resource management and monitoring) are defined. Then, the actual transport protocol and its setup (including authentication and transport layer security) are specified. `Services` (including Java, Python services), reuse all these configuration types and form a basis for the more specific `Connectors`. Applications and their service meshes are defined on top, imported by `UI` (settings) and finally the top-level `IIPEcosphere` module for global technical installation settings such as installation paths. Module names suffixed with a * are extensible through the import mechanism explained in Section 6.1.

An actual copy of the meta-model is included in each implementation/example project after downloading the model through Maven. In these projects, the actual configuration is typically defined in one or two modules to focus on the specific properties of the example and to ease gaining an overview.

When the platform takes over control of its own model, fixed structures are needed so that the configuration component can store the configured settings, service instances and application meshes. This form of models is currently in testing and will be enabled as soon as the management user interface provides respective editor functionality, in particular a graph-based service mesh editor based on the information provided by the configuration AAS. The model structure required then is depicted in Figure 53.

In this setup, the topmost module is the `PlatformConfiguration` storing settings that override global non-frozen configuration options. Service instances are stored in `AllServices`, related type definitions for input/output specifications in `AllTypes`. In turn, `AllTypes` relies on `AllConstants`, containing e.g. server host names or commonly used port numbers. The application instances (`ApplicationPart`*X* with *X* being the application name) are stored in individual extensions pointing to their linked service mesh parts (`ServiceMeshPart`*X* with *X* being the mesh name). The top-level meta-model module IIPEcosphere and transitively imported modules are linked through `AllConstants` into the managed configuration model. Further, `TechnicalSetup` defines the fundamental technical abilities of the platform, e.g., transport protocol, monitoring or management UI setup. In turn, TechnicalSetup may rely on the constants in `AllConstants`.

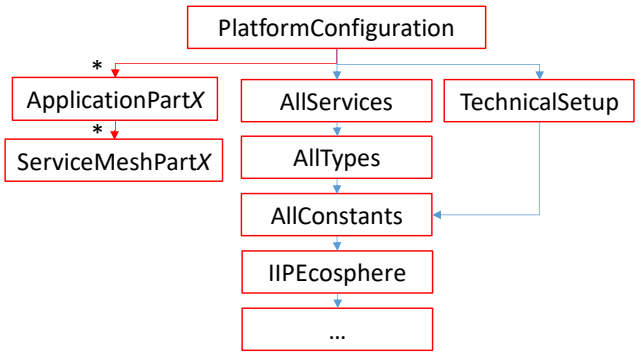

Figure 55: Managed model structure.

The configuration (meta-)model is used as a consistent data basis for the management user interface. Technically, the configuration model is transformed into an AAS, which imposes some limitations:

- IVML variables shall not be named `value`, as this idShort is a reserved name in AAS/BaSyx. To reflect such variables in the same style as any other variables, we automatically adjust the name by prefixing it. As this name is the basic for naming UI elements, instead of `value`, e.g., `avalue` may be displayed, which could be confusing for the user. For this purpose, IVML variables can be annotated with a `displayName`, overriding the displayed name on the UI. Implicitly, we resolve a field misnomer[93] for some version fields that are named `ver`, by creating a display name version for variables of type `OktoVersion` with name `ver`.
- IVML variables of type Any in the MetaConcepts model are intended to be redefined with a specialized type. However, if this does not happen, we assume them to be superfluous and ignore them when creating the configuration AAS, so that they do not occur in the UI.
- IVML variables can be annotated with the integer value `uiGroup`. Elements that shall not be visible shall have the value 0, positive numbers denote mandatory groups in decreasing order of priority, similarly negative numbers associated optional groups.

6.3 Support for Standardized Connectors/Protocols

Defining the input/output data types for complex, nested data structures can be a complex process. For standardized protocols/information models such as OPC UA or AAS, more and more data structures with standardized fields and semantics are created. For a standard-based platform like the platform it is important to take up such approaches and to ease the use of standardized data structures, thus, supporting the platform user in creating applications.

As an example, we turn now to the OPC UA Companion Specifications, a set of standardized models for OPC UA. Currently, more than 50 such models have been specified and further models are in

[93] We will try to resolve that by renaming the respective fields in the model. This may have impact on examples, existing application configurations and the documentation.

preparation. Over time, also the defined model structures are evolving. Thus, manually translating OPC UA Companion Specifications could be an initial approach, which will not turn out to be sustainable. Fortunately, the OPC Companion Specifications are available in a machine-readable XML format, which can automatically be translated into IVML [IVML] and then used further in application configurations. We demonstrate this approach by an automated model translator [Cep23].

The model translator reads an OPC Companion Specification XML file and translates it into IVML using a base meta-model, which extends the configuration meta-model, in particular the `DataTypes` module. The created IVML files, one per companion spec can be imported into an own application model when needed. Besides the main types representing a Companion Specification, also declared subtypes can be used in custom applications, in particular as input/output to generated OPC connectors.

The approach was successfully validated with all 55 available OPC UA companion specs. Valid IVML models were produced for all companion specs and valid connector code was generated. The largest specs, e.g., the Tobacco Machine Communication (TMC)[94] or the IEC61850-7-4 spec triggered a restructuring of the generated connector code to cope with the more than 120 KLOC XML specification. As far as available in the VDW UMATI OPC test server, 3 of the generated connectors were successfully functionally validated against an OPC reference implementation [Cep23].

6.4 Selected Configuration Elements

In this section, we detail core configuration elements as a reference for used-defined applications and as a basis for the next steps towards building an own application. All elements are defined in the configuration meta model and for each variable/field a descriptive text is given in the respective `.text` file[95].

6.5 Platform Instantiation Process

After successfully configuring a platform and the apps to run on the platform, the configuration must be instantiated. This happens through further languages of EASy-Producer [VIL], namely the Variability Instantiation Language (VIL) to express the control over the instantiation process and the Variability Template Language (VTL) to modify or create artifacts of a certain type, e.g., XML or Java code files.

Figure 56 illustrates the steps that are executed during the instantiation. The VIL model defined thee major entry points, which are available through the `PlatformInstantiator` tool, namely

- `generateInterfaces` generates the interfaces of the declared applications as a basis for the realization of user-defined (non-platform provided) services as well as default basic implementations of the service interfaces, e.g., to ease the implementation of parameters and data ingestors.
- `generateAppsNoDeps` instantiats the applications but intentionally leaves out all implementation dependencies. In addition to `generateInterfaces`, this provides also access to implementing base classes. However, the user-defined service implementations (whether they already exist or not) are not considered during the instantiation process.
- `generateApps` for the instantiation of (currently) all defined apps for a platform including all required dependencies. The resulting service artifacts are packaged to be executable, although of course bugs and errors may be located in the application logic.

[94] https://reference.opcfoundation.org/TMC/v200/docs/8.1

[95] EASy-Producer separates the description of modelling elements from the definition of the modelling elements so that multiple languages can be supported. By default, all modelling elements are described in English language.

- `generateBroker` generates an example broker/service instance the configured transport protocol. Typically, we rely on the broker implementations that we use during regression testing. These instances may not be intended for production code, but they are helpful for the first setup or for examples.
- `generatePlatform` for the instantiation of the platform components
- `main` (not shown in Figure 56) which executes all aforementioned entry points, in particular for testing.

One could generate the interfaces for applications in one step with the application code and packaged as part of the same artifact, e.g., using different Maven classifiers. Although this approach is easier to realize, it hinders service reuse and may cause cyclic builds as it seems that Maven does not separate between dependencies of the main and classified artifacts. In contrast, we rely on a separation into an artifact that contains the generated interfaces for all applications defined for a platform and, in individual Maven artifacts, the service implementations. It is important to mention that the service implementations can be organized along applications, but, to facilitate reuse, an artifact can also realize individual services, in extreme a single service. Figure 56 mainly illustrates the separated instantiation approach.

For the instantiation of the **application interfaces**, we first iterate over all data types declared in a platform configuration and create their Java and Python realization (`JavaType`, `PythonType`). Moreover, we create the related serializers based on the declared types, for Java in order to realize the platform transport wire format (here JSON is just one alternative) and for Python a JSON-String Serializer to link a Service into the Python Service Environment. For all services in the platform configuration, we generate the service interfaces (`JavaServiceInterface`, `PythonServiceInterface`) and where feasible a basic implementation for service parameter and ingestor handling (`JavaServiceBaseImpl`)[96]. Please note that, as discussed in Section 3.5.2, Java and Python Service Environments are similar, but also differ in the required level of programming, which is reflected in the different instantiation steps. Thus, Java services require a more complex code generation due to the direct integration into the service execution engine than Python services that are "just" executed by the Python Service Environment, which, in turn, is integrated through Java code into the service execution. At the end of the creation of the application interfaces, we create a Maven assembly descriptor for the Python interfaces, a Maven build specification that creates the deployable artifacts as well as an Ant script to execute the deployment in the continuous integration.

For obtaining the applications (integrating the handcrafted service code), we iterate over all application specifications and all their declared service meshes. Here, we create the code to bind the respective service into Spring Cloud Stream (`JavaSpringCloudStreamMeshElement`) and for further integrations a stub class based on the service interface to transparently integrate non-Java implementations into the mesh. Depending on the actual service, also further artifacts and assembly descriptors may be generated, e.g., as shown in Figure 56 for KIPROTECT KODEX. Furthermore, for each app, the deployment descriptors (if specified in the configuration), a starter class (for registering the mesh elements to the service framework and for registering the serializers), and the Maven POM file (`AppMvn`) are created. The POM file is executed, ultimately creating several artifacts, including one containing the programming interfaces as well as one representing the application-specific service artifact (including dependencies, service code etc.). For deploying the artifacts to Maven in our continuous integration environment, also an ANT file is created.

[96] To allow for service implementations that are not based on the basic implementation, the default values of the service parameters are set after the instantiation of the service through the reconfiguration operation, i.e., the parameter values are not available during the execution of a constructors but shortly after, usually when the first data arrives for processing.

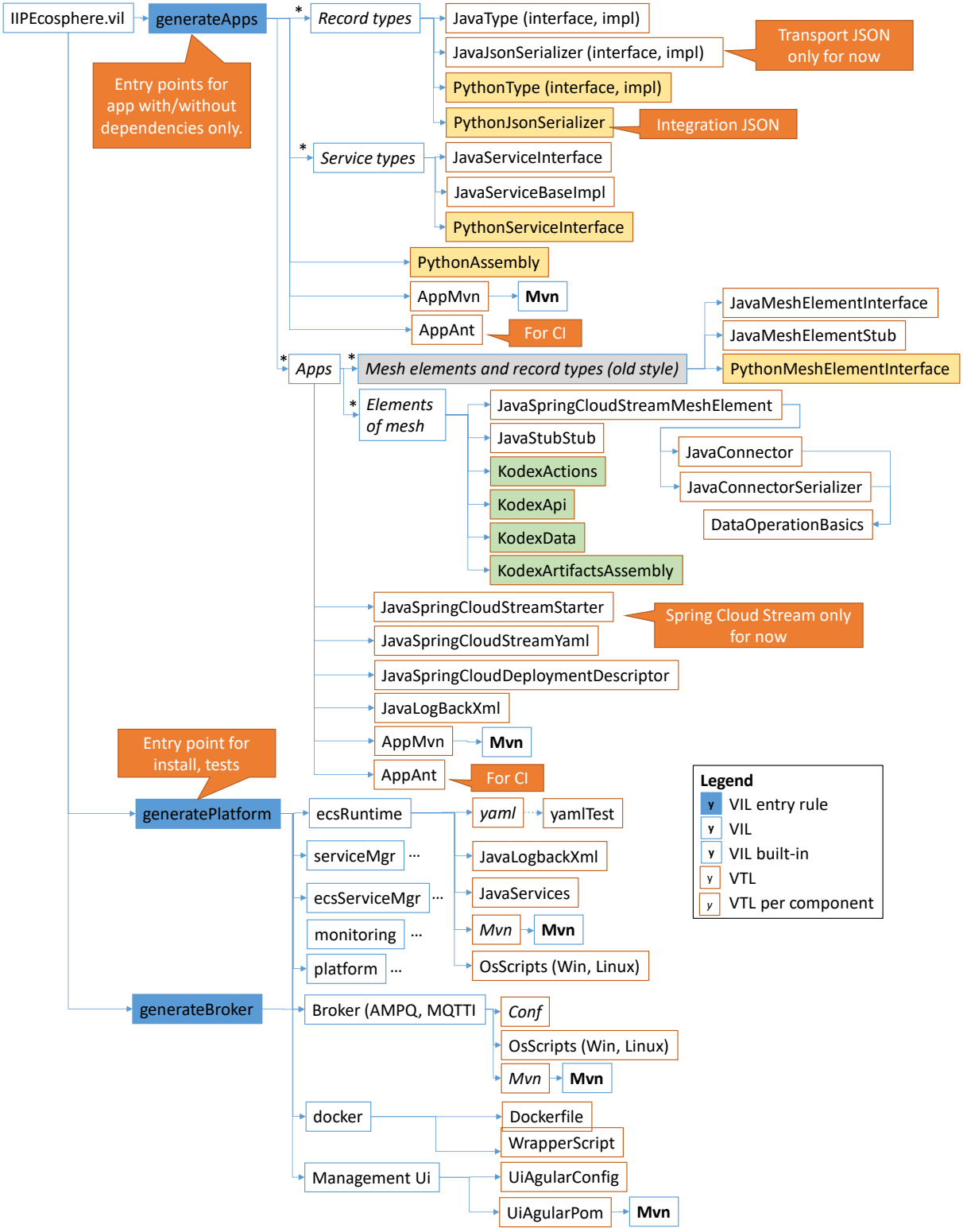

Figure 56: Overview of the platform instantiation process.

On platform level, the instantiation process creates packaged artifacts containing the ECS-Runtime, the service manager, a combined version of ECS-Runtime and service manager as well as the central platform services. For each of these components, first the application setup file (*yaml*) and a test class to validate the Yaml file are created. Then the logging configuration, the selected JSL service descriptors and ultimately a Maven POM with the respective components selected in the platform configuration are created. The Maven POM is executed, ultimately creating the respective artifact, more precisely, folders containing all required dependencies. Moreover, for starting the components,

operating-system specific scripts for Windows and Linux (here also descriptors for automatically starting the components as operating system services) are created. Similarly, the configured transport protocol leads to the instantiation of a corresponding (test) broker, which is also an important prerequisite to generically run examples from an IVML configuration. Further, Docker containers are instantiated based on service, dependency and installation information as well as the download/instantiation of the platform management UI is performed.

The code and artifact instantiation for Spring Cloud Stream currently applies a special strategy for connecting services. This strategy stems from practical experiences, where asynchronous data connections between services did not (always) work out as expected. Already reported Spring Cloud Stream problems may be the root cause here, but we were not able to identify the problem. Thus, we currently route almost all asynchronous connections directly via the transport layer. If we would fully rely on Spring Cloud Stream, a binder plugin would transparently do this for us, which failed in some cases. One particular exception are connections from asynchronous services to synchronous services, which cannot be realized without Spring Cloud Stream mechanisms. We plan to revert to Spring Cloud Stream based on more regression and use case tests in the future.

Specific services such as KODEX, RTSA or the upcoming Flower-based federated learning integration are extensions to the configuration model and the code generation and are represented in their own modules that are dynamically loaded into the platform configuration/code generation.

6.6 Container Instantiation

Virtualization of platform or application functionality intertwines technical requirements with convenience/ease-of-use. On the one side, several kinds of devices, in particular edge devices, do not allow for extensive software installation outside their own ecosystem. In such environments, Java/Python in general or required versions of these languages or their libraries in particular may not be available and an installation may not be permissible. Due to the trend of virtualization in IIoT, in particular for edge devices, but also due to the availability of Docker and reasonable CPU and memory capacities (sometimes even GPU or TPU processors), virtualization becomes a mandatory functionality for the platform (see also [ESA+21]). On the other side, complex services require non-trivial dependencies, which, in particular for Python, typically requires a physical installation of the required packages, which, in turn, depending on the underlying operating system or Linux distribution, may demand the installation of native libraries, the execution of operating system installation procedures or C code compilers. Such operations may not be permitted on a target system, but, more importantly, separating specific dependencies among different (versions) of applications or uninstalling/cleaning up such dependencies without virtualization capabilities may end in a nightmare.

Creating adequate containers for an IIoT application requires technical knowledge and, dependent on the actual (performance) goals, may involve non-trivial tradeoff decisions, e.g., pre-installed vs. dynamically installed dependencies or regarding the layering of the target containers, in other words, what to do first while creating the container in order to maximize with respect to desired system properties (transfer time, upgradability, adaptability). For this purpose, the platform aims at automatically creating the required containers based on a set of basic container creation strategies. As ideally no human is involved in this process, this allows for application and device specific containers (e.g., vendor builds of certain libraries) instead of one-size-fits-it-all containers as often created in other/industrial approaches, typically for virtualizations in cloud settings where certain resources do not matter so much [EPR+22, EPN22].

Container instantiation is the process of creating container images that include an application with all the dependencies that it needs to run an application at hands. We primarily focus on the Docker

container technology (mentioned but not required in [ESA+21]), but allow for extension by other approaches, e.g., LXC. Container instantiation happens during the platform instantiation.

The instantiation process is enabled when the configuration model for an application sets the variable `createContainer` to `true`. Then, based on the technical setup of `ContainerType` in the configuration model, three different types of container images are created. Successfully created container images are stored in the respective container registry as defined in the configuration model. The registry could be a public registry such as DockerHub, but in particular also private, locally installed registries are supported in order to allow for applications that are based on licensed or IPR protected components.

Currently, as discussed in more details in [Sta22], the platform supports the automated creation of six types of container images (based on the configured value for `ContainerType`). The container types are illustrated in Figure 57. These types are

1. `Ecs_Svc_App` to create a container for each application that contains an ECS runtime and a service manager running as separate processes. The container contains the dependencies of all services for the configured application as well as a local communication broker to facilitate local transport-layer data communication among application services.
2. `EcsSvc_App` to create a container for each application that will have an ECS-runtime and a service manager running as one single process. Although it is advisable to run these two central services as own processes to increase resilience, a single process allows for saving memory resources. As for `Ecs_Svc_App`, the container includes the dependencies of all services for the configured application as well as a local communication broker to facilitate local transport-layer data communication among application services.
3. `C1Ecs_C2Svc_App` to create two separate containers for each application: one container will have an ECS-runtime and the second container will have a service manager, the application dependencies and a local broker for local transport-level communication among services.
4. `Ecs_Svc_AllApps` to create a container for all applications that contains an ECS runtime and a service manager running as separate processes. The container contains the dependencies of all services for the configured applications as well as a local communication broker to facilitate local transport-layer data communication among application services.
5. `EcsSvc_AllApps` to create a container for all applications that will have an ECS-runtime and a service manager running as one single process. Although it is advisable to run these two central services as own processes to increase resilience, a single process allows for saving memory resources. As for `Ecs_Svc_AllApps`, the container includes the dependencies of all services for the configured applications as well as a local communication broker to facilitate local transport-layer data communication among application services.
6. `C1Ecs_C2Svc_AllApps` to create two separate containers: one container will have an ECS-runtime and the second container will have a service manager, the dependencies for all applications and a local broker for local transport-level communication among services.

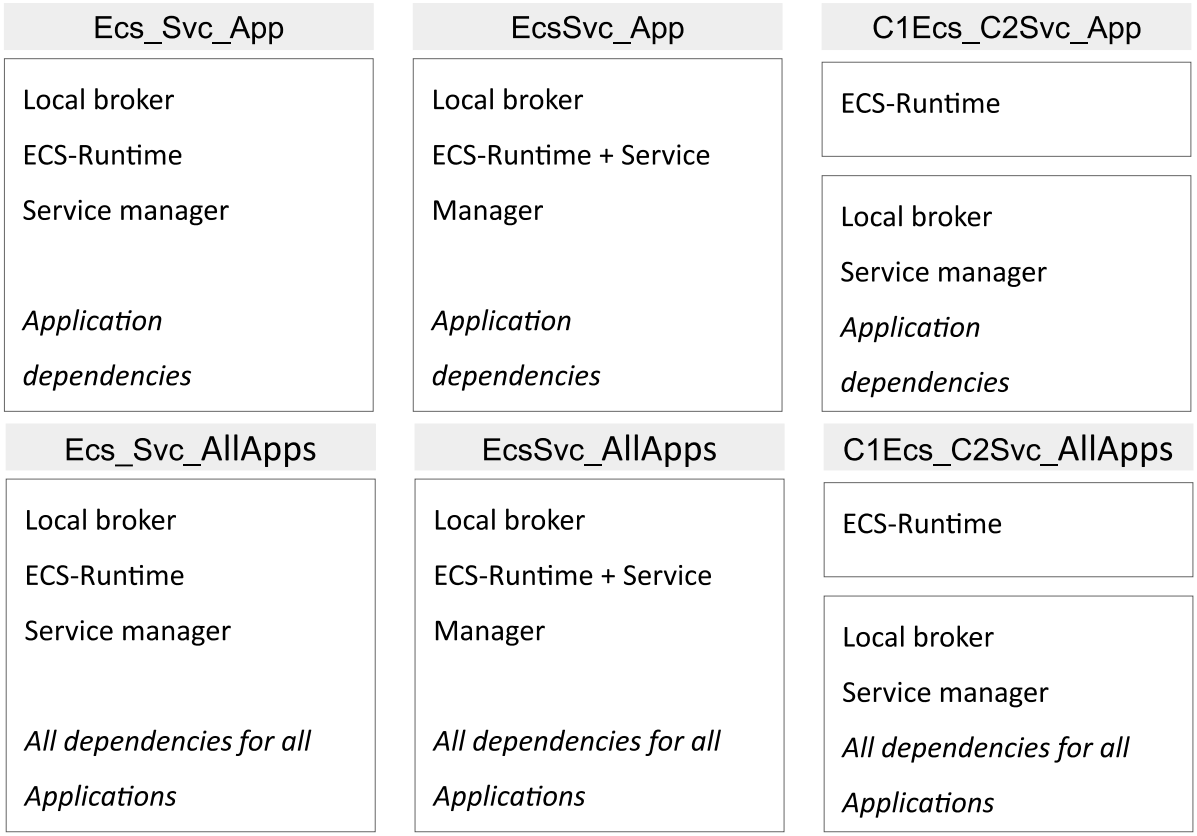

Figure 57: Container types and contained services/parts as supported by the platform.

After creating the images and pushing them to the registry, a descriptor for each Docker image is created specifying properties of the container image. Besides descriptive characteristics of the container, the descriptor in particular contains the container startup options (environment variables, exposed network ports, utilized volumes, Docker-out-of-Docker settings for the `Ecs`, `C1Ecs_C2Svc_App`, or the `C1Ecs_C2Svc_AllApps` container type) and the location from where to obtain the container, e.g., an image name pointing to the local container registry. These descriptors are stored in the artifacts folder of the platform as specified in the configuration model and can be obtained by the management user interface (to display the available containers), but in particular by the container manager installed on a device (e.g., `Ecs` container type) in order to obtain an application/service container on demand and to start using the startup parameters specified in the container descriptor.

During the instantiation process, each container image is created for a specific device as configured in the device type definitions in the platform configuration (each device has its own specifications and configuration). Each container image will have the JAR files created specifically for the device. Additionally, each device may define provided dependencies. In this case, the provided dependencies will not be added to the container. With this, the instantiation process is more flexible and container images are more device-oriented.

During the container image creation, there is an option to use an image-based manner by setting the configuration model variable `containerBaseImageMethod` in the configuration model for an application to be[97] `true`. With this option, the container image creation is done as shown in Figure 58: Container base image creation. The steps are:

[97] Please note that this requires freezing the variable, cf., Section 7.5.

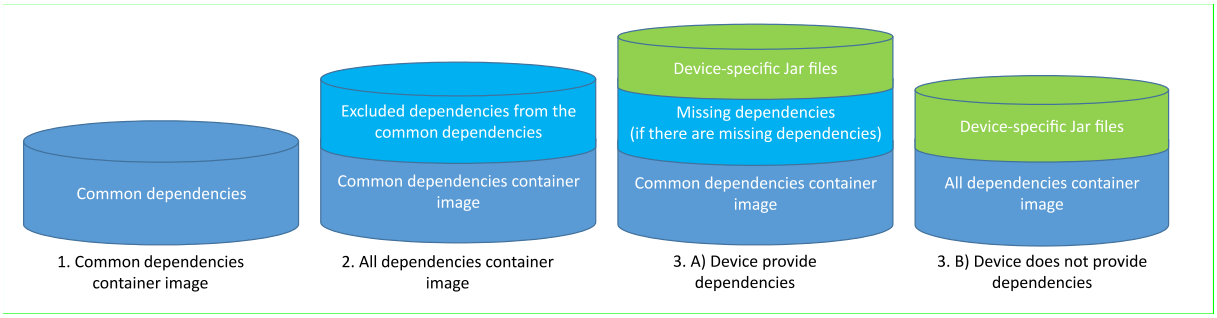

Figure 58: Container base image creation

1. Create a common dependencies container image: This container image has just the dependencies shared among all the devices.
2. Create an all dependencies container image: This container image will use the common dependencies container image as a base image. Then, it will add all the excluded dependencies from the common dependencies container image (this container will have all the dependencies regardless of the device).
3. Application container image (or all applications container image): This container image will use either the common dependencies container image or all dependencies container image as a base image, determined by the dependencies provided by the device.
    a. If the device provides dependencies, then the common dependencies container image will be used as a base image. Then, add the missing dependencies (if there are missing dependencies). Finally, add the device-specific JAR files.
    b. If the device does not provide any dependencies, then the all dependencies container image will be used. Finally, add the device-specific Jar files.

Using the image-based manner will reduce the container creation time and the storage size of the container.

The instantiation process gives the option to generate a container image for the platform and its dependencies. The **platform container image** generation is enabled when the configuration model for an application sets the configuration model variable `platformContainerGeneration` to `true`.

Also, during container image creation, the instantiation process employs **semantic fingerprinting** for all the files used in containers to identify if the relevant content has changed. MD5 checksums for the files used during the last build of the containers stored as fingerprints. Relevant files may be source files including Python dependency lists, binary files, or archive files (based on their contained files). The instantiation process uses those fingerprints to avoid unnecessary re-building of the containers if there are no changes, even if the timestamp in Java JAR files or installations of Python libraries have changed. The semantic fingerprinting can be skipped by setting the configuration model variable `forceContainersCreation` to `true` (default is `false`).

If individual Python services have a **conda environment** defined (cf. Section **Error! Reference source not found.**), the container instantiation will install the related Python dependencies into that conda environment instead of performing a global dependency installation. In addition to separating Python services into different packages (cf. Section 3.5.2), this allows for an even deeper separation of potentially conflicting Python dependencies, e.g., if different services require the same library in different versions and for certain reason there is no common library version. However, conda environments also have an impact on the resource usage of a container as there is no sharing amoung conda enviroments, i.e., all required dependencies must be installed multiple times. This could only be avoided through service-specific containers, which is currently not in scope of the container creation of the platform.

6.7 Example Applications

For diving into the platform concepts and operations from a practical perspective, examples are desirable. One source for such examples are the configuration model regression tests, which contains two simple applications (`SimpleMesh` and `SimpleMesh3`) as well as a more complex example (`RoutingTest`). However, due to their very nature, these test cases are rather artificial and use a single implementation project for the realization of their services to ease the overall build process. While this is acceptable for regression tests, it may not be an adequate starting point to better understand the platform.

To make the platform accessible and understandable, we started to collect some examples that can be published without breaking IPR. On the one side, this set of examples is still small and in development (to cope with unavoidable platform changes). We aim at providing more examples and accompanying explanation (such as slides or videos) in the future. On the other side, the platform example set is one extension point, where you can easily hook into and contribute own examples (also discussing alternative approaches, e.g., to code organization).

It is important to mention that the examples are meant to be created/completed by the platform instantiation. In other words, when you import and open one of the examples the first time in an Integrated Development Environment or just by executing Maven, **you may run into compile errors or missing dependencies**. These are (usually) not bugs, rather than parts that were intentionally not committed and that must be created using the platform instantiation process, i.e., some of the command interaction points of the `PlatformInstantiator`.

We briefly summarize the individual examples below. The examples can be obtained from GitHub[98]. Each example includes a `README.md` file with more detailed explanations. Typically, a Linux build script is included, which is used for regression testing of the platform. The example set consists of

- **examples.python:** This example focuses on the integration of an asynchronous service realized in Python (through the Python service environment) and demonstrates the integration with accompanying Java services (a simple data source as well as a simple asynchronous data sink). Please recall, the Python service implementations must be based on (generated) platform interfaces, located in a specific module (`services`) and packaged into an own Maven artifact (type “python”) to become available to the platform instantiation. Similarly, the Java services for sources and sinks must realize respective interfaces and be packaged. However, as in the VDW example, neither the actual configuration meta-model nor the required interfaces are available after download but must be obtained/created through instantiation steps. As the services are linked in a stream-based manner, also a broker must be available. For this purpose, using the command (`mvn -U install`) you obtain the configuration meta-model, instantiate the broker based on the example configuration, create the application interfaces and basic implementation, compile the example, and finally integrate the service code in the example with the app. The application can be executed through a running platform, but also standalone, which requires a specific setup that is included in the example.
- **examples.pythonSync:** This example follows the same setup as `examples.python`, but integrates the Python service code into a synchronous service passing its data on to a synchronous sink. The remaining properties of the example are the same.
- **examples.rtsa:** This example shows how to utilize the RapidMiner RTSA as service in a platform application. Regarding the setup, the example is rather similar to `examples.python`, except for the use of RTSA instead of a manually implemented Python service. One specific aspect is that RTSA is commercial software and not included in the example. As alternative, we use our

[98] https://github.com/iip-ecosphere/platform/tree/main/platform/examples

simple FakeRTSA from the regression tests. As above, using the command (`mvn -U install`) you obtain the configuration meta-model, instantiate the broker based on the example configuration, create the application interfaces and basic implementation, compile the example, and finally integrate the service code in the example with the app. The application can be executed through a running platform, but also standalone, which requires a specific setup that is included in the example.

- **examples.KODEX:** This example illustrates how to use KIPROTECT Kodex as anonymization/pseudonymization service in an application. Again, the example is rather similar to the RTSA and the Python example. One specific aspect is that Kodex is a generic platform-provided service that is customized for the use in your application based on the application configuration, e.g., what data to anonymize. A further aspect is that this example integrates an in-memory database as data sink. The build steps are the same, i.e., using the command (`mvn -U install`). Also this application can be executed through a running platform, but also standalone, which requires a specific setup that is included in the example.
- **examples.template:** This example contains the configuration model and the service implementation used as blueprint for the service development workshop in November 2022 and for the service development tutorial videos[Error! Bookmark not defined.]. The example structurally i s similar to `examples.python` and can be built in the same manner (`mvn -U install`).

Figure 59: Using the platform to realize a robot-based visual quality inspection for Hannover Fair 2022 (HM'22).

- **examples.hm22:** For the Hannover Messe 2022 (HM'22) and, in an improved version, for the Tage der Digitalen Technologie in Berlin (TddT'22) the IIP-Ecosphere project team developed a demonstrator application that involves IoT/Factory hardware, the platform as well as a controlling application. For short, the application demonstrates a visual AI-based quality inspection process for a configurable lot 1 production, where a robot with mounted camera takes pictures from three sides of an aluminum car model. A QR code on the indicates an (external) AAS describing the configuration of the car. The two other pictures are utilized by a Python-based AI to detect the number of windows, the color of the tires, the presence of an engraving as well as whether "scratches" were accidentally caused on the surface of the car. The services, in particular the AI, can be executed in different deployment settings, e.g., on a Phoenix Contact AXC 3152 PLC/edge device. For this purpose, the application consists of two generated connectors (OPC UA to obtain input from the PLC, AAS to request information on the car configuration), a camera source, the AI (as service family to switch between alternative AI implementations), the action decider (controlling the overall process) as well as a customized version of the generic, platform-provided `TraceToAAS` service. The inspection results are displayed on an Angular Web Application running on a tablet based on the

information in the application AAS (via the TraceToAAS service). The application involves forward data flows as described above, but also backward data flows for controlling the process. Figure 59 illustrates the physical setup, more details on the application and the lessons learned while creating/integrating the application can be found in [EPR+22]. The build process is similar to the examples described above. The application can be executed standalone for testing as well as via the platform through deployment plans (examples included).

The examples.hm22 application has been evolved to a federated learning showcase for Hannover Fair 2022 and, in an improved version, for the Days of Computer Science 2023 in Hildesheim as well as for Nürnberg Digital 2023. This demonstrator showcased the application of a platform-supplied federated learning component with two cobots representing two distinct factories. One cobot learned anomalies from the other without having seen them before.

- **examples.emo23:** For the final public exhibition of the IIP-Ecosphere project at EMO'23, the Hannover Fair 2023 demonstrator has been extended towards a visual quality inspection process of two collaborating cobots (Figure 60). One cobot grips a model of the car and identifies it through a sensor provided by MIP technology (AI-enhanced sensor reading, finally leading to the retrieval of the respective car AAS). After successful identification, the whole cobot still gripping the car is moved by a Lenze linear drive close to the other cobot, thus overlapping the safty perimeters of both cobots. There, the model car is placed and the linear drive moves the identification cobot back to its original position. On the forward way, obstacles like friction and tension can be applied to the drive to be detected by a condition monitoring AI service. The second cobot then performs the quality detection as done in the examples.hm22 application.

Figure 60: Showcasing the platform to realize a magnetic product identification and a robot-based visual quality inspection with two cobots for EMO 2023.

- **examples.VDW:** The IIP-Ecosphere partner VDW is prominently involved in standardization activities for industrial production, in particular in OPC UA through the UMATI initiative. VDW offers a test server providing model instances according to OPC UA companion specs. As one of the connectors that is shipped with the platform is an OPC UA connector, an example based on that connector may be interesting to the reader. Connectors for the platform may be created by hand or, the preferred way, generated from a configuration model. This example

illustrates both, a handcrafted connector as well as the integration of a generated connector into a demonstrating piece of code. It is important to mention that we do not read out the whole UMATI test server structure, rather than just a small piece for the OPC UA Woodworking Companion Spec[99]. The example represents the respective part of the Companion Spec in terms of IVML, imports the structure into the configuration model, adds some information about caching and server coordinates and creates a simple application mesh that just consists of a source (the connector), i.e., no further processing with the obtained information happens here (and some of the generated artifacts may be unusually empty). For executing this example, using the command (`mvn -U install`) you first execute build steps to obtain the actual configuration meta-model (intentionally not included), run the application generation and compile the example.

- **examples.templates:** This example project displays the usage of out `impl.model` to create an application with three services of which one is a python service utilizing AI. It consists of two parts, the `examples.templates.model` and the `examples.templates.impl`. The `model` part is based on the `impl.model` and contains the configuration `.ivml` files, with the use of `mvn generate-sources` we can generate templates fitting these configuration files. The `examples.templates.impl` is created by importing the resulting .zip file in the gen directory of the `exmples.templates.model` project, it contains the implementation of the concrete service functionality. The services implemented in this project are made available to the `examples.templates.model` project by running `mvn install` in the `.impl` project. The application is completely build by then running `mvn install` in the `.model` part of the project. The result is an application that takes the data from the resource directory and evaluates it utilizing a python service with a pre-trained random forest classifier.
- **examples.MIP:** Use of the meta-model extension for MIP technologies as applied in examples.emo23. The meta-model extension encompasses a customized MQTT connector and related data types pre-defined in the configuration meta-model. The example illustrates a data ping-pong between the MQTT connector and a Python AI service to improve magnetic identification measurements utilizing (a non-included AI). The sensor software pushes data to an MQTT broker, which is taken up by the connector, improved by the AI, passed back to the sensor software, validated and approved by the sensor software, pushed again and bypassed by the Python AI to a downstream application. The example cannot be executed without the respective hardware by MIP technologies.
- **examples.modbusTcp:** Regression test example for the MODBUS/TCP connector. Contains a simple application, but just for completeness of the configuration model. The application is not used, rather than the connector is directly tested.
- **examples.REST:** Regression test example for the REST connector. Contains a simple application, but just for completeness of the configuration model. The application is not used, rather than the connector is directly tested.
- **examples.MDZH:** Conceptual example for an energy demonstrator in the Mittelstands Digital Zentrum Hannover. Includes OPC UA, AAS, serial and INFLUX connector.

6.8 Creating an Application

Due to the configuration and instantiation process, the (manual) creation of an application for the platform is not just a matter of some programming tasks. In this section, we summarize and detail the steps, which in future versions shall be better supported by respective platform tooling. For this version, you shall be able to realize code in Java or Python (depending on the configuration that you

[99] Woodworking is not really related to the aims of our work, but it was the first one that we identified as potential candidate in the UMATI test server and that was reasonable large but also not too large to be turned into an IVML model (automated work in this direction is planned for the next release).

will create) as well as get to know how to work with Maven. More details on project structures or default build sequences will be presented in the next sections.

Figure 61, summarizes all steps that are necessary to create an application consisting of Java services. We assume that you use `example.python` or `example.rtsa` as blueprint. While Figure 61 is meant to provide an overview, we briefly discuss now the individual steps (assuming that you have a most recent version of the platform sources from GitHub in your local workspace). For more details, please follow our tutorial videos on service implementation[Error! Bookmark not defined.].

1) Download the template project `impl.model` from GitHub[100]. `impl.model` contains a setup for the configuration and code generation. You may rename the projects as you desire. Akin, you may rename the maven artifact name in the respective `pom.xml` file. We will continue referring to the project as `impl.model`.
2) In `impl.model` you need to adjust the .ivml files in `/src/main/easy` to fit your needs i.e. adding or changing datatypes, for this you can refer to our example projects as a guidance i.e. `src/main/easy` in `examples.rtsa` or `examples.python`, the install package or `SimpleMesh` and `SimpleMesh3`. If you plan to have multiple examples/configurations on the same machine, please consider changing the Maven artifact name in the variable `sharedArtifact` in `TechnicalSetup.ivml` so that each project can be based on individual interface artifacts. Make sure your `pom.xml` does contain the properties `<iip.model>` with the value `PlatformConfiguration` as well as `<iip.resources>` with the value `resources`. Afterwards execute `mvn -U generate-sources` to obtain an actual copy of the configuration meta-model and create the `/gen/py/` directory containing `ApplicationInterfaces`, the application itself and a template directory with the `.zip` to implement the services. The `ApplicationInterfaces` are needed for the service creation and will be added to your maven repository. To utilize the templates, you we recommend importing them as a separate project right besides your `impl.model` project. The template project will contain template Java service sources, a matching Java jUnit test suite, template Python service sources, tests and a Python Eclipse editor setup file for PyDev, if configured, template mocking test input files, and an initial YAML-based identity store file.
3) Import the generated implementation template into your Eclipse (we refer to it as project `impl.impl`). Depending on the services stated in your model, now the services must be implemented in terms of Java or Python code, respectively. Please consider in particular architectural constraint C9 stating that generated code must not be altered. When the services are ready, compiled and tested, execute `mvn install` so that the artifacts of `impl.impl` are installed and become available. Also implement the unit tests for the services and validate your implementation. For later steps, adjust the mocking input JSON files in `src/test/resources`.
4) After successfully running `mvn install` in the template project, you can go back to your `impl.model` project and run mvn install there, make sure that the values for artifact in your `.ivml` configuration are correct and point to the location of the services form the template project. You can check this by looking at the `POM.xml` of the template model and reading the `groupId` as well as `artifactId`.
5) Step 5 generates the full application artifact for Spring Cloud Stream. We recommend to execute generated Spring Cloud Stream test cases (documented in `README.md` of `impl.model`) for each individual service first. Then, you may execute the application on your local machine (as documented in `README.md` of `impl.model`) and finally on the platform (see also Sections 3.11 and 7.4).

[100] https://github.com/iip-ecosphere/platform/tree/main/platform/tools

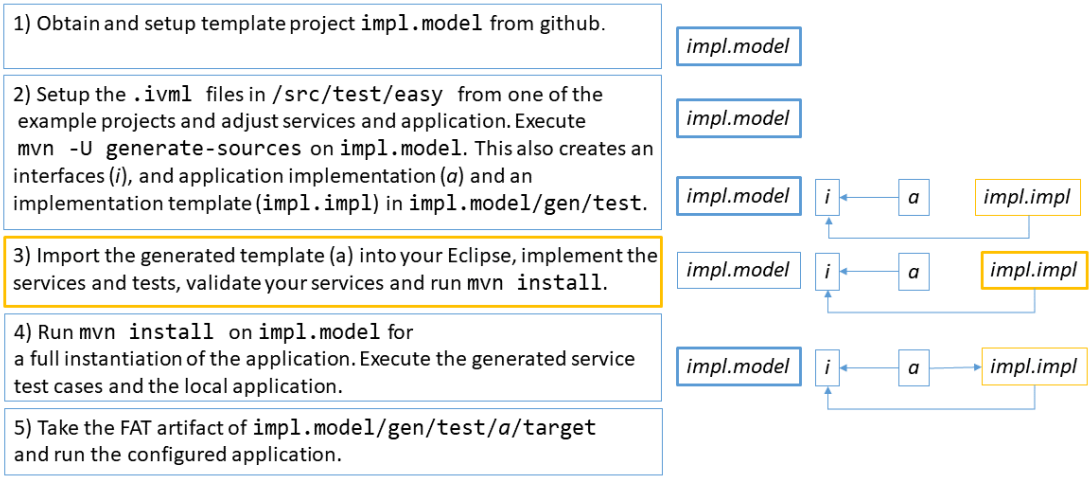

Figure 61: Steps to manually define services and create an application.

6.9 Project Structures

Due to the Maven build process and the generation of code during the instantiation, the service implementation creates/assumes a certain project structure that we will introduce in this section. As creating structures across multiple programming languages may be challenging, we also offer generated template projects for configured applications/services as we will discuss at the end of this section.

Figure 61 illustrates the overall structure of an all-in-one implementation project as we use it for most of the platform examples. On top-level, it consists of three folders, one folder for the generated code (gen), one folder for the own sources (`src`) relying on the generated code, the `target` folder containing the created/compiled binaries and the Maven build specification (`pom.xml`). There may be further files, e.g., for the integration into a CI environment.

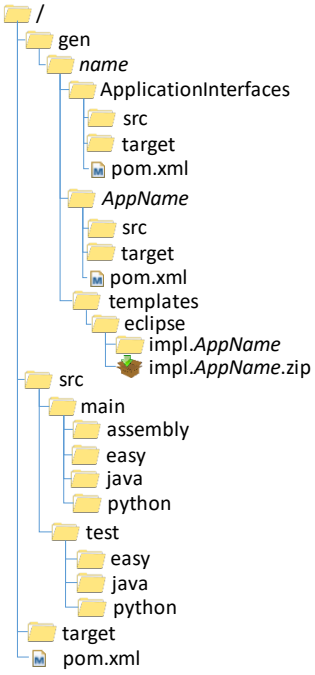

Figure 62: Overall structure of an implementation project.

Initially, the gen folder may not exist as it is created and filled during the instantiation of the project. This may lead to the effect that after an initial checkout, (local) dependencies are not in place, i.e., your IDE shows errors. After executing the instantiation/code generation and updating your implementation project, these errors usually disappear. The `src` folder contains your code based on the generated code. `src/main/assembly` contains project-specific Maven packaging scripts, e.g., for Python. These assembly scripts must be called from the main `pom.xml` of the project, i.e., if you need further assembly scripts, e.g., for packaging AI models for multiple services in individual re-usable assets[101], you must hook them manually. `src/main/java` contains production Java code, similarly `src/main/python` (may not exist if your services do not need python scripts). `src/main/easy` contains the configuration model of the application[102]. Akin, the `src/test` tree contains testing code, e.g., for Java or Python

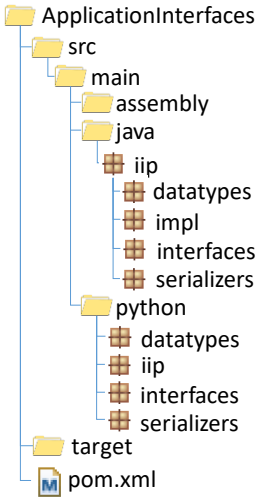

Figure 63: Detailed structure of the generated application interfaces.

After the first instantiation run, the gen folder consists of three sub-folders and a template POM file. The `ApplicationInterfaces` folder (if the name was not changed in the configuration model) contains the data type interfaces, the data type serializers and the service interfaces for all applications of a platform installation. The second folder, which is named based on the respective application configuration in the model, contains the service integration into the service execution engine and service engine specific testing code, which relies on the `ApplicationInterfaces`, but also on your code. Thus, besides downloading the configuration (meta)-model, the build process consists of creating/deploying the sources in the `ApplicationInterfaces` folder, the compilation of your code based on `ApplicationInterfaces` and the generation of the second folder with the service integrations, which leads to the packaged application. The typical structure, which is rather similar to the overall project structure, is shown in Figure 63. The difference comes from the generated code and assembly descriptors and the assumptions about the underlying code structure. On the Java side, `datatypes` (representing the application data to be passed between connectors and services), the implementation (`impl`) of default services as basis for your service implementation, the `interfaces` for the services and the wire format `serializers` (based on the format selected in the configuration model) are generated. Similarly, on the Python side, `datatypes`, service `interfaces` and `serializers` are generated. The `iip` folder contains the Python service environment and is

[101] See `examples/examples.emo23` in github as a blueprint.
[102] Until version 0.6.0, the configuration model was in `src/test/easy` and the downloaded meta-model in `src/main/easy`. In version 0.7.0 we untangled this "heritage" to comply with usual Maven conventions, i.e., we migrated the meta-model into the temporary location `target/easy` and the configuration into the production location `src/main/easy`.

created/filled during the build process. If Python-based services are configured, respective `assembly` descriptors are generated and linked into the generated build process in `pom.xml`.

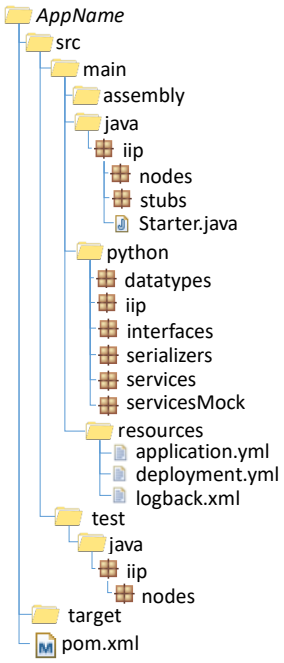

Figure 64: Detailed structure of the generated service integrations.

The name of the generated folder for the application depends on the configured application name. Also this folder represents a typical Maven project structure, here meant to complement the application interfaces and your implemented code in order to derive and package a full application. Similar to the application interfaces, `assembly` descriptors, Java code and, if configured, Python code is generated. The Java code consists of the `nodes` representing the integration of individual connectors or services into the configured service execution environment (by default Spring Cloud Stream). The stubs contain local parts of remote service implementations via AAS[103]. The `Starter` class is the main class of the application and responsible for serializer and service registration (as an extension of the respective class in the service environment). The `python` folder accumulates the generated code from the `ApplicationInterfaces` including the Python service environment (both parts downloaded/updated during the build process). The `services` folder is the default place for Python service implementations and taken over from your implementation project (similarly the `servicesMock` folder, which contains Python service implementations for testing). The `resources` folder contains additional files that shall be made available on the classpath, in particular the Spring application configuration, the service deployment descriptor and the logging configuration. The `test` folder currently contains only a folder for Java sources, here the generated connector and service tests. The `target` folder contains the compiled binary and, in particular, the packaged applications, here in the formats discussed in Section 3.5.2.1.

The third directory contains the application/service implementation `templates`. This folder contains one folder per supported IDE (currently only Eclipse) and within that one folder for the template sources and one ZIP file containing the template sources to simplify the import into Eclipse. The template project as illustrated in Figure 65 contains a `README.md` detailing the generated parts and pieces, some top-level configuration files as well as a `src` folder for the production code (`main`) and the tests in `test`. Both folders may contain Java source code, Python source code and resources according to your configured services. In particular, the `test` folder contains connector connectivity

[103] The stubs are currently not used and may be removed in a future release.

tests (in `iip.connectivity`), simple programs that are intended to run a generated connector against a real device to initially validate the functionality of the device-connector-communication on a rather low level

As mentioned above, most of the provided platform examples follow this structure, in particular to reduce the number of projects in an IDE. In addition, services may be implemented in two separate projects, one containing the model and one the service implementation. Templates for such a setup can be found in the GitHub repository of the platform.

For the future, we envision such a separation allowing the platform to take control over its configuration model. While it then shall still be possible to access the configuration model for download, it will not be in the control of the user/developer anymore. It is then a prerequisite that that the platform offers means to modify the configuration model and to define new/modify existing applications. For this purpose, the configuration component shall mirror its configuration into the platform AAS and allow the management UI to access and modify the configuration. Ensuring the validity of the platform configuration, executing the instantiation process and providing access to implementation-level artifacts will then be in the responsibility of the platform. First steps in this direction are already taken in this version: an initial access to the configuration model through the AAS of the configuration component and the creation of template implementation projects (for Eclipse). A generated template project contains template files to be extended, completed by the developer, assuming for now that a respective `impl.model` project exists. A generated template project for Eclipse contains in particular a

- `README.md` explaining the build process and the test steps
- Eclipse specific files, i.e., `.project` and `.classpath`
- An initial `.gitignore` file
- A `pom.xml` Maven build process file containing the basic dependencies and linking the generated service/connector test cases.
- Java template source files according to the class names of Java services in the configuration file also considering the given packages of these class names.
- Python template service source files if Python services are configured. If Python services are configured, also the python.xml assembly descriptor will be generated and hooked into `pom.xml`.
- Test data specification in `src/test/resources` (to be filled and renamed)
- Initial identity store file in `src/main/resources`

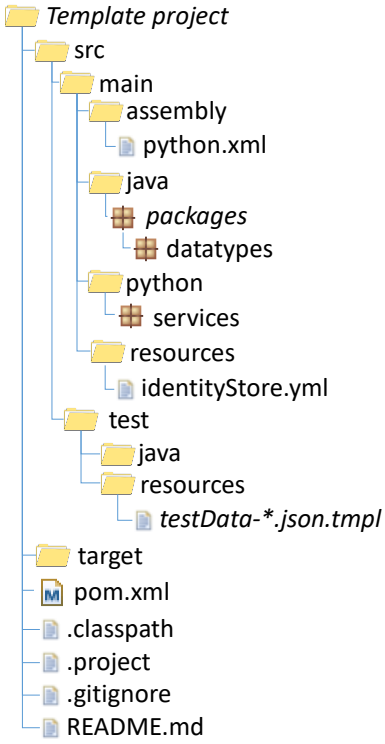

Figure 65: Exemplified structure of a generated service implementation template.

6.10 Default Build Sequences

Section 6.8 provided an overview on how to implement applications with the platform, while Section 6.9 introduced usual implementation project structures. To ease the realization of different applications and their building, we suggest using a set of common build commands across the application implementation projects. These commands have been mentioned before and are summarized in this section.

The build process of the platform relies on Maven, i.e., is specified in Maven build specifications (`pom.xml`). The basis of a build specification based on the platform is the inclusion of the platform dependencies as parent project, which introduces common dependencies and their versions as well as common build steps and plugins (which can be refined in own build specifications if need be).

An application build specification consists of two types of build commands:

1. Build commands related to the configuration model and the (application/service) code generation provided by the platform instantiation process as discussed in Section 6.2. These commands involve obtaining the actual version of the configuration (meta-)model (`generate-sources`), instantiating the broker (`generateBroker`), generating interfaces (`generateAppsNoDeps`) and generating/integrating the full application (`generateApps`)
2. Build commands for the application/service code in the actual project, i.e., compiling the code against the generated application/service interfaces as well as tests of the (mocked) code in the actual project, usually the implemented services.

These two types involve different dependencies and define different build commands. In particular, it is important that some of the dependencies are not mixed among the two types as this usually leads to build errors. This difference is constituted by dependencies to artifacts created by the application/service instantiation, in particular `generateAppsNoDeps`. While the application/service code in the actual project requires the generated interfaces and the testing even the generated services and service tests, the build commands for the configuration model do not require these

dependencies as they are stated in the configuration model. Even worse, if these dependencies are declared for the build commands for the configuration model, they break the initial build as these dependencies are not present: These dependencies are created with the first run of `generateAppsNoDeps` and they are intentionally not deployed to the platform Maven repositories as they may contain IPR protected code, licensed components etc.

For this purpose, a typical build file defines two profiles, one profile per type of build commands. In addition, common dependencies may be defined in the (usual) main part of the Maven build specification, and, thus, are made available to both profiles. These profiles are:

1. `EasyGen`: This profile defined the build commands for configuration model and the code generation (the first type introduced above). As explained above, this profile must not have any dependencies on the code in the actual project and also not to Maven artifacts generated by the platform instantiation commands, in particular `generateAppsNoDeps`. By default, the commands declared in this profile call the `PlatformInstantiator` to create the `gen` folder and to instantiate the platform or application artifacts within there. For the platform examples, the commands are defined in a way that the application configuration model is in `src/main/easy` with an example-specific name and the platform configuration (meta-)model is stored in `target/easy`. The template projects include template build specification that can be adjusted to the naming/locations of your application/company.
2. App: This profile defines the specific dependencies of the application, in particular those that must not be declared in `EasyGen` as explained above (`ApplicationInterfaces` is usually a compile-scope dependency, the application specific artifact explained in Section 6.9 a test-scope dependency). Besides the build process for the project specific code, which typically is covered by the build steps defined in the platform dependencies, this profile may perform specific steps, e.g., packaging though artifact descriptors (mentioned in Section 6.9), but also executions of tests to ease the development (e.g., as in the template build project introduced in Section 6.9) or an execution of the full application for (mocked) dry runs. Usually, all these executions include the Maven test-scope and, thus, not only the `ApplicationInterfaces`, which are required for compilation, but also the application specific artifact containing the generated integration into the service execution environments as well as the generated connector/service tests (the latter also in test-scope).

In many cases it is sufficient to state the specifics of the build process in terms of static values, e.g., the folder where the application configuration model is located or the name of the IVML configuration (file). One particular exception is the resources folder. As discussed in Section 6.9, the resources folder contains files that shall be bundled with the application, including IPR-protected files. Currently, we follow the approach to define two folders, one with non-IPR content, e.g., a fake implementation of RapidMiner RTSA and a second folder with IPR content. The non-IPR content folder can be shared, e.g., in GitHub, while the IPR-content folder is not uploaded. For a correct instantiation, it is important to tell the instantiation process which folder to use and there are situations where you need instantiations for both folders. Thus, the recommended build sequence allows changing the resources folder from outside Maven, via `-Diip.resources="NewFolderName"`, where `NewFolderName` is the name of the actual resources folder.

6.11 Service Realization Rules and Considerations

Realizing an application within the conventions and assumptions of frameworks or an entire platform is not trivial, in particular if conventions and assumptions are not documented. While we detailed the architectural constraints for the platform in Section 4 and we will discuss the implementation in Section 7 and frequently asked questions in Section **Error! Reference source not found.**, it is worth to t

hink about service implementation considerations at this point, i.e., after the overview on creating an application in the previous section.

- As stated in Section 3.5.1, a **service is a long-running function** that may continuously be fed with data. A service starts when its status is set by the platform to STARTING. The responsibility of the service is to do then all preparations for starting, e.g., resource allocation, and to set itself into status `RUNNING` (or `FAILED` for good reasons). Akin to the startup, at any point in time, the platform may request a change to `STOPPING`, giving the service the opportunity to stop processing and to release resources gracefully. The responsibility of the service is to go to `STOPPED` when the cleanup is done. In particular, this means that **global code** of a service implementation needs to be implemented carefully to not execute unintended parts. While variable/attribute initializations are usually fine, e.g., checking for the presence of a GPU causing an exception in certain cases may prevent successful loading of the service even prior to the execution of the application[104].
- Both, startup and shutdown activities shall only **take time as long as absolutely necessary**. We are aware of the fact that some processes take a longer startup time, e.g., Python with TensorFlow or a complex Java service like RTSA.
- While a service is running, it is **kept in memory (to conceptually run forever)** by the platform in order to process data, i.e., it shall be ready to receive data and, depending on its function, produce output in synchronous or asynchronous manner. In other words, a service may keep its resources, ML models, etc. in memory. Of course, in particular on resource constrained devices, it makes sense to keep an eye on the overall memory consumption and to dynamically allocate/release resources that are not frequently used.
- A service **never returns/ingests** "the non-existing object", e.g., `null` in Java or None in Python. Synchronous services always return a value upon their invocation, while asynchronous services may ingest any number of return values.
- **Take care for the executability**, i.e., do not kill your service/the executing JVM and catch/handle your exceptions properly. The reason for this is that the service execution sets the `FAILED` state on a service that unexpectedly dies. This implies that you must also have an eye on the used libraries whether they cause such problems (JVM termination, throwing of runtime exceptions) and that you as service developer are responsible for handling such issues on behalf of your service. For Python services, the service execution environment tries to catch unexpected exceptions on the methods declared by the service interfaces as far as possible. Moreover, Python libraries sometimes overlap and prevent execution, e.g. tensorflow and tensorflow-lite. In this case, the ability of the Python integration can be used allowing the definition of individual service packages to separate conflicting libraries.
- **Most state change activities are handled for you** in default implementations. In most cases, the existing/generated service frame allows you to hook into this process. In particular, platform supplied services do care for their own lifecycle state. Generated Python services are currently not informed about state changes. This will happen when extending the Python service environment. We expect no changes to existing service implementations as the Python services are based on classes with implementation, i.e., just some further methods will occur there.
- **Services are classes.** This is no big deal for Java as in Java every language unit is a class. However, it may be a surprise for Python, as there are also Python Scripts. The Python Service environment requires that a service is a class and that your operations are correctly declared

[104] We experienced such problematic behaviour in a Python service integrated from external standalone sources, where the global code was assumed to be executed with/before the main method. This code failed and prevented loading of the Python service class by the Python service environment.

with `self` parameters. The generated code tries to give you hints about the expected instances for input and output in terms of Python type annotations.

- **Services shall be re-entrant**, i.e., further input data may arrive while your service is processing. While synchronous services may be throttled by the service execution engine, for asynchronous services there is no external visible correlation between input and output data. In other words, for an asynchronous service, the service developer is responsible for handling parallel input data. It is also important to consider that the application design may prevent parallel inputs, but from the service side no assumptions on this can be made. Future versions of the platform may allow services to declare themselves as non-reentrant or a maximum parallelization degree.
- Please note further that **Java services may employ Python** functions, e.g., by executing a script. Here, the class requirement for Python does not apply, as the Java service calls Python as a whole (in contrast to Python services that are hooked into the Python service environment). However, it is important that such Python functions are packaged correctly at compile/packaging time and unpackaged at runtime. For the compile time, a Maven assembly descriptor shall package all Python scripts and their required resources into a ZIP file and deploy it to the Maven repo (classifier `python`, type `zip`) along with accompanying Java code. The Java code shall use the `ProcessSupport` class of the Java service environment, which cares for correct unpacking and execution of Python scripts. The POM of the utilizing service must declare both, the Java and the Python artifact (classifier `python`, type `zip`) as dependencies.
- **Don't be blocking.** All service operations are expected to be executed as fast as possible, in particular service management operations. While there may be operations that take a certain amount of time, e.g., starting a JVM or a complex Python script within a service, longer running, resource consuming (blocking) operations are not permitted, e.g., reading a file and waiting for some reactions of the service. Such blocking operations may interfere with other services or the service control and leave the impression of timeouts, which may cause the respective service to enter the `FAILED` state.
- Do not make assumptions about **locations of file resources**, e.g., images or ML models. A service will be packaged (along with other services of the same application) into a service artifact. The layout of the artifact depends on the capabilities and conventions of the service execution engine. Currently, the platform only employs Spring Cloud Stream with two packaging strategies, see Section 3.5.2, and you do not know which artifact, i.e., which packaging layout, will ultimately be started by the user. Moreover, if further service engines occur, they may come with their own conventions. Thus, the following rules apply:
    - For **Java services**, please rely on the `ResourceLoader` (see Section 3.3.2.2), which is designed to cope with this insecurity in a systematic manner. If for some reason needed, you may introduce your own (local) resource resolver.
    - For **Python services**, please rely only on the existence of the files that you ship with your own service (see above), which is unpackaged into an own temporary directory at runtime. This directory is used as process working directory for the Python process, which then has access to all Python modules and resources, i.e., resources shall be loaded only through relative paths starting at the directory where the top-level Python modules of your service are located. Python Scripts used by Java services are treated similarly.

  For any **other resource**, e.g., the temporary folder, do not make any assumptions about their location. While it is usually at `/tmp` under Linux, this may have changed during platform installation. For windows, the temporary folder is typically located somewhere in the user profile (partially also depending on Windows version and user interface language). Java, as well

as Python offer programmatic access to the temporary folder, e.g., via `de.iip_ecosphere.platform.support.FileUtils` in Java or `tempfile.gettempdir()` in Python.

- Rely on **default service implementations** where feasible. For each service interface, we generate a service interface and a default implementation during the instantiation process (see also Section 6.2). For Java, interface and default implementation are separated, for Python the interface and implementation are formed by the same class. Currently, such a default implementation contains code for handling the service parameters as well as for the asynchronous data ingestors[105]. Due to type safety, both aspects, parameters and ingestors cannot be realized in a generic manner. Although it is not difficult to implement both aspects manually, it is also a tedious task. Moreover, it is a common programming error to miss adjusting the parameters or ingestors when your application model changes, e.g., when parameters are added or multiple ingestors are needed due to multiple output streams of a specific service. Inconsistent service implementations easily lead to long debugging tasks, while just a parameter declarator is missing, a parameter name is wrong or the service implementation expected to receive just a single ingestor. To ease the work of service developers and to keep up with changing models, we recommend to rely on generated default service implementations wherever feasible.
- Service output to the console is typically subject to **logging**. The logging target may depend on the active service execution engine, e.g., for Spring Cloud Stream, a temporary folder per service is created, which contains a folder with the deployment name and within that folder one file for the standard output or standard error stream. It is important to mention that depending on the use of Python for services, the logging target may differ. In the current version of the platform, Python services being executed through the Python service environment log into both streams, i.e., the log output appears separated. The process-based execution of Python functions, which may be used for service implementation, currently joins both streams for technical reasons and logs them to standard output.
- In addition to logging, further service output can be used for **debugging**, e.g., the generated (optional) logging or tracing of messages. Here, logging is more for local debugging while tracing (messages via the transport layer) can be helpful for remote debugging. To support remote debugging, the platform contains a transport message logger, which allows for receiving, emitting, storing and basic filtering of status, trace and monitoring messages.

[105] An ingestor is a (lambda-)function that encapsulates the data ingestion so that the service developer does not have to care for stream names or routing aspects, which are typically specific for the selected service execution framework.

7 Implementation

In this section, we briefly discuss aspects of the implementation of the platform, i.e., decisions we made during the implementation (Section 7.1), the mapping between implementation projects and platform components/layers (Section **Error! Reference source not found.**), how to obtain a binary v ersion (Section 7.2), the dependencies and how to compile the sources (Section 7.3), and how to install and to use the platform (Section 7.4). Section **Error! Reference source not found.** outlines further considerations for a permanent or distributed installation. Finally, Section 7.5 introduces an environment for testing and evaluating the platform and applications.

Intentionally, we do not discuss code in the platform handbook. For this purpose, we refer the reader to the IIP-Ecosphere GitHub repository[16] and in particular the Markdown[106] readme files that are provided for the platform and for individual components.

7.1 Implementation Decisions

We briefly discuss now technical decisions or issues that occurred during the development of the platform. This list may not be complete[107] 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 3.3).
- 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 and the implementation is stable) as well as BaSyx as AAS implementation. See architecture rule C6 and C8 in this respect.
- 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 platform dependencies (and transitively in components such as EASy-Producer).
- For **logging**, we rely on the logging support (see Section **Error! Reference source not found.**). L ogging 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

[106] https://de.wikipedia.org/wiki/Markdown

[107] 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

operation or property call, which builds up a network connection to an AAS implementation 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 attached 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.

- As an operationalization of architectural constraint C11, we limit the type of the **exceptions** to be used in Java code. While in Java there is a style of throwing exceptions of different types for different purposes, we decided to use only a few types for a more uniform exception handling, namely, as checked exceptions `IOException` if any form of input/output may fail, `ExecutionException` if processing may fail (in particular indicating failing AAS operations) and `IllegalArgumentException` as the only unchecked exception if parameters are semantically wrong (e.g., for object construction)[108]. However, unchecked exceptions may be false friends and shall be used carefully. As discussed for C11, emitting exception information like stack traces to the console is generally forbidden and logging shall not be considered as the best or only option but rather as the ultimate option if no other exception handling like alternative/default processing or recovery is possible. Logging of exception traces shall only be applied if one cannot clearly distinguish between an error or a warning situation.
- Test artifacts shall be strictly separated from production artifacts as already mentioned in Section 7.1.

[108] Generally, please avoid such Ninja exceptions or at least document them where absolutely required in terms of (unneeded) `throws` clauses as well as in the related Javadoc comment.

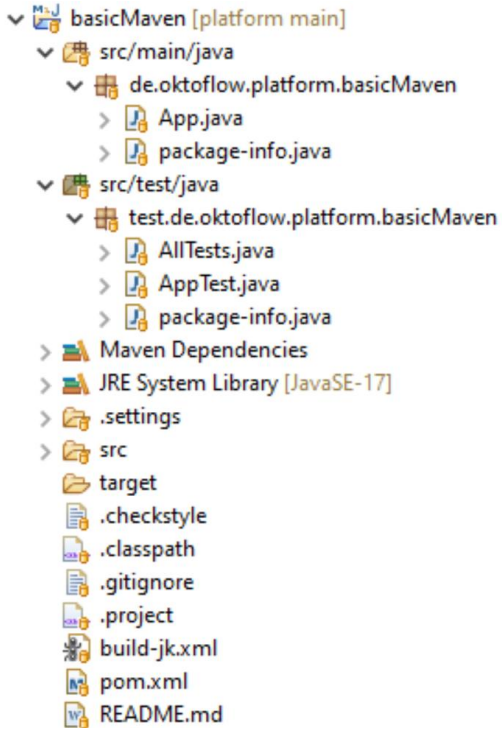

Figure 66: Structure of the component template "basicMaven" in the GitHub repository.

For easing the implementation of new components or examples, we provide several templates that already have built in conventions and development setup. Figure 66 illustrates the template for a Maven-based Java component[109].

- The name of the project indicates the component name that shall be reflected in the java packages for production and testing code as well as in the Maven POM file. The name of the test package shall start with "test", but if too many methods have to be declared public to be accessible to tests, the package may directly start with "de" to allow for package access. Packages shall be documented (`package-info.java`), the main test suite is `AllTests.java` in the test package (only tests declared there will be executed).
- The component is by default developed for JDK 17 as specified as system library and compiler settings in the Maven POM. Where applicable please rely on the platform's JDK version, i.e., the smallest possible JDK version for compatibility.
- The component template ships with a checkstyle setup taking the style information from the `platformDependencies` project (which must be obtained from GitHub and installed by Maven before) use. If you plan to realize an implementation component/plugin, please check whether `platformDependenciesBOM` or `platformDependenciesSpring` is more adequate rather than re-defining managed dependencies. Please also consider, that certain support plugins (cf. Table 4) with helpful functionalities do exist and shall be used over re-including similar dependencies.
- There is a default `.gitignore` file that excludes the target folder. Please ensure that only needed files that are not generated/obtained during the build process are committed, e.g., the configuration model (usually in `target/easy`) shall not be committed rather than obtained from its Maven deployment (usually phase `generate-sources`) so that model updates can easily be followed by issuing that command, also during automated builds.
- The `build-jk.xml` file is needed as starting point for the continuous integration (Jenkins, therefore "`-jk`"). This file refers to further ANT imports containing the settings for Jenkins as

[109] Located in https://github.com/iip-ecosphere/platform/tree/main/platform/tools

well as macros for Maven execution and deployment. As these files are in other repositories, it may be that your IDE issues errors about missing files. You can ignore these errors. Upon first use of the template, please change the name of the project as well as the pattern for the files to be taken from the Maven target folder for Maven deployment. In specific cases, further adjustments are needed here. Some example projects do not have this file rather than a build.sh Linux script, which in these cases is (so far) more convenient then an ANT file.

- `pom.xml` is the Maven build specification. Usually, it declares the platform dependencies as parent, defines only its `artifactId` (taking over the IIP-Ecosphere group), its deployment form, name, description and dependencies. The build plugins are inherited from the parent. In some cases, e.g., for obtaining/unpacking specific artifacts like the configuration model further steps can be added, which usually extend the existing build setup.
- Ultimately, `README.md` is the readable documentation of this component. It shall briefly explain the aim of the component, its setup (Yaml-Structure), its specific requirements/limitations but also actual issues and problems. Please keep this file up to date. Upon first commit, this file shall be linked into the parent GitHub folder `README.md` (see the platform layering for selecting a proper folder) as well as in the top-level `README.md` file of the platform by HTML links.

Along with this project, the `tools` folder of GitHub contains also the template project `impl.model` for the setup of applications. In contrast to the example structures, which unify both sides currently through two separate Maven files, these projects shall help setting up applications in separate folders. Moreover, the `tools` folder also contains the project `MavenCentral`, which contains tools for creating a release, like changing Maven version numbers, but also a tool to clean up a local Maven repository from superfluous SNAPSHOT versions.

7.2 Obtaining the Platform

The sources of the IIP Ecosphere platform are available on GitHub[110]. Released binaries of the platform can be obtained from Maven Central[111]. Snapshots from the continuous integration can be obtained from the SSE Maven repository[112].

7.3 Compiling the Platform

Due to the various optional and alternative components in the platform that we manage in individual artifacts/Eclipse projects, compiling the platform is not trivial. Usually the binaries of the individual components are either available via Maven central (releases) or SSE Maven repository (snapshot, releases). These builds are created and deployed by the SSE Continuous Integration (CI) server as illustrated in Figure 67, which 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).

[110] https://github.com/iip-ecosphere/platform/
[111] https://repo1.maven.org/maven2/de/iip-ecosphere/platform/
[112] https://projects.sse.uni-hildesheim.de/qm/maven/de/iip-ecosphere/platform/

✓		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 67: Screenshot of the SSE Continuous Integration server (cropped)

For completeness, we discuss below the dependencies among the individual components of the platform (as illustrated in Figure 68). As indicated in Figure 68, some components need specific settings for successful testing, e.g., the RTSA components need to know which JDK to use for RTSA execution (strict Java 8 for the original RTSA).

The `platformDependencies` project defines the common build process steps as well as administrative information such as authors or source code management required for Maven Central. However, since version 0.8, the `platformDependencies` only declare properties for dependencies that are used by multiple platform components, but no dependencies (cf. constraint C16) Thus, `platformDependencies` serves as parend POM for all platform core components, which must not have any external dependencies except for the Java library. These version properties are turned into managed dependencies in the platform bill-of-material (`platformDependenciesBOM`), which is (directly or indirectly) used as parent POM for all implementing components (cf. Section 4). Such a bill-of-material is helpful to restrict the diversity of dependency versions, while, if required, individual implementing components can override the version. In an ideal case, if all dependencies are homogenized after an entire update of the dependencies, such overrides are not needed. Moreover, components and application services that rely on the application/service execution version of Spring (Cloud Streams) are based on `platformDependenciesSpring`, which extends `platformDependenciesBOM`.

Building the platform starts with compiling and packaging the platform's Maven plugins for dependency management (obtaining, unpacking and updating the configuration meta-model), Maven invocation (executing sequences of goals in different profiles on the same POM) and Python (syntax check and testing). As these plugins are used by the build process defined in the platform dependencies, they must be built before without upstream dependencies.

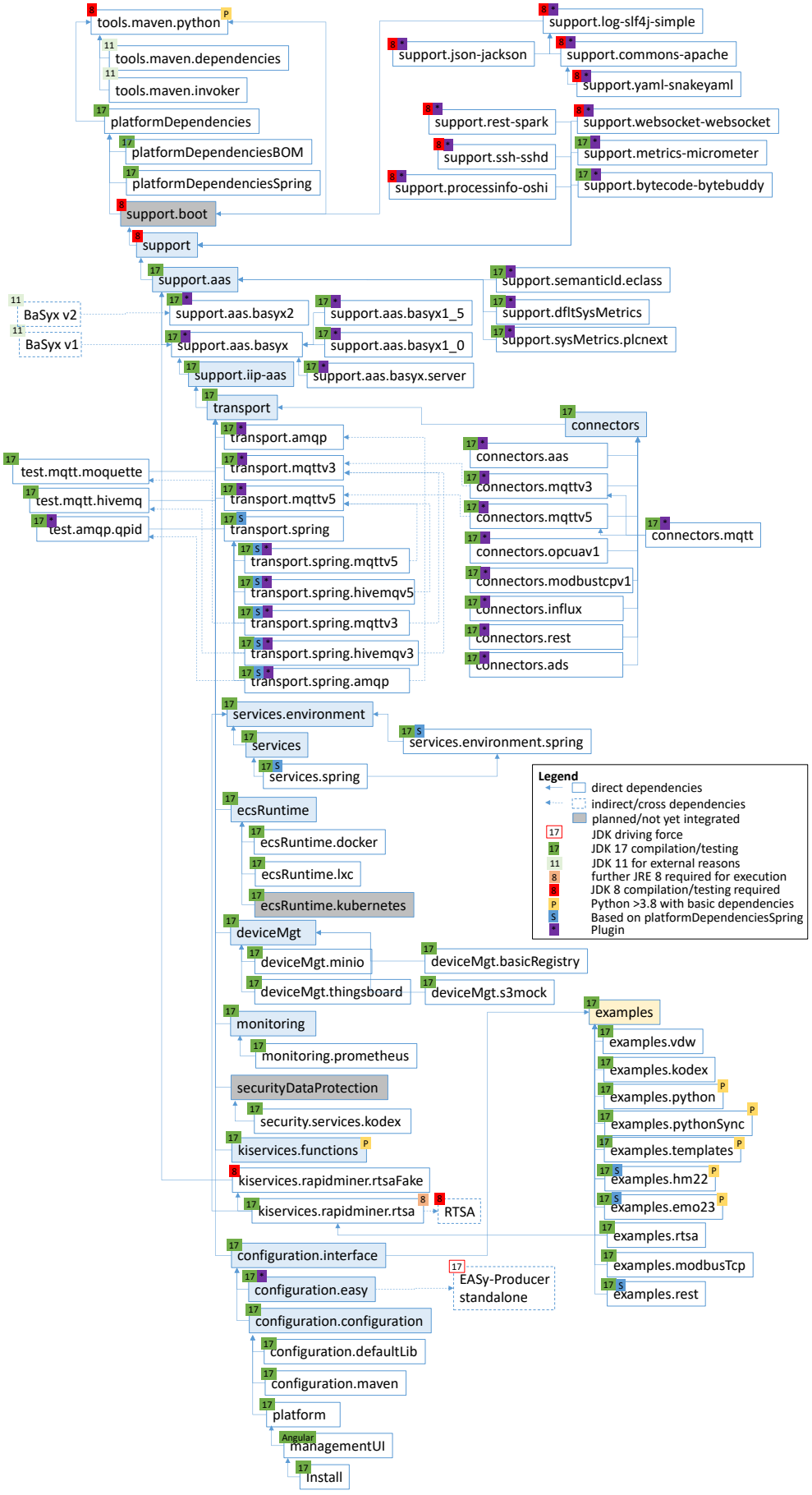

Figure 68: Dependencies among the components (platform examples not shown, folder names in GitHub)

The Support Components consisting of the basic plugin mechanisms and plugin interfaces in `support.boot` (along with implementing plugins), the basic utility classes and interfaces (with their implementing plugins) in `support` and the AAS abstraction `support.aas` are the most basic components without further dependencies to the platform. As `support.boot` and `support` provide some basic functionality that is also used in components that strictly depend on Java 8 or would break their build flow, `support.boot` (and its implementing plugins) as well as `support` are also based on Java 8. However, plugins for which interfaces are defined in `support` but which are not used by support may be based on more recent Java versions. In particular, `support` integrates a Python helper class identifying the actual Python binary from the platform's Maven plugin, which, in turn, forces the Maven Plugin to Java 8.

The BaSyx default implementation and the `iip-aas` support functions depend directly on `support.aas` and are build when `support.aas` changes. Further, two basic Maven plugins for realizing an integrated build process for the platform and the applications directly depend (like `support.aas`) on the platform dependencies.

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 plugin for Spring Cloud Streams (`services.spring`) as well as the generic service environment (`services.environment`) and the Spring-specific service environment (`services.environment.spring`). `services.spring` contains as subprojects `test.simpleStream.spring`, a testing artifact realizing a simple stream processor chain as well as `services.spring.plugintests` for testing the Spring service manager in a class-loading isolated clean-root environment. The platform's Maven plugin for Python depends on the generic service environment, in particular to rely on common capabilities to determine the actual Python binary to be executed (which may differ in the CI environment for historical reasons).

The resource/deployment components (`ecsRuntime.*`) are partially realized, e.g., the ECS runtime and the container manager for Docker. The container manager for Kubernetes and the platform monitoring are in planning/realization and not part of this release. The first components for the device management have been integrated (`deviceMgt.*`), including an optional integration of `Minio` for object storage and `ThingsBoard` for device management. Similarly, initial steps towards the integration of security and data protection services (`securityDataProtection.*`) have been done. Within the reusable intelligent services, the RapidMiner RTSA (version 14) requires Java 8 for execution. As RTSA is an IPR protected component, the regression testing and the RTSA example are based on a fake version of RTSA, a program that pretends to be RTSA without its AI capabilities. In turn, our fake RTSA must be built with Java 8 to mimic the prerequisites of RTSA.

The `platform` server(s) component provides the startup sequence for central services as well as the preliminary command line interface for platform functionality. Currently, neither the device management nor the security services are part of the assembled platform server component.

Further, there the integration of the configuration model (`configuration.configuration`) is based on the configuration platform interfaces (`configuration.interface`) and the default configuration technology plugin (`configuration.easy`), which utilizes on the capabilities of EASy-Producer (stand-alone, Maven-based integration, JDK driving force through Eclipse/xText). `configuration.easy` contains a sub-project for defining the implementations of test application services (`test.configuration.configuration`) for testing the configuration model. The platform's Maven plugin `configuration.maven` for executing the platform instantiation depends on `configuration.configuration` and the actual configuration technology plugin. This Maven plugin provides goals for all instantiation tasks, an encompassing testing task, which can start an entire platform, as well as a goal to manipulate text files.

Dependent on the configuration model, there is the platform management user interface. As the Web user interface is realized in Angular based on information from the platform AAS as backend, this requires a different build process. The TypeScript code of the UI is compiled using angular, packaged, archived by the CI server and then, using a pseudo Maven POM, deployed as binary component into the Maven repository of the platform. Testing the platform management interface heavily relies on the configuration Maven plugin, which instantiates a platform and runs it together with the Angular tests in coordinated manner. The platform instantiation takes this binary up as usual for other components with binary processes, unpacks and customizes the UI.

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. Apache Qpid (due to requested bug fixes) and HiveMq are further driving forces for the selection of the JDK for the platform.

The platform examples (`example.*`) act as platform regression tests, i.e., the respective configuration model is instantiated and the application is executed in testing mode using the test goal of the configuration maven plugin. Usually, the platform examples are all-in-one projects, which also heavily rely on the orchestration of the platform's invoker Maven plugin (running the instantiation of the broker, the interfaces, the application build and the application instantiation/integration in coordinated manner as sub-maven executions).

If a local build is required, we provide a multi-module Maven POM on the top-level of the platform code repository. Given that all required software (Java, Maven, Python and dependencies due to building the examples) is installed as discussed in Section 7.4, the Maven command `mvn install` builds the full platform. Due to specific dependencies between tests and their service implementation, a first offline build may require the setting `-Diip.build.initial=true` and to speed up the build (otherwise it may take around two hours), tests may disabled with `-DskipTests`. Both optional settings must be stated at the end of `mvn install`.

7.4 Installing and Using the Platform

As discussed above, the platform must be configured and instantiated before it can be executed or installed. Thus, the continuous integration or the released binaries do not provide or represent complete platform executables, respectively. For agile documentation, the detailed installation procedure with alternatives is now in github[113].

[113] https://github.com/iip-ecosphere/platform/blob/main/platform/documentation/INSTALL.md and https://github.com/iip-ecosphere/platform/tree/main/platform/tools/Install

7.5 Environment for Testing and Evaluating the Platform/Applications

With all the features and functionalities provided by the platform and in addition to that, the developed applications and services added to the platform, there is a need to test the platform and the (distributed) applications to make sure that they operate without errors. Moreover, besides validation, it is interesting to evaluate the platform and applications for their runtime properties, e.g., to determine memory consumption, CPU usage, performance and residual capacities for the evolution of the platform as well as individual applications. As the whole cycle of configuring, installing, deploying starting, evaluating, stopping, and uninstalling applications as well as platform components is not trivial, we aim at providing support through the **Platform Evaluation and Testing Environment** (PETE) that allows for automated, distributed, monitored, and repetitive tests and evaluations.

The concept of an experimentation workbench was presented by [SEK21]. They used Jupyter Notebook Project[114] which supports experimentation and to some limited degree replication and sharing. Jupyter Notebook in a nutshell is an open-source web-based interactive development environment for notebooks, code, and data. Jupyter Notebook supports over 40 programming languages like Python, R, Julia, and Scala. Using Jupyter Notebook gives the ability to perform a stepwise execution of the experimentation. We decided to use Jupyter Notebook Project to develop and realize the steps in the PETE. In addition to that, there is a script that converts the Jupyter Notebook to a Python script which can be used to do a headless execution (without using Jupyter Notebook).

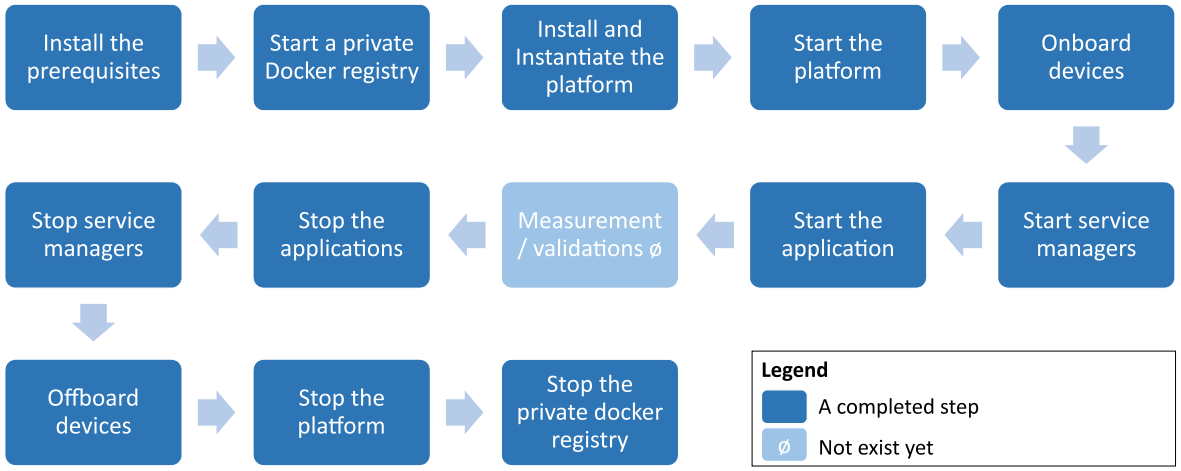

Figure 69: The steps executed automatically by PETE

PETE covers 13 steps as shown in Figure 69: The steps. Those steps are executed automatically by PETE. The step “Measurement / validation” (light-colored step that has “ø” sign) is not implemented yet. The steps are:

1. Install the prerequisites: the required setup needed for the platform on the server and devices that should be used for the test.
    a. Server setup includes Java, Maven, Docker, and Python with basic dependencies.
    b. Devices setup: If the device will use the generated containers, then the setup includes just Docker. If the device will use normal scripts, then the setup includes Java, Maven, Docker, and Python with basic dependencies.
2. Starting a local Docker registry: pull and run the registry container, then setup the local Docker registry to not use certificates (as mentioned in Section 8.6).
3. Install and Instantiate the platform:
    a. For the server, it is a full installation of the platform with creating the artifacts and containers for the applications and then adding the containers to the local Docker registry.

[114] https://jupyter.org/

   b. For the device, setup Docker to use the local Docker registry to pull images (specify that no certificate needed for pulling images from the local Docker registry).
4. Start the platform: run the platform scripts on the server (broker.sh, platform.sh, mgtUi.sh, and monitoring.sh).
5. Onboard (join) devices:
   a. Using the generated containers: pulls the respective ECS container from the local Docker registry and runs it on the device to join the platform.
   b. The local installation way using scripts: download and run the respective scripts on the device to join the platform (broker.sh, ecs.sh, and serviceMgr.sh).
6. Start service manager: if the device uses the generated containers then run the respective service manager container.
7. Start the application:
   a. By services functionality in the platform.
   b. By deployment plan functionality in the platform.
8. Measurement and validations ø (not yet implemented).
9. Stop the applications.
10. Stop service managers: if the device uses the generated containers then stop and remove the respective service manager container.
11. Offboard (Remove) devices:
   a. Using the generated containers: Stop and remove the respective ECS container on the device.
   b. The local installation way using scripts: Stop the respective scripts on the device (broker.sh, ecs.sh, and serviceMgr.sh).
12. Stop the platform: Stop the platform scripts on the server (broker.sh, platform.sh, mgtUi.sh, and monitoring.sh).
13. Stop the local Docker registry: Stop and remove the registry container.

The installation guideline how to setup PETE and how to execute PETE stepwise or headless is in the platform github[115]. We are using Python scripts developed in Jupyter Notebook to manage tests. The Python scripts generate command scripts either shell- (Linux) or batch- (Windows) based on the operating system of the machine that runs the Jupyter Notebook, this machine that runs the Jupyter Notebook is called the **TestManager**. The generated command scripts execute the actual scripts remotely to interact with the server and devices. Those scripts are executed remotely to do different actions like installing the prerequisites, starting a local Docker registry etc.

The PETE consists of folders and files:

1. **Setup:** A file (`TestSetup.yaml`) used to manage the testing environment, the available and usable machines/devices, the platform, the applications to execute, and the tests and their properties (e.g., repetitions, testing/evaluation time, etc.).
2. **Scripts:** A folder that holds the shell (Linux) or batch (Windows) scripts that perform the actual execution of the testing/evaluation process on the involved machines. There are two types of scripts - Server and Device scripts - based on the respective role of the machine in the test. Those scripts are moved automatically from the **TestManager** to the machines that are included in the test based on the role of the machine.
3. **Commands:** A folder that holds scripts generated by Jupyter Python Notebook, those commands are generated based on the operating system in the **TestManager**. Those commands remotely execute the scripts moved to the machines.
4. A **Jupyter Python Notebook** (`TestManagementScript.ipynb`) that reads the `TestSetup.yaml` setup file then creates and runs the commands which in turn remotely execute the scripts moved to the machines.

[115] https://github.com/iip-ecosphere/platform/blob/main/platform/tests/test.environment/README.md

5. **Logs:** a folder that holds all the logs from running the commands and scripts.
6. **AllLogs**: a folder that holds the logs from previous runs of the test named by the date and time of the test.
7. **RunTheTest.sh**: a script that run the test in headless execution (Without Jupyter Notebook):
    a. Convert Jupyter Python Notebook to Python script.
    b. Extract the number of repetitions of the test from `TestSetup.yaml` setup file and repeat the test for the number of times defined in the `TestSetup.yaml` setup file.
    c. Move the current logs from **Logs** folder to **AllLogs** folder and give it a name (the date and time of the test)

8 Summary & Conclusions

Realizing an open (experimental) IIoT/I4.0 platform is a significant amount of work. A solid foundation was performed in IIP-Ecosphere and is taken on in the oktoflow platform. This handbook provides technical insights into the ideas, concepts, rationales, designs and implementation state of the current release of the oktoflow platform. The rationale behind this document is to enable interested parties to discuss with us 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. Future versions may include lessons learned and (reactions to) 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 [ESA+21, SSE21]) 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 summary, the release accompanied by this handbook realizes a good basis for future work and case studies. For the next release, we plan in particular for the following missing functionality:

- Improved UI including access to configuration
- Optional integration of Kubernetes based on flexible protocols
- Integration of further (AI) services.

9 References

[BBB+20] S. Bader, H. Bedenbecker, M. Billmann, A. Bondza, B. Boss, S. Erler, K. Garrels, T. Hadlich, M. Hankel, O. Hillermeier, M. Hoffmeister, M. Kiele-Dunsche, J. Neidig, A. Orselzki, S. Pollmeier, B. Rauscher, W. Rieder, S. Stein, B. Waser, Generic Frame for Technical Data for Industrial Equipment in Manufacturing (Version 1.1), Plattform Industrie 4.0, 2020, https://www.zvei.org/fileadmin/user_upload/Presse_und_Medien/Publikationen/2020/December/Submodel_Templates_of_the_Asset_Administration_Shell/201117_I40_ZVEI_SG2_Submodel_Spec_ZVEI_Technical_Data_Version_1_1.pdf

[Cas21] M. G. Casado, Service and device monitoring on devices in IIP-Ecosphere, IT-Studienprojekt, Universität Hildesheim, 2021

[CE21] M. G. Casado, H. Eichelberger, Industry 4.0 Resource Monitoring - Experiences with Micrometer and Asset Administration Shells, Symposium on Software Performance 2021, EUR Workshop proceedings

[Cep23] J.-H. Cepok, Projektarbeit, Uni Hildesheim, 2023

[Eic16] H. Eichelberger, A Matter of the Mix: Integration of Compile and Runtime Variability, *Workshop on Dynamic Software Product Lines, FAS'16*, 2016.

[IVML] 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

[ESA+25] H. Eichelberger, C. Sauer, A. S. Ahmadian, C. Kröher, Industry 4.0/IIoT Platforms for manufacturing systems — A systematic review contrasting the scientific and the industrial side, Journal of Information and Software Technology (IST), volume 179, 107650, https://doi.org/10.1016/j.infsof.2024.107650, 2025

[ESS22] H. Eichelberger, H. Stichweh, C. Sauer, Requirements for an AI-enabled Industry 4.0 Platform – Integrating Industrial and Scientific Views, SOFTENG'22, pp. 7-14, 2022

[EN23] H. Eichelberger, C. Niederée, Asset Administration Shells, Configuration, Code Generation: A power trio for Industry 4.0 Platforms, ETFA'23, pp. 1-8, IEEE, 2023

[EPR+22] H. Eichelberger, G. Palmer, S. Reimer, T. Trong Vu, H. Do, S. Laridi, A. Weber, C. Niederée, T. Hildebrandt, Developing an AI-enabled IIoT platform - Lessons learned from early use case validation, SASI4'22 @ ECSA'22, 2022

[EPN22] H. Eichelberger, G. Palmer, C. Niederée, Developing an AI-enabled Industry 4.0 platform - Performance experiences on deploying AI onto an industrial edge device, Symposium on Software Performance (SSP'22), 2022

[ESA+21] 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, 2021

[EQS+16] H. Eichelberger, C. Qin, R. Sizonenko, K. Schmid, Using IVML to Model the Topology of Big Data Processing Pipelines In Proceedings of the International Systems and Software Product Line Conference SPLC'16, p. 204 – 208, 2016.

[EW25] H. Eichelberger, A. Weber, J. Hildebrand, ADS Performance Revisited, Softwaretechnik-Trends Band 45, Heft 1, 2025

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

[IDTA 02003-1-2] IDTA 02003-1-2 Generic Frame for Technical Data for Industrial Equipment in Manufacturing (https://industrialdigitaltwin.org/wp-content/uploads/2022/10/IDTA-02003-1-2_Submodel_TechnicalData.pdf)

[IDTA 02004-1-2] IDTA 02004-1-2 Handover Documentation (https://industrialdigitaltwin.org/wp-content/uploads/2023/03/IDTA-02004-1-2_Submodel_Handover-Documentation.pdf)

[IDTA 02011-1-0] IDTA 02011-1-0 Hierarchical Structures enabling Bills of Material (https://industrialdigitaltwin.org/wp-content/uploads/2023/04/IDTA-02011-1-0_Submodel_HierarchicalStructuresEnablingBoM.pdf)

[IDTA 2023-01-24] IDTA 2023-01-24 Product Carbon Footprint (https://github.com/admin-shell-io/submodel-templates/blob/main/published/Carbon%20Footprint/0/9/IDTA%202023-0-9%20_Submodel_CarbonFootprint.pdf)

[IDTA 02008-1-1] IDTA 02008-1-1 Time Series Data (https://industrialdigitaltwin.org/en/wp-content/uploads/sites/2/2023/03/IDTA-02008-1-1_Submodel_TimeSeriesData.pdf)

[IDTA 02002-1-0] IDTA 02002-1-0 Submodel for Contact Information (https://industrialdigitaltwin.org/wp-content/uploads/2022/10/IDTA-02002-1-0_Submodel_ContactInformation.pdf)

[IDTA 02007-1-0] IDTA 02007-1-0 Nameplate for Software in Manufacturing (https://industrialdigitaltwin.org/wp-content/uploads/2023/08/IDTA-02007-1-0_Submodel_Software-Nameplate.pdf)

[IDS] International Data Spaces, IDS reference architecture model version 3.0, https://internationaldataspaces.org/22m-3-0/

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

[KGR20] H. Koziolek, S. Grüner, J. Rückert, A Comparison of MQTT Brokers for Distributed IoT Edge Computing, ECSA, 2020

[LNI40] 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

[MBB+18] E. Maleki, F. Belkadi, N. Boli, J. van der B. Zwaag, K. Alexopoulos, S. Koukas, M. Marin-Perianu, A. Bernard, D. Mourtzis, Ontology-Based Framework Enabling Smart Product-Service Systems: Application of Sensing Systems for Machine Health Monitoring, IEEE Internet of Things Journal, 5 (6), pp. 4496-4505, 2018

[NE25] C. Nikolajew, H. Eichelberger, Industry 4.0 Connectors - A Performance Experiment with Modbus/TCP, Softwaretechnik-Trends Band 45, Heft 1, 2025

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

[Pid21] D. Pidun, Geräteverwaltung von IoT-Geräten für die IIP-Ecosphere Plattform, BSc-Abschlussarbeit, Universität Hildesheim, 2021

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

[SEA+20] 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, 2020

[SEK21] K. Schmid, S. El-Sharkawy, C. Kröher, Improving Software Engineering Research through Experimentation Workbenches. arXiv e-prints, arXiv-2110, 2021

[SE15] K. Schmid, H. Eichelberger, EASy-Producer: From Product Lines to Variability-rich Software Ecosystems, SPLC' 15, 2015

[Sch23] L. Schulz, Container-Virtualisierung mit LXC in der IIP-Ecosphere-Plattform, BSc Arbeit, Uni Hildesheim, 2023

[Sta20] M. Staciwa, Experimentelles Container-Deployment auf Industrie 4.0 Geräte, Projektarbeit, Uni Hildesheim, 2020

[Sta22] M. Staciwa, Modell-basierte Erstellung von containervirtualisierter Industrie 4.0 Anwendungen am Beispiel der IIP-Ecosphere-Plattform, Bachelorarbeit, Uni Hildesheim, 2022

[SSE21] 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, 2021

[ZVEI-N] ZVEI, Specification Submodel Templates of the Asset Administration Shell – ZVEI Digital Nameplate for industrial equipment (Version 1.0), https://www.plattform-i40.de/IP/Redaktion/DE/Downloads/Publikation/Submodel_Templates-Asset_Administration_Shell-digital_nameplate.html