Architecture Blueprints to Enable Scalable Vertical Integration of Assets with Digital Twins

Frank Schnicke
Virtual Engineering
Fraunhofer IESE
Kaiserslautern, Germany
0000-0002-4351-4488

Ashfaqul Haque
Virtual Engineering
Fraunhofer IESE
Kaiserslautern, Germany
a_haque19@cs.uni-kl.de

Thomas Kuhn
Embedded Systems
Fraunhofer IESE
Kaiserslautern, Germany
0000-0001-9677-9992

Daniel Espen
Virtual Engineering
Fraunhofer IESE
Kaiserslautern, Germany
0000-0001-5317-9516

Pablo Oliveira Antonino
Virtual Engineering
Fraunhofer IESE
Kaiserslautern, Germany
0000-0002-9631-8771

2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA) | 978-1-6654-9996-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/ETFA52439.2022.9921728

***Abstract*—Many Industry 4.0 use cases require the integration of live data, e.g., from sensors and devices. However, the large number of legacy fieldbus protocols and proprietary data formats turns this integration into an effort-consuming task. As the number of digital twins in a factory increases rapidly, data source integration has to scale well. Until now, little guidance is available on how to implement this integration in a scalable and reusable manner for Industry 4.0. To close this gap, we define five architecture blueprints based on our experience in various Industry 4.0 projects. These blueprints detail various integration scenarios differentiated by key attributes like frequency of data consumption and data production. In these architecture blueprints, two core components, the *Updater* and the *Delegator*, are identified. By providing and evaluating our open-source implementation of these two components, we show the feasibility of the defined blueprints. Utilizing the provided open-source components and the defined architecture blueprints will benefit practitioners as well as researchers when it comes to data integration with digital twins.

Keywords—Digital Twins, Vertical Integration, Scalability, Asset Administration Shell, IoT, Industry 4.0, Architecture

I. Introduction

The digitalization of production, which is referred to as Industry 4.0, promises a plethora of benefits, such as process transparency, more resilient supply chains, or the efficient production of small lot sizes [1]. A common use case for digitalization is to obtain a holistic view on the shop floor that includes all relevant data. The digital twin (DT), which originated from the testing of avionics systems [2], is a key concept to support this; it represents all properties and services of all relevant assets with unified interfaces.

DTs describe all relevant assets in a digitalized production. Relevant assets include, e.g., products to be produced, workpieces, devices, certificates, or the manufacturing process. As defined by the IIC [3], *"a digital twin is a formal digital representation of some asset, process or system that captures attributes and behaviors of that entity suitable for communication, storage, interpretation or processing within a certain context"*. Thus, in the context of this paper, a DT provides access to asset data and services. Additionally, simulation models for predicting system behavior and 3D models complement the digital representation.

This requires digital twins to be integrated with real (physical and non-physical) assets. DTs must be provided live access, e.g., to sensor data in order to identify optimization potential in the manufacturing process. Especially when creating DTs for older devices, this is a challenge, due to the large variety of protocols and bus systems that need to be integrated. Common examples include the Modbus, which was first implemented in 1979, as well as state-of-the-art automation protocols like OPC UA. Developers need to consider both asynchronous communication (e.g., MQTT, OPC UA Pub/Sub) and synchronous communications (e.g., HTTP/REST). Furthermore, different protocols use different message encodings for their payload, for instance JSON, XML, or binary formats.

Despite the need to integrate legacy devices and sensors, little guidance is available on how to perform this integration in a scalable and reusable manner. To bridge this gap, we formulated architecture blueprints based on past Industry 4.0 projects to support practitioners in integrating their DTs with real-world assets more efficiently. These blueprints conserve our experiences and best practices with respect to the integration of legacy devices with DTs. In this paper, we document five architecture blueprints and classify them according to key properties of Industry 4.0 data integration scenarios, e.g., frequency of data consumption and production. From our architecture blueprints, we identified the *Updater* and the *Delegator* components as core integration components, which implement push and pull semantics for connecting native devices and data sources. We implemented both components with available open-source software for Industry 4.0 digital twins (Eclipse BaSyx) and data integration (Apache Camel), and contribute both components as open-source components in the context of Eclipse BaSyx. To support system architects with future integration projects, we evaluated both components in different contexts using varying data sizes and discuss resulting scalability issues and processing loads.

This paper is structured as follows: Section II provides an overview of Industry 4.0 and digital twins, different scalability measures, as well as quality-drive specification of architecture drivers. In Section III, we describe two Industry 4.0 projects in which we were involved as motivation for the subsequent blueprints. Based on these project experiences, we derive architecture blueprints and present them in Section IV. Additional, we classify them by the relevant dimensions of data integration scenarios, e.g., data creation frequency. The presented architecture blueprints are quantitatively evaluated & discussed in Section V using an open-source implementation that utilizes the Asset Administration Shell (AAS) and Apache Camel. Section VI presents an overview of related work, and in Section VII, we draw conclusions and give an outlook on potential future work.

II. State of the Art & State of the Practice

A. Industry 4.0 and digital twins

Industry 4.0 (I4.0) is the digital transformation, i.e., the end-to-end digitalization of the manufacturing industry. While manufacturing devices already collect large amounts of data from different sources, this data is only available through native interfaces. In the automation pyramid (cf. Fig. 1), this corresponds to the programmable logic controller (PLC) layer. Data use is therefore limited to one machine.

In contrast, I4.0 advocates a peer-to-peer architecture that allows interaction across layers, e.g., to enable enterprise resource planning (ERP) systems to directly access sensors for tracking the quality of manufacturing processes. In our projects, we have had to use legacy protocols and use machine-specific encodings to realize DTs and to enable cross-layer communication for I4.0 [4].

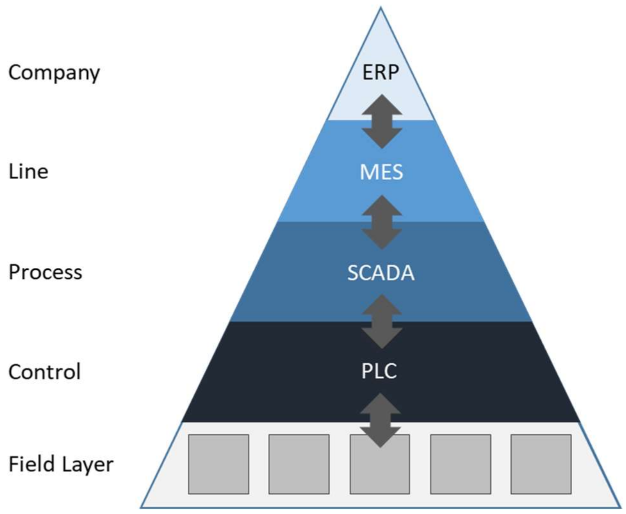

Figure 1 – The architecture of the automation pyramid

B. Vertical vs. horizontal scalability

The more complex Industry 4.0 and (I)IoT use cases require the integration of multiple data sources, which quickly yield larger quantities of data to be integrated. In consequence, scalability has become one of the most important requirements regarding fulfillment of the goals of Industry 4.0. Two types of scalability need to be considered here: vertical and horizontal scalability.

Vertical scalability is a system's ability to increase its capabilities by adding more resources to an existing system [5]. For instance, this means running an application in a device with better memory and computing power. Horizontal scalability describes a system's ability to increase its capabilities by increasing the number of devices that perform a specific task [5]. Common examples of horizontal scalability are load balancers, which balance the load between devices. Vertical scalability is often limited by the system itself; for example, the amount of computation and memory resources that can be added to a single system is often limited. Horizontal scalability is more flexible, but requires consideration in system architectures.

C. Architecture of plant automation

The dominant system architecture in manufacturing plants reflects the "automation pyramid" reference architecture as defined by IEC 62264 [6] (cf. Figure 1). It consists of separate layers with defined interaction points. PLCs execute cyclic programs to implement specific production steps. Supervisory control and data acquisition (SCADA) realizes the distributed control of several PLCs. Manufacturing Execution Systems (MES) manage the overall production by controlling high-level manufacturing steps. Enterprise Resource Planning (ERP) systems manage resources like the number of items in stock. The layered structure of the automation pyramid is a consequence of the large number of protocols and proprietary data formats that are used. As explicit adapters need to be created between any pair of communicating systems, a layer often only accesses information that is provided by adjacent layers. Creating additional information flows consumes a high amount of effort. Most existing and ready-to-use I4.0 solutions therefore often use explicit edge devices to collect data for analysis backends. While this works for isolated applications, like the creation of dashboards and predictive maintenance, it only yields more isolated data. In consequence, data is kept separate from assets, and the layers of the automation pyramid remain intact. Larger process changes still require significant modifications in all layers, and the integration of new backend applications might even require the installation of new edge devices for data collection.

A proper I4.0 architecture therefore needs to support cross-device and cross-application integration for data and device access that breaks the strict layering. This requires harmonized communication endpoints, mutual understanding of data, and the ability to bridge communication protocols. The *Industrial Internet Reference Architecture* [7] is not only a reference architecture for I4.0, but also addresses other domains like energy, smart cities, and healthcare. It presents the architectural viewpoints of business, usage, functionality, and implementation. The *Reference Architecture Model Industry 4.0* [8] defines a three-dimensional model decomposed into hierarchy levels, lifecycle and value stream, and layers. It is a framework for creating a common understanding among stakeholders and for identifying gaps and solutions for production systems. The *Stuttgart IT Architecture for Manufacturing* [9] decomposes its architecture into three middlewares: integration, mobile, and analytics, with a focus on service-oriented architectures.

III. Data Integration Scenarios

As a basis for the development of the architecture blueprints, our experience from various Industry 4.0 integration projects that applied I4.0 to industrial contexts was considered. In the following, two of these projects will be presented and their respective integration will be detailed.

A. Improving product quality through continuous monitoring

DigiPro4BaSys, as presented by Kannoth et al. [10], focuses on improving the production quality of a contract manufacturer that specializes in coating and lamination of technical textiles. To achieve the goal of improving product quality, multiple data sources needed to be made available in their respective digital twins and displayed in a comprehensive manner to enable holistic process analysis.

At the contract manufacturer's plant, textiles are moved through a powder-coating system and sprinkled with a thermal fusion adhesive. Next, the textiles are heat-treated by gas burners. Finally, the textiles are stored to cool down.

The devices involved in the aforementioned process include a variety of sensors and actors, like gas burners, motors for rotating the coils, and temperature sensors. The production assets provided data in a variety of protocols and data formats like Modbus, or via explicit pins.

Therefore, the variety of protocols and data formats needed to be integrated with the DT. For this, (i) data needed to be acquired from the different data sources, (ii) transformed into a data format appropriate for the chosen digital twin format, and (iii) pushed into the DT to enable third-party applications to access the device data.

Data integration was realized by first installing a gateway hardware that realized an OPC UA interface to Modbus data. In the next step, the OPC UA data was periodically sampled and integrated with the digital twin of the respective device. A simplified architectural overview of the data integration chain is shown in Figure 2. The permitted communication delay was about 1 second, and the data size was about 1 Kbyte.

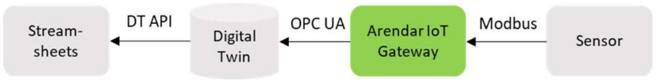

Figure 2 – Simplified architecture of DigiPro4BaSys data integration

B. Continuous optimization of quotations

BaSyPaaS, as also presented by Kannoth et al. [10], tackles the optimization of quotations for a contract manufacturer providing different lot sizes. The goal is for these quotations to be created based on experience regarding, e.g., tool wear and the respective cost of tool replacement.

However, tool wear data needs to be collected first. Thus, data from computer numerical control (CNC) machines and a handling robot needs to be made available in their respective digital twins. Similar to the project described in Section III.A, the production was first digitalized using an OPC UA gateway. In contrast to the cyclic integration of data in DigiPro4BaSys, event-driven integration using OPC UA Pub/Sub was chosen, which used the MQTT protocol. DTs received and processed the MQTT messages. The resulting simplified architecture overview is shown in Figure 3.

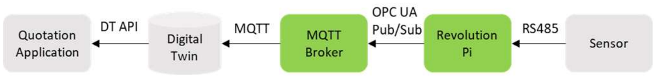

Figure 3 – Simplified architecture of BaSyPaaS data integration

IV. Scalable Data Integration Blueprints

Based on the project experiences described in Section III, multiple data integration blueprints were extracted. Each blueprint contains three core elements:

- *Sensor*: Produces data
- *DT Server*: Provides the DT of the sensor (e.g., its live data)
- *DT Application*: Accesses data provided by the DT server

The blueprints are segregated along the orthogonal dimensions of integration scenarios:

- Frequency of sensor data creation
- Frequency of sensor data consumption by the application
- Amount of data produced by the sensor

For the frequency aspect, we distinguish between high and low frequencies. For the data amount aspect, we distinguish whether the amount of data can be stored as part of the DT (i.e., little data) or whether it is too much (i.e., much data) and requires external storage.

Table 1 provides an overview of the relevant combinations of the aforementioned dimensions covered by the architecture blueprints. An "*" indicates that the characteristic of the specific dimension does not matter for the specific blueprint. The described blueprints enable scalability of the system by scaling horizontally. For instance, if a new sensor has to be integrated, an appropriate blueprint can be instantiated for this sensor without having to consider the already existing integrations. Additionally, if a parameter of the integration scenario changes, integration blueprints can be exchanged arbitrarily without any changes on the application side because the integration patterns are transparent for the application itself.

Table 1 – Dimensions mapped to blueprints

Data Consumption Frequency	*Data Creation Frequency*	*Amount of Data*	*Respective Blueprint*
Low	*	Little	IV.A
High	Low	Little	IV.B
*	Low	Much	IV.C
High	High	Little	IV.D
High	*	Much	IV.E

A. Low frequency of data consumption with little data

If the application is consuming data with low frequency and the amount of the data itself is little, a delegation approach as shown in the blueprint in Figure 4 is viable. Here, the *DT Server* delegates requests on DT properties to the *Delegator* component. The *Delegator* actively queries the *Sensor*, transforms the data if required, and returns the data to the *DT Server*, which in turn returns the data to the *DT Application*. In consequence, the *Delegator* retrieves data on-demand. The delegation is handled transparently, meaning the *DT Application* is not aware of the delegation mechanism's existence. Instead, it utilizes the DT access API provided by the *DT Server*.

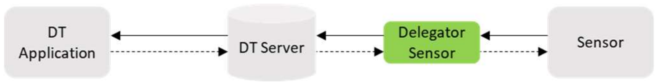

Figure 4 – Data integration via delegation (dashed: control flow, solid: data flow)

B. *High frequency of data consumption, low frequency of data production with little data*

If the data is consumed at a higher frequency than it is produced, for instance when multiple *DT Applications* are accessing the same value in the *DT Server*, delegation is not viable. Due to the high access frequency, the delegation mechanism would most likely overwhelm the network infrastructure. Instead, cyclic data retrieval is more appropriate.

Thus, the *Updater* component realizes the bridge between *Sensor* and *DT Server*. Similar to the *Delegator* component, it queries data and transforms it, if required. However, in contrast to the *Delegator*'s on-demand data acquisition, it cyclically polls data and pushes it to the *DT Server* after transformation.

As a consequence, data consumption of the *DT Application* and data production of the *Sensor* are decoupled. Thus, network load south of the *DT Server* is reduced. However, the frequency of the cyclic update needs to be appropriate for the *Sensor*.

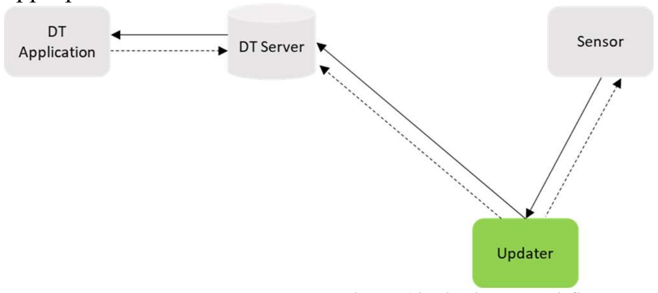

Figure 5 – Data integration via Updater (dashed: control flow, solid: data flow)

C. *Low frequency of data consumption with much data*

If it is not feasible to store the amount of data being produced by the *Sensor* in the *DT Server*, intermediate storage is required. In Figure 6, this storage is represented by the *Efficent Storage*.

In this scenario, simultaneous deployment of the *Updater* and the *Delegator* component is needed.

In contrast to the blueprint described in Section IV.B, in this blueprint the *Updater* pushes data into the *Efficient Storage* instead of into the *DT Server*. For data retrieval, the *Delegator* component is used, similar to Section IV.A. However, instead of delegating to the *Sensor* directly, the call is delegated to the *Efficient Storage*. In consequence, data access is handled transparently while data generation and access are decoupled at the same time.

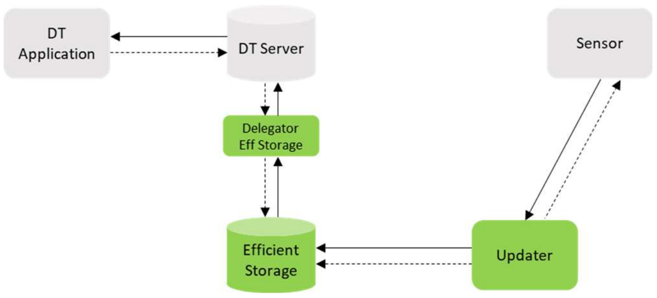

Figure 6 – Data integration via Updater & Delegator (dashed: control flow, solid: data flow)

D. *High frequency of data consumption, high frequency of data production with little data*

If both data consumption and data production are of high frequency with little data being produced, the blueprint shown in Figure 7 is required. Due to the high frequency of production and consumption, moving data through the *DT Server* may lead to bottlenecks. Thus, the data produced by the *Sensor* is made available on a message bus (e.g., MQTT). The *Updater* component described in Section IV.B is used to retrieve the data from the *Sensor,* transform it, and publish it on the message bus. If the *Sensor* already provides the data in the expected format via the expected message bus, the *Updater* component is not needed.

The DT contained in the *DT Server* provides a reference to the message bus (e.g., URL, topic) as well as a semantic description of the data being published on the bus. Thus, the *DT Application* can read the *DT Server* and use the provided information for accessing and processing the *Sensor* data. However, this requires the *DT Application* to communicate via the native event bus protocol, which exists outside the common DT access API. To alleviate this issue, the DT stored in the *DT Server* can provide a driver, e.g., as Java code, in addition to the access information. Thus, the *DT Application* can load the drivers necessary for event bus access at runtime. This means that there is no need to consider the specific *Event Bus* protocol at development time.

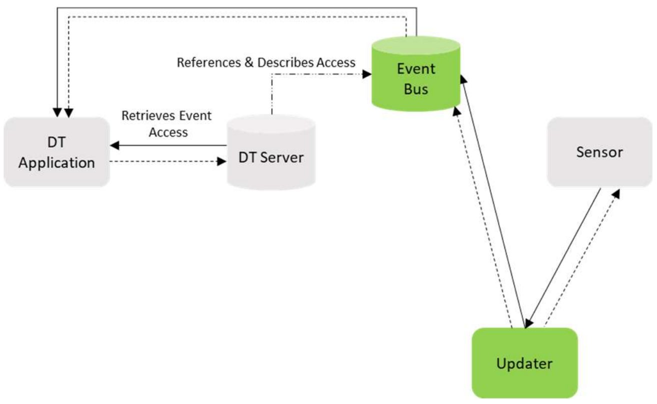

Figure 7 – Data integration via message bus (dashed: control flow, solid: data flow)

E. *High frequency of data production with much data*

In the case of high frequency of data production with much data, the blueprint of Section IV.D is combined with the blueprint described in Section IV.C. Instead of publishing the data itself over the message bus, only an update event is sent. In addition to sending the update event, the *Updater* component pushes the data into the *Efficient Storage*. Thus, the DT application is notified via an event that new data exists and can then retrieve it from the *Efficient Storage*. The overall blueprint is showcased in Figure 8.

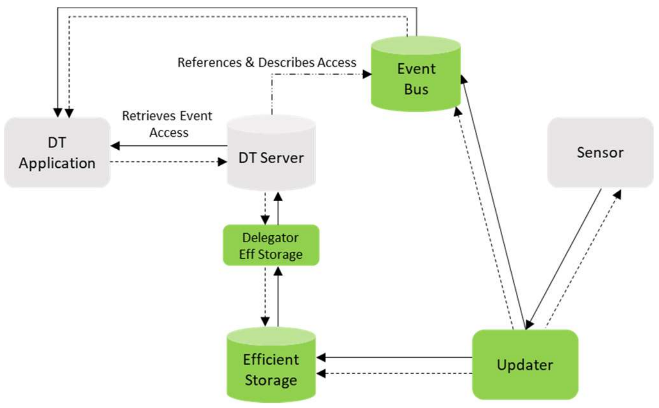

Figure 8 – Data integration via message bus and efficient storage (dashed: control flow, solid: data flow)

V. Implementation & Evaluation

For evaluation purposes, we implemented the core components of the data integration, i.e., the *Updater* and the *Delegator* component, as described in Section V.A. Since each blueprint is composed of one or both of these components, the evaluation focused on their performance. We evaluated both components using a variety of data sources, e.g., Apache Kafka or ActiveMQ. The evaluation results are shown Section V.B. and discussed in Section V.C.

A. Implementation

For the implementation of the blueprints, Eclipse BaSyx[1] realizing the Asset dministration AShell (AAS) was chosen as DT technology. In combination with BaSyx, Apache Camel [2] was used to implement the *Updater* and the *Delegator* component. Apache Camel enables efficient processing of data in a pipelined fashion, thus delivering core functionality
By utilizing Eclipse BaSyx and Apache Camel, the DataBridge was realized and is provided as open source[3].

B. Evaluation

The implemented *Updater* and *Delegator* components were evaluated in various test scenarios to determine their fitness for real-world applications. Figure 9 shows one example setup used for the evaluation. Here, Apache Kafka was used for data production while JSONata[4] was used for data transformation.

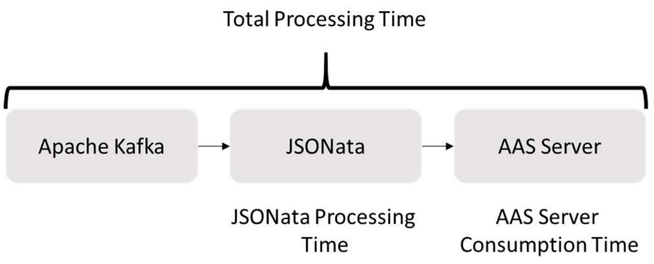

Figure 9 – Evaluation of integration scenario example

For the evaluation, a time series database, InfluxDB, was used to store the metrics generated from the *Updater* and *Delegator* components. The data was then visualized using Grafana. The results were evaluated based on the message execution time of each node in a Camel route and the total server response time of a message. The metrics were collected using a computer with the following hardware/software specifications: operating system: Windows 10, processor: Intel Core i5-1035G1 (1.00-3.60 GHz) 8 cores, 16 GB RAM, Intel UHD graphics, and 512 GB SSD hard drive. All modules used for collecting the evaluation results were run either in Docker or as a standalone software on this computer.
Four different data sizes of JSON messages were used for performance evaluation: 200 Bytes, 2 KB, 1 MB, and 2 MB. These values were chosen to cover a broad range of use cases in the interval of small data sizes, e.g., a temperature sensor reporting a temperature, and high data sizes, e.g., cameras providing high-definition images.
For the data source, ActiveMQ and Kafka were used. To test the *Delegator*, a synchronous dummy HTTP data source was used with the aforementioned message sizes.
For the *Updater* test cases of all message sizes, messages were sent to the *Updater* with an emission frequency of one message per 100 ms. The *Updater* component was run for five minutes to evaluate the message propagation time in a Camel route. The resulting chart for a 2 MB payload sent with Kafka is shown in Figure 10. For comparison, the same scenario with a 200 Byte payload is shown in Figure 11. The results for all data sources and message sizes are shown in Table 2.

Table 2 – Overview of Updater component performance using various data sources and payload sizes (time in ms)

	200 Byte		**2 KB**		**1 MB**		**2 MB**	
	Mean	Worst	Mean	Worst	Mean	Worst	Mean	Worst
ActiveMQ	2.65	3.92	2.32	4.36	57.7	94.2	89.30	127
Kafka	2.19	2.80	1.94	5.00	32.1	48.1	43.80	97.30

Similar to the *Updater* component, the *Delegator* component was evaluated for a HTTP/REST delegation. The results are shown in Figure 12 and Table 3.

[1] https://www.eclipse.org/basyx/
[2] https://camel.apache.org/
[3] https://github.com/eclipse-basyx/basyx-java-components/tree/feature/updater/basyx.components/basyx.components.updater
[4] Jsonata enables transformation of JSON payloads

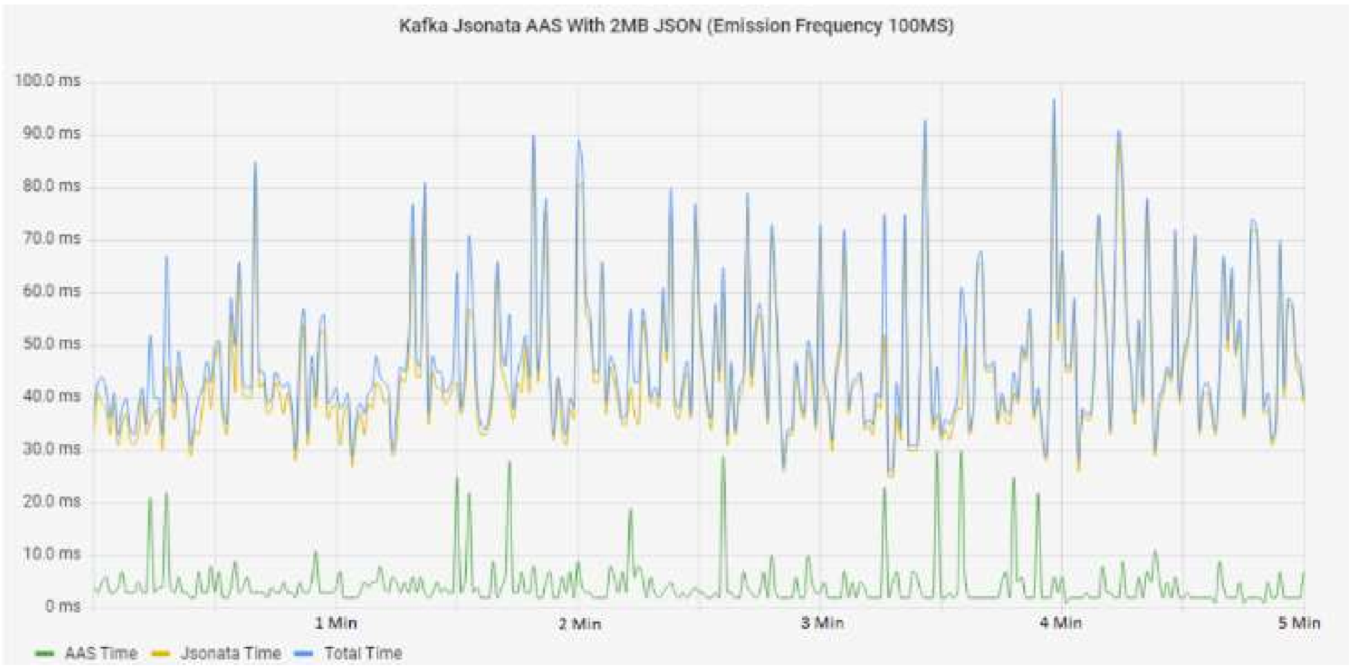

Figure 10 – Kafka integration route with 2 MB payload

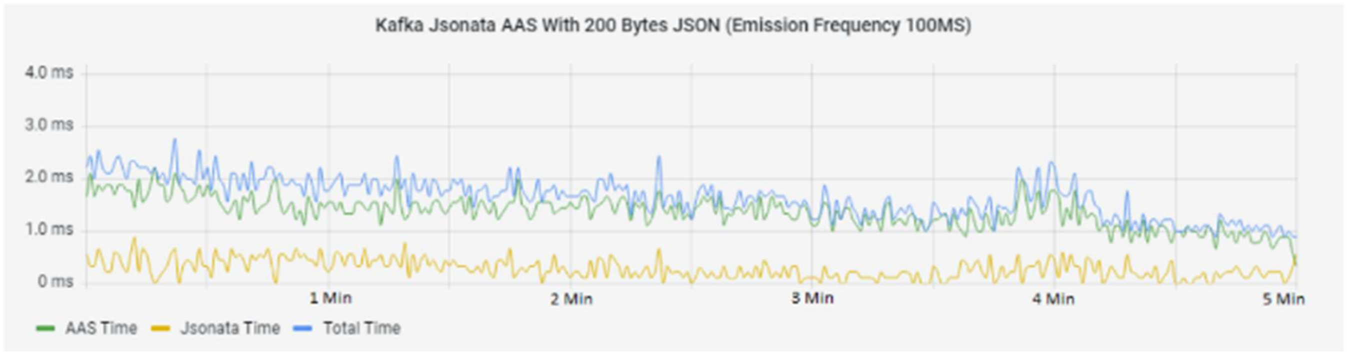

Figure 11 – Kafka integration route with 200 Byte payload

Table 3 – Overview of Delegator component performance using various payload sizes and HTTP/REST (time in ms)

	200 Byte	2 KB	1 MB	2 MB
Receiving Message	20.35	32.85	55.10	78.94
Total Response Time	44.20	54.15	82.95	107.35

C. *Discussion*

As can be seen in Table 2, the performance of the *Updater* component does not show a linear increase with respect to message size. With small messages (200 Bytes and 2 KB), the total response time did not make any difference. The increase from 2 KB to 1 MB did increase the data by a factor of 500, but processing time only increased by a factor of 50. Figure 11 shows that the response time was higher at the beginning than at the end of the evaluation. This can be attributed to route initialization. Gradually the response time became more stable. Some spikes, ups, and downs are seen in Figure 10, which are due to resource bottlenecks of the computer used. For instance, in the evaluations of 1 MB and 2 MB messages, CPU and memory usage of the computer rose to 100% and created a performance bottleneck for the *Updater*. This inconsistency should be mitigated in a production scenario with better resources.

In the performance evaluation, the total emission time of a message, i.e., the time needed by the message to be produced by the sensor, was not considered. Only the time of the route processing the messages is shown in the graphs. For faster message generation such as 50 ms, 20 ms, etc., message flow time will increase due to waiting time in the message queues of the data sources. This can be mitigated by integrating the data sources using concurrent consumers.
The results show that the *Updater* and *Delegator* components are reliable tools for processing messages. There were no cases of failed transport and the tools were able to process all messages successfully. With different message load, the response time of each message increased but was still within an acceptable interval. It could also be observed that the response time of the *Delegator* was low under different message loads.

VI. Related Work

There exists a plethora of related works regarding heterogeneous data source integration and scalable solutions in SOA, IoT, and Industry 4.0. In this section, we will present a brief overview of previous work on this topic. In the field of SOA, a scalable video conferencing system, GLOBALMMCS [11], has been proposed that uses microservices and a publish-subscribe JMS-based brokering

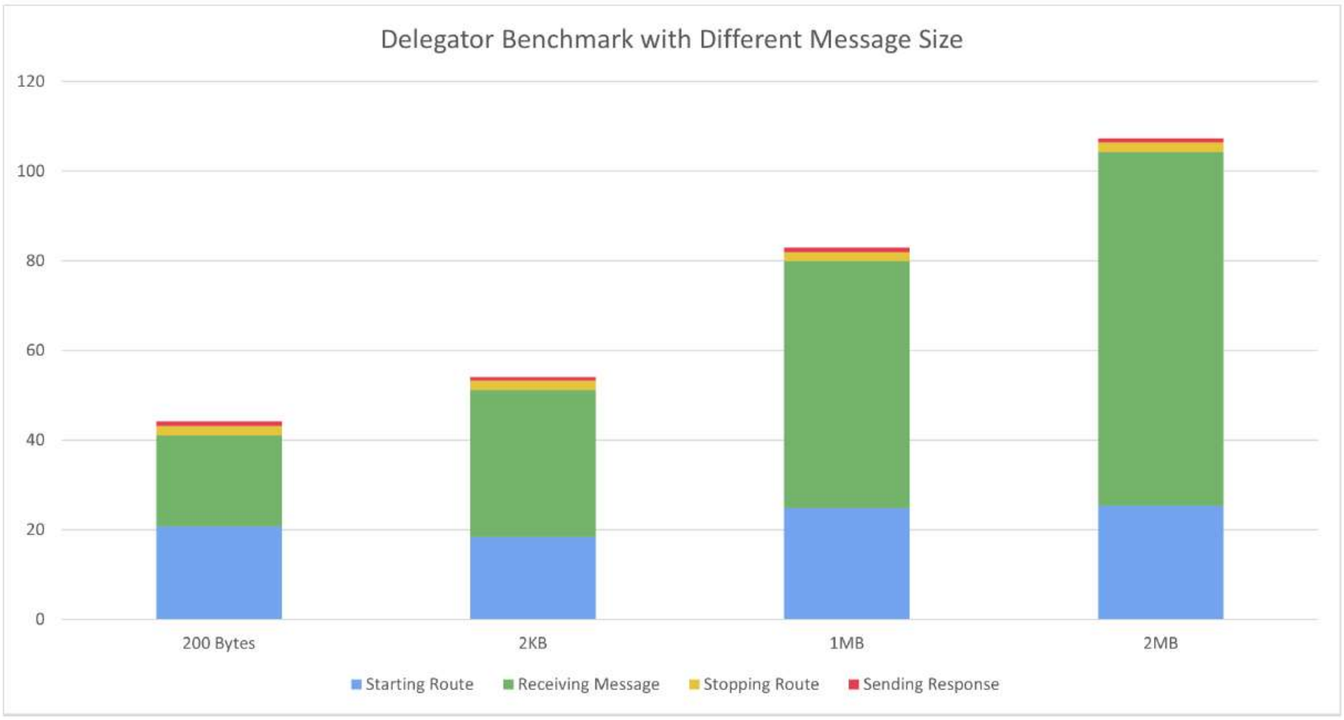

Figure 12 – Delegator benchmark for HTTP/REST (time in ms)

network. The advantages of the framework are fault tolerance, scalability, and location independence.

The authors of [12] propose a SOA cloud-based travel reservation software as a service using SOA and cloud. Here, the main architecture goal are distributed, visualized, parallel, and reliable computing mechanisms using multiple remote services offered by different travel service companies and a multi-tenancy architecture using messaging protocols. The authors of [13] propose an SOA-based scalable healthcare information system, presenting the composition of heterogeneous systems and addressing the important issue of system scalability. They describe a scalable architecture using SOAP. It presents a service registry where each service can be registered with multiple IP addresses. Load balancers can balance the loads between these instances. The system is divided into two parts, service consumers and service providers. The service providers are deployed to multiple addresses and are registered via a service registry. When a service consumer needs a service provider, it goes through the load balancer.

In the field of scalable IoT, the authors of [14] present an analysis and classification of service interactions using an IoT scenario for real-time ECG monitoring. They propose a system called EmoNet where four types of interactions between services are stated: direct service interaction (RPC or HTTP), indirect service interaction (using a service bus), event-driven interactions (using a messaging service), and exogenous service interactions (workflow coordination). The authors of [15] propose a distributed, interoperable architecture for IoT to overcome most of the obstacles in the process of large-scale expansion of IoT. It specifically addresses the heterogeneity of IoT devices and enables seamless addition of new devices across applications. The authors of [16] present MicroThings, a generic IoT architecture proposed to solve communication issues due to divergent protocols and standards. MicroThings integrates the application environment and the information aggregation environment with a logically centralized controller. Each of the two environments supports heterogeneous cooperation with a set of uniform interfaces. The authors of [17] propose IoT PAAS, a cloud-based IoT service delivery using cloud service PAAS. It inherits the multi-tenant character of the cloud to enable the concept of virtual verticals, as opposed to physically isolated vertical solutions. The system achieves scalability by its IoT Daemon, which combines multiple layers and functionalities.

In the field of Industry 4.0 and digital twins, the authors of [18] propose a scalable architecture using an open-source toolkit. This architecture provides real-time IoT connectivity and data acquisition, virtual representation and management, horizontal data pipeline connectivity (Kafka), time series data storage, data analytics, and visualization. To support digital twins for industrial process, the authors of [19] propose a scalable Kafka and Kepler fusion named Micro-Workflows. Scalability is achieved by redesigning monolithic workflows into sets of loosely coupled micro-workflows as independent computational services that communicate via a streaming middleware. The authors of [20] present another microservice-based Kafka stream, DSL, for stateful stream processing of digital twins using an Apache Kafka stream as a central nervous system of data processing between several microservices. For the management of product lifecycle data, the authors of [21] propose another scalable digital twin prototype. Scalability is achieved by using Kafka streams as a message broker and elastic search for data storage, search, and analytics.

Bellini et al. [22] describe the high-level control of a chemical plant via the Snap4Industry architecture. Here, vertical integration is achieved by utilizing IoT brokers as well as OPC UA and connecting existing systems like SCADA to them.

Kassner et al. [23] propose an architecture with a focus on data-driven manufacturing called Stuttgart IT Architecture for Manufacturing (SITAM). The architecture vertically integrates systems by proposing hierarchically arranged Enterprise Service Buses.

The concept of the Manufacturing Service Bus (MSB) for vertical integration is proposed by Schel et al. [24]. The MSB is divided into five layers and utilizes asynchronous communication patterns.

VII. Conclusion and Future Work

In this paper, we presented five architecture blueprints for the scalable integration of heterogeneous data sources derived from our experience in past projects. We classified the different blueprints according to the three dimensions of data integration scenarios: *data creation frequency*, *data consumption frequency*, and *amount of data* being sent.

In these blueprints, we identified the *Delegator* and the *Updater* components as core components for the integration of devices/native data sources. To support the choice of a blueprint from a scalability viewpoint, we evaluated both components quantitatively based on our open-source implementation and discussed the findings. Researchers can build on and extend our architecture blueprints using the aforementioned open-source implementation for further experiments.

Future work by the authors will integrate the configuration of the *Updater* and the *Delegator* components as well as the overall blueprint configuration with the Asset Administration Shell and thus with digital twins. For example, the Asset Interface Submodel being specified by the Industrial Digital Twin Association (IDTA)[5] is being considered.

Acknowledgment

This work is partially supported by the German Federal Ministry of Education and Research in the scope of the BaSys 4.2 project (01IS19022) and by the German Federal Ministry for Economic Affairs and Climate Action in the scope of the FabOS project (01MK20010A).

References

[1] Kiel, D., Müller, J. M., Arnold, C., & Voigt, K. I. (2020). Sustainable industrial value creation: Benefits and challenges of industry 4.0. In *Digital Disruptive Innovation* (pp. 231-270).

[2] Glaessgen, E., & Stargel, D. (2012, April). The digital twin paradigm for future NASA and US Air Force vehicles. In 53rd AIAA/ASME/ASCE/AHS/ASC structures, structural dynamics and materials conference 20th AIAA/ASME/AHS adaptive structures conference 14th AIAA (p. 1818).

[3] Definition of Digital Twin, https://www.iiconsortium.org/pdf/IIC_Digital_Twins_Industrial_Apps_White_Paper_2020-02-18.pdf, last accessed 2022/05/03

[4] Rosen, R., Von Wichert, G., Lo, G., & Bettenhausen, K. D. (2015). About the importance of autonomy and digital twins for the future of manufacturing. *Ifac-papersonline*, *48*(3), 567-572.

[5] Gupta, Anisha, Rivana Christie, and R. Manjula. "Scalability in internet of things: features, techniques and research challenges." Int. J. Comput. Intell. Res 13.7 (2017): 1617-1627.

[6] International Electrotechnical Commission. 2003. "IEC 62264-1 Enterprise-control system integration–Part 1: Models and terminology." IEC, Genf (2003)

[7] Industrial Internet Consortium. 2015. "Industrial internet reference architecture." Technical Article. Available online: http://www. iiconsortium. org/IIRA. htm (accessed on 05 Mai 2020)

[8] DIN specification "91345: 2016-04 (2016) Reference Architecture Model Industrie 4.0 (RAMI4.0)

[9] Kassner, L., Gröger, C., Königsberger, J., Hoos, E., Kiefer, C., Weber, C., ... & Mitschang, B. (2016, April). The Stuttgart IT architecture for manufacturing. In *International Conference on Enterprise Information Systems* (pp. 53-80). Springer, Cham.

[10] Kannoth, S., Hermann, J., Damm, M., Rübel, P., Rusin, D., Jacobi, M., ... & Antonino, P. O. (2021, September). Enabling SMEs to Industry 4.0 Using the BaSyx Middleware: A Case Study. In *European Conference on Software Architecture* (pp. 277-294). Springer, Cham.

[11] Uyar, A., Wu, W., Bulut, H., & Fox, G. (2005, July). Service-oriented architecture for a scalable videoconferencing system. In *ICPS'05. Proceedings. International Conference on Pervasive Services, 2005.* (pp. 445-448). IEEE.

[12] Namjoshi, J., & Gupte, A. (2009, September). Service oriented architecture for cloud based travel reservation software as a service. In *2009 IEEE International Conference on Cloud Computing* (pp. 147-150). IEEE.

[13] Yang, T. H., Sun, Y. S., & Lai, F. (2011). A scalable healthcare information system based on a service-oriented architecture. *Journal of medical systems*, *35*(3), 391-407.

[14] Arellanes, D., & Lau, K. K. (2018, July). Analysis and Classification of Service Interactions for the Scalability of the Internet of Things. In *2018 IEEE International Congress on Internet of Things (ICIOT)* (pp. 80-87). IEEE.

[15] Sarkar, C., Nambi, S. A. U., Prasad, R. V., & Rahim, A. (2014, March). A scalable distributed architecture towards unifying IoT applications. In *2014 IEEE World Forum on Internet of Things (WF-IoT)* (pp. 508-513). IEEE.

[16] Shen, Y., Zhang, T., Wang, Y., Wang, H., & Jiang, X. (2017). Microthings: A generic iot architecture for flexible data aggregation and scalable service cooperation. *IEEE Communications Magazine*, *55*(9), 86-93.

[17] Li, F., Vögler, M., Claeßens, M., & Dustdar, S. (2013, June). Efficient and scalable IoT service delivery on cloud. In *2013 IEEE sixth international conference on cloud computing* (pp. 740-747). IEEE.

[18] Kamath, V., Morgan, J., & Ali, M. I. (2020, June). Industrial IoT and Digital Twins for a Smart Factory: An open source toolkit for application design and benchmarking. In *2020 Global Internet of Things Summit (GIoTS)* (pp. 1-6). IEEE.

[19] Radchenko, G., Alaasam, A. B., & Tchernykh, A. (2018, December). Micro-workflows: Kafka and kepler fusion to support digital twins of industrial processes. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)* (pp. 83-88). IEEE.

[20] Alaasam, A. B., Radchenko, G., & Tchernykh, A. (2019, October). Stateful stream processing for digital twins: Microservice-based kafka stream dsl. In *2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON)* (pp. 0804-0809). IEEE.

[21] Schranz, C., Strohmeier, F., & Damjanovic-Behrendt, V. (2020, November). A Digital Twin Prototype for Product Lifecycle Data Management. In *2020 IEEE/ACS 17th International Conference on Computer Systems and Applications (AICCSA)* (pp. 1-6). IEEE.

[22] Bellini, P., Cenni, D., Mitolo, N., Nesi, P., Pantaleo, G., & Soderi, M. (2022). High level control of chemical plant by industry 4.0 solutions. *Journal of Industrial Information Integration*, *26*, 100276.

[23] Kassner, L., Gröger, C., Königsberger, J., Hoos, E., Kiefer, C., Weber, C., ... & Mitschang, B. (2016, April). The Stuttgart IT architecture for manufacturing. In *International Conference on Enterprise Information Systems* (pp. 53-80). Springer, Cham.

[24] Schel, D., Henkel, C., Stock, D., Meyer, O., Rauhöft, G., Einberger, P., ... & Seidelmann, J. (2018). Manufacturing service bus: an implementation. *Procedia CIRP*, *67*, 179-184.

[5] https://industrialdigitaltwin.org/en/