



Hochschule für Technik und
Wirtschaft Dresden
University of Applied Sciences

Belegarbeit

Equipmentmonitoring mit AAS und EDC

Vorgelegt von:	Josia Rudolph
Studienbereich:	Informatik/Mathematik
Vorgelegt am:	28.02.2026
Ort:	Dresden
Matrikelnummer:	s88598
Erstgutachter:	Prof. Dr. rer. pol. Dirk Reichelt
Zweitgutachter:	Dipl.-Inf. Stefan Vogt, Dipl.-Wirt.-Inf. Paul Patolla

Inhaltsverzeichnis

Abkürzungsverzeichnis	IV
Abbildungsverzeichnis	V
Tabellenverzeichnis	VI
1 Einleitung	1
1.1 Motivation	1
1.2 Aufgabenstellung	2
1.2.1 Allgemein	2
1.2.2 Sommersemester 2025	2
1.2.3 Wintersemester 2025/26	3
1.3 Aufbau der Arbeit	4
2 Grundlagen	5
2.1 Halbleiterfertigung: Die Subfab und ihre Prozesse	5
2.2 Industrie 4.0 und der Digitale Zwilling	6
2.3 Die Verwaltungsschale (Asset Administration Shell) als Standard	7
2.4 Der Eclipse Dataspace Connector	8
3 Konzeption	10
3.1 Zielarchitektur	10
3.2 Herleitung des Simulationsansatzes	12
3.3 Auswahl des Implementierungs-Frameworks: Eclipse BaSyx	14
3.4 Entwurf des AAS-Modells	16
3.5 Konzeption des souveränen Datenraums	17
3.6 Konzeption der Client-Anwendung und Visualisierung	18
3.7 Architektonische Erweiterbarkeit für KI-gestützte Analyseverfahren	20
4 Implementierung	21
4.1 Aufbau der AAS-Laufzeitumgebung	21
4.2 Modellierung und Erstellung der Verwaltungsschalen	21
4.3 Realisierung und Anbindung des Drucksimulators	23
4.4 Bereitstellung der lokalen EDC-Infrastruktur	25
4.5 Die Client-Applikation	25
4.5.1 Autorisierung und Vertragsaushandlung	28
4.5.2 Datenpersistierung	28
4.5.3 Visualisierung	28

5 Tests und Ergebnisse	30
5.1 Automatisierte Bereitstellung des Demonstrators	30
5.2 Evaluation der souveränen Datenpipeline	30
5.2.1 Datengenerierung und -bereitstellung	30
5.2.2 Vertragsaushandlung und Datentransfer	32
5.2.3 Auswertung, Persistierung und Visualisierung:	32
6 Fazit und Ausblick	33
6.1 Zusammenfassung der Ergebnisse	33
6.2 Kritische Reflexion und methodische Einordnung	33
6.3 Ausblick und zukünftige Forschungsansätze	34
Literatur	35
A Struktureller Aufbau der Verwaltungsschale	37
B Docker-Compose für BaSyx Off-The-Shelf Komponenten	38
C Drucksimulator Python Script	41
D Setupscript zum starten des Demonstrators	50
E Logs des AAS Clients	52

Abkürzungsverzeichnis

API	Application Programming Interface
AAS	Asset Administration Shell
ABAC	Attribute Based Access Control
EDC	Eclipse Dataspace Connector
HTTP	Hypertext Transfer Protocol
IESE	Fraunhofer-Institut für Experimentelles Software Engineering
REST	Representational State Transfer
SDK	Software development kit
IDTA	Industrial Digital Twin Association
IDSA	International Data Spaces Association
CSV	Comma Separated Value
JSON	Java Script Object Notation
ERP	Enterprise Resource Planning
EDR	Endpoint Data Reference

Abbildungsverzeichnis

Abbildung 1 Vereinfachte Architektur der Datenübertragung	10
Abbildung 2 Datenfluss über den EDC-Datenraum	11
Abbildung 3 Modellierung Pumpen-Abatement-System	12
Abbildung 4 Bearbeiten der Verwaltungsschale mit dem AASX- Packageexplorer	23
Abbildung 5 Python Drucksimulator GUI	24
Abbildung 6 Klassendiagramm der Client-Applikation	27
Abbildung 7 Benutzeroberfläche des Simulators	31
Abbildung 8 Webansicht der Verwaltungsschale	31
Abbildung 9 Grafana - Visualisierung der Druckwerte	32

Tabellenverzeichnis

Tabelle 1 Matrix über Druckverhältnisse und mögliche Szenarien	13
Tabelle 2 Use-Cases für die Simulation	14

1 Einleitung

1.1 Motivation

Der Halbleiterproduktionsprozess stellt besonders hohe Anforderungen an seine Prozessumgebungen, weshalb Frontend Fabriken von Halbleiterherstellern als Reinraum aufgebaut sind. Sie bestehen aus drei Ebenen: dem Reinraum mit den Prozess- und Messmaschinen auf mittlerer Ebene, der Sub-Fab mit Maschinen und Anlagen zur Aufbereitung von Reinstwasser, Abgasreinigung und Maschinenmedienversorgung über und unter dem Reinraum.

Bei den Produktionsprozessen fallen verschiedene Abgase und Prozesschemikalien an, die nicht direkt in die Umgebung abgegeben werden dürfen. Die Abgase entstehen typischerweise in Prozesskammern (z.B. Ätz- oder Implantationsanlagen). Diese Gase müssen sicher aus dem Reinraum entfernt und in der SubFab in einem Abatement-System (Gasreinigungssystem) behandelt werden. Spezielle Vakuumpumpen saugen die Gase ab, da viele Prozesse unter Unterdruck stattfinden. Die Pumpen sind oft ebenfalls in den SubFab-Bereich verlagert, um Vibrationen und Partikelemissionen im Reinraum zu minimieren. Das Abatement-System reinigt die Abgase durch verschiedene Verfahren, abhängig von der Zusammensetzung der Schadstoffe (z.B. durch Thermische Oxidation oder per Nasswäscher). Nach der Reinigung werden die neutralisierten Abgase sicher in die Abluftanlage der Fabrik abgeleitet.

Die Zuverlässigkeit von Vakuumpumpen und Abatement-Systemen ist daher essenziell für stabile Produktionsprozesse. Unerkannter Verschleiß an Pumpen oder verstopfte Zuleitungen zum Abatement können die Produktion beeinträchtigen. Die Druckverhältnisse in beiden Systemen bestimmen die Effizienz des Abgastransports. Wartungsbedarfe frühzeitig zu erkennen, ermöglicht nicht nur Kosteneinsparungen, sondern minimiert auch ungeplante Stillstände. Durch einen kontinuierlichen Abgleich von Druckwerten (Solldruck der Pumpe, Druck am Ausgang der Pumpe und Druck am Abatement) lassen sich Rohrverschmutzungen erkennen und Rückschlüsse auf den Gesundheitszustand von Pumpen ziehen. Aktuell können diese Messwerte nur direkt an den Geräten in der SubFab abgelesen werden. Das bedeutet, dass Wartungstechniker regelmäßig vor Ort Messungen durchführen und Anomalien manuell identifizieren müssen. Diese zeitaufwändige reaktive Wartung erlaubt jedoch keine kontinuierliche, vorausschauende Diagnose (Predictive

Maintenance) mit optimierten Wartungsempfehlungen, wie es durch einen direkten Zugriff des Equipmentherstellers per Ferndiagnose möglich wäre.

Im Rahmen des FuE-Seminars soll untersucht werden, inwieweit strukturierte Daten aus dem Sub-Fab Bereich einer Halbleiterfabrik zur Remote-Überwachung von Prozessen genutzt werden können, um proaktive Wartungsempfehlungen zu geben. Hierbei steht die Integration von Digitalen Zwillingen auf Basis der AAS im Fokus. Die AAS ermöglicht eine standardisierte hierarchische Beschreibung von digitalen Zwillingen. Mit dem EDC können Daten, wie auch AAS, in einem gemeinsamen Datenraum zugänglich gemacht werden. Der Fokus bei dieser Form des Datenaustausches liegt auf der Souveränität der Daten und der damit verbundenen Kontrolle des Datenanbieters über seine Daten.

1.2 Aufgabenstellung

1.2.1 Allgemein

Die übergeordnete Zielsetzung dieser Forschungsarbeit ist die Integration und der souveräne Austausch von Equipmentdaten – spezifisch von Vakuumpumpen und Abatement-Systemen – zur kontinuierlichen Überwachung und Wartungsoptimierung in der Halbleiterfertigung. Der Forschungs- und Implementierungsaufwand dieses Vorhabens erstreckt sich über zwei konsequente Projektphasen, beginnend im Sommersemester 2025 bis zum Abschluss im Wintersemester 2025/2026. Die vorliegende Forschungsarbeit führt beide Erarbeitungsstände in einer Gesamtdarstellung zusammen. Im Kern soll untersucht werden, wie die **Asset Administration Shell (AAS)** als standardisierter Digitaler Zwilling und der **Eclipse Dataspace Connector (EDC)** für einen sicheren Datenaustausch genutzt werden können, um die informationstechnische Lücke zwischen dem Fabrikbetreiber und dem Equipmenthersteller zu schließen.

1.2.2 Sommersemester 2025

Die erste Phase dieses Forschungsvorhabens legt die technologische und konzeptionelle Grundlage für die gesamte Datenpipeline. Der Schwerpunkt liegt auf der Erstellung einer validen, standardisierten und dynamischen Datengrundlage. Dies umfasst zunächst eine fundierte Einarbeitung in die relevanten Technologiefelder der Halbleiterproduktion, der Asset Administration Shell und des Eclipse Dataspace Connectors.

Auf dieser Wissensbasis erfolgt die Konzeption und Erstellung von AAS-Modellen für eine Vakuumpumpe und ein Abatement-System. Die Modellierung soll dabei möglichst

umfangreich sein, um die Assets detailliert abzubilden. Ein zentrales Element ist die Erstellung eines Konzepts zur Bereitstellung der für die Wartungsanalyse entscheidenden Druckwerte. Konkret müssen die folgenden drei Werte über die Verwaltungsschalen zugänglich gemacht werden: der Solldruck der Pumpe, der tatsächlich erreichte Druck an ihrem Ausgang sowie der Druck am Eingang des nachgeschalteten Abatement-Systems. Die Differenz und das Verhalten dieser Werte zueinander bilden die informationstechnische Basis für die spätere Fehlerdiagnose.

Um eine realitätsnahe Datengrundlage für die Analyse zu schaffen, wird parallel ein Druck-Simulator entwickelt. Dessen Aufgabe ist die Modellierung und Validierung verschiedener Wartungsszenarien. Der Simulator muss das Druckverhalten in Abhängigkeit von Faktoren wie Verschmutzung, Durchfluss und Pumpenleistung abbilden können. Insbesondere sollen Szenarien für typische Störfälle wie Pumpenausfälle und Rohrverstopfungen modelliert werden. Dies schließt den normalen Betrieb, eine allmähliche Verschmutzung, eine plötzliche Blockade sowie einen abrupten Pumpenausfall oder Druckabfall mit ein. Die Implementierung dieser Komponenten und die Verfassung der vorliegenden wissenschaftlichen Dokumentation bilden den Abschluss dieser ersten Projektphase.

1.2.3 Wintersemester 2025/26

Aufbauend auf den technologischen Grundlagen der ersten Projektphase verlagert sich der Fokus im Wintersemester 2025/2026 auf den systemübergreifenden Datentransfer sowie den Aufbau einer nachgelagerten Client-Infrastruktur. Ein zentrales Ziel ist die Implementierung einer prototypischen Datenpipeline, welche den souveränen Datenaustausch zwischen der Halbleiterfabrik und dem Equipmenthersteller innerhalb eines verteilten Datenraums realisiert. Im Fokus steht dabei die Integration des **EDC** der als HTTP-Proxy fungiert, um die sichere Kommunikation sowie die notwendige Vertragsaushandlung informationstechnisch abzubilden.

Um die über den EDC bereitgestellten Verwaltungsschalen datenökonomisch zu verwenden, bedarf es der Entwicklung einer dedizierten Client-Anwendung. Diese Applikation hat die Aufgabe, die kontinuierlich aktualisierten Druckwerte aus der BaSyx-Umgebung über den Datenraum strukturiert abzurufen. Zur Gewährleistung der Nachvollziehbarkeit und für weiterführende Inspektionen sollen die extrahierten Zeitreihendaten persistiert und für eine Dashboard-gestützte, visuelle Aufbereitung zugänglich gemacht werden.

Gemäß der Zielsetzung bildet die Auswertung der generierten Druckdifferenzen den analytischen Abschluss dieser Projektphase, wofür perspektivisch eine KI-gestützte Mustererkennung avisiert ist. Um eine solche komplexe maschinelle Lernmethode methodisch sauber zu fundieren, bedarf es jedoch zunächst einer validierten Baseline. Der Schwerpunkt liegt in dieser Arbeit daher auf dem Entwurf und der Implementierung einer deterministischen, schwellwertbasierten Analyse-Schnittstelle innerhalb des Clients. Durch diese regelbasierte Auswertung sollen spezifische Anomalien und Fehlermuster grundlegend klassifiziert werden, um die prinzipielle informationstechnische Machbarkeit der prädiktiven Wartung auf Basis der verteilten Datenarchitektur zu validieren und den Proof of Concept für spätere KI-Modelle zu erbringen.

1.3 Aufbau der Arbeit

Die vorliegende wissenschaftliche Arbeit ist in sechs Kapitel gegliedert, die den Leser systematisch von der Problemstellung über die theoretischen Grundlagen und die Konzeption bis hin zur praktischen Implementierung und deren Überprüfung führen.

Kapitel 2 legt das theoretische Fundament zum Verständnis der Arbeit. Es werden zunächst die prozesstechnischen Besonderheiten der Halbleiterfertigung mit Fokus auf die Sub-Fab erläutert. Anschließend werden die Kernkonzepte von Industrie 4.0, des Digitalen Zwillings und eine Technologie für einen souveränen Datenaustausch vorgestellt. Aufbauend auf diesen Grundlagen wird in **Kapitel 3** das spezifische Lösungskonzept für die Aufgabenstellung entwickelt. Dies umfasst den Entwurf der informationstechnischen Zielarchitektur und den Aufbau der demonstrator Pipeline.

Kapitel 4 dokumentiert die praktische Umsetzung des entworfenen Konzepts. Die einzelnen Komponenten der Zielarchitektur werden hier detailliert beschrieben.

Die Verifikation der entwickelten Komponenten ist Gegenstand von **Kapitel 5**. Hier werden die durchgeführten Funktionalitätstests der einzelnen Komponenten der Pipeline dargelegt und die Ergebnisse präsentiert.

Abschließend fasst **Kapitel 6** die Arbeit zusammen, reflektiert die erzielten Ergebnisse im Kontext der ursprünglichen Aufgabenstellung und gibt einen Ausblick auf zukünftige Forschungsarbeiten.

2 Grundlagen

2.1 Halbleiterfertigung: Die Subfab und ihre Prozesse

Die moderne Halbleiterfertigung, wie sie beispielsweise im Halbleiterwerk von Bosch in Dresden betrieben wird, ist ein Paradebeispiel für eine hochautomatisierte, datengetriebene Produktionsumgebung im Sinne der Industrie 4.0. In diesen „Fabriken der Zukunft“ werden immense Datenmengen generiert, um Prozesse mittels künstlicher Intelligenz zu überwachen, zu steuern und zu optimieren. Ein zentrales Merkmal dieser Fertigungsstätten ist die strikte räumliche und funktionale Trennung zwischen dem Reinraum und der unterstützenden Infrastruktur. (Bosch, 2021)

Im Reinraum finden die hochsensiblen Kernprozesse der Chipherstellung statt. Dazu gehören unter anderem die Fotolithografie, Ätzprozesse sowie diverse Beschichtungs- und Depositionsverfahren. Der Erfolg dieser Schritte hängt von einer quasi perfekten Umgebung ab, die frei von Partikeln, Vibrationen und anderen Störeinflüssen ist. Aus diesem Grund wird die gesamte notwendige Versorgungsinfrastruktur in einen separaten Bereich, die sogenannte Sub-Fab, ausgelagert. (Hilscher, 2023)

Die Sub-Fab ist das maschinelle Rückgrat des Reinraums. Sie beherbergt eine Vielzahl prozesskritischer Aggregate, darunter Stromversorgungen, Kühl- und Heizsysteme sowie die für diese Arbeit zentralen Vakuumpumpen und Abgasreinigungssysteme (Abatement). Die hohe Dichte an energieintensiven Anlagen macht die Sub-Fab für einen erheblichen Teil, teils bis zu 40%, des Gesamtenergieverbrauchs einer Fabrik verantwortlich, was ihre betriebswirtschaftliche und ökologische Relevanz unterstreicht. (Nitzschner & Krauß, 2024)

Die Vakuumpumpen spielen eine entscheidende Rolle, da viele der im Reinraum ablaufenden Ätz- und Beschichtungsprozesse ein prozessspezifisches Vakuum oder definierte Niederdruckbedingungen erfordern. Diese kontrollierte Atmosphäre ist notwendig, um chemische Reaktionen zu ermöglichen und Kontaminationen zu verhindern. Die Pumpen evakuieren die Prozesskammern und fördern die dabei anfallenden gasförmigen Nebenprodukte ab. Ihre Unterbringung in der Sub-Fab ist essenziell, um die Übertragung von störenden Vibrationen auf die Präzisionsanlagen im Reinraum zu verhindern. (Hilscher, 2023)

Die abgepumpten Prozessgase sind häufig umwelt- oder gesundheitsschädlich und dürfen nicht direkt in die Atmosphäre gelangen. Sie werden daher den Abatement-Systemen

zugeführt, die ebenfalls in der Sub-Fab installiert sind. Diese Anlagen reinigen die Abgase durch thermische, chemische oder physikalische Verfahren, bevor sie sicher abgeleitet werden. (Nitzschner & Krauß, 2024)

Die zuverlässige Funktion der Kette aus Prozessanlage, Vakuumpumpe und Abatement-System ist für einen stabilen Produktionsablauf unerlässlich. Ein Ausfall in diesem Versorgungspfad kann zum sofortigen Stillstand der betroffenen Produktionsmaschinen führen. Die Analyse von Betriebsdaten dieser Aggregate zur vorausschauenden Wartung (Predictive Maintenance) und zur Erkennung von Anomalien ist daher ein zentraler Hebel zur Steigerung der Gesamtanlageneffektivität und zur Reduzierung ungeplanter Stillstände. (Hilscher, 2023)

2.2 Industrie 4.0 und der Digitale Zwilling

Die Vision der Industrie 4.0 zielt auf die Schaffung intelligenter, flexibler und weitgehend selbstorganisierter Produktionssysteme ab. Eine Grundvoraussetzung hierfür ist die lückenlose digitale Vernetzung aller an der Wertschöpfung beteiligten Akteure und Objekte – von der einzelnen Maschine bis zur gesamten Fabrik. (Plattform Industrie 4.0, 2023)

Der Digitale Zwilling ist ein umfassendes, virtuelles Abbild eines konkreten physischen Objekts oder sogar eines immateriellen Prozesses, das dynamisch mit seinem realen Gegenstück über dessen gesamten Lebenszyklus hinweg gekoppelt ist. Er ist somit weit mehr als nur ein statisches digitales Modell. Ein Kernaspekt des Digitalen Zwillings ist die Integration und Bündelung sämtlicher relevanter Informationen und Daten an einem zentralen, digital zugänglichen Ort. Dazu gehören sowohl statische Daten wie technische Spezifikationen und Dokumentationen als auch dynamische Betriebsdaten, die in Echtzeit von Sensoren aus der physischen Welt erfasst werden und den aktuellen Zustand des Assets widerspiegeln. (Krauß et al., 2023)

Die wahre Stärke und Bedeutung des Digitalen Zwillings für die Industrie 4.0 entfaltet sich durch seine Fähigkeit, nicht nur die Vergangenheit und Gegenwart eines Assets abzubilden, sondern auch dessen zukünftiges Verhalten zu simulieren. Durch die Einbettung von Verhaltensmodellen ermöglicht er die Analyse von Was-wäre-wenn-Szenarien, die virtuelle Inbetriebnahme von Anlagen oder die prädiktive Vorhersage von Wartungsbedarfen. Er wird somit zur informationstechnischen Grundlage, die es erlaubt, Produktionsprozesse zu überwachen, zu analysieren und zu optimieren, ohne direkt in den laufenden physischen Betrieb eingreifen zu müssen. Indem der Digitale Zwilling eine einheitliche

und herstellerübergreifende Datengrundlage schafft, löst er Inselsysteme auf und wird zum entscheidenden Wegbereiter für Interoperabilität und datengetriebene Wertschöpfung in der intelligenten Fabrik der Zukunft.

2.3 Die Verwaltungsschale (Asset Administration Shell) als Standard

Während der Digitale Zwilling das übergeordnete, konzeptionelle Leitbild für die umfassende digitale Repräsentation eines Assets darstellt, bedarf es für dessen praktische Umsetzung in der Industrie 4.0 einer konkreten, herstellerübergreifenden Spezifikation. Diese standardisierte Implementierung des Digitalen Zwillings ist die Verwaltungsschale (Salari, 2019). Ihre Entwicklung wird maßgeblich von der Plattform Industrie 4.0 und der **Industrial Digital Twin Association (IDTA)** vorangetrieben, um eine zentrale Herausforderung der Digitalisierung zu lösen: die Interoperabilität. Die **AAS** schafft eine einheitliche digitale Sprache, die es Komponenten, Geräten und Anwendungen ermöglicht, über Unternehmens-, Branchen- und Ländergrenzen hinweg nahtlos zu kommunizieren.

Das Grundprinzip der Verwaltungsschale lässt sich am besten mit der Analogie eines digitalen Karteikartensystems beschreiben. Die **AAS** selbst ist der Kasten mit genormter Größe, der alle Informationen zu einem spezifischen Asset – beispielsweise einer Vakuumpumpe – enthält. Die einzelnen thematisch sortierten Karteikarten in diesem Kasten sind die Teilmodelle (engl. Submodels). So gibt es beispielsweise ein Teilmodell für technische Daten, eines für die Dokumentation und ein weiteres für operative Live-Daten. Diese Struktur wird durch ein übergeordnetes Metamodel formal definiert. Das Metamodel ist der Satz von fundamentalen Gestaltungsregeln, der festlegt, aus welchen Bausteinen eine jede **AAS** bestehen muss, welche Eigenschaften diese Bausteine haben und wie sie zueinander in Beziehung stehen. Die zentralen Bausteine sind dabei die **AAS** selbst, die zugehörigen Teilmodelle und die darin enthaltenen Teilmodellelemente (engl. SubmodelElements), welche die eigentlichen Datenpunkte als Eigenschaften, Operationen oder Dateien mit definierten Datentypen repräsentieren. (*Asset Administration Shell Specification - Part 1: Metamodel*, o. J.))

Ein entscheidender Aspekt für die Skalierbarkeit und Wiederverwendbarkeit ist die Unterscheidung zwischen Typen und Instanzen. Ein Asset kann als Typ (z. B. das generische Pumpenmodell eines Herstellers) oder als Instanz (die spezifische Pumpe mit der Seriennummer XYZ) existieren. Analog dazu werden für Teilmodelle Templates definiert. Ein solches Template agiert wie ein standardisiertes, vorgedrucktes Formular

oder ein Datenbankschema. Es gibt die Struktur, die Bezeichner und die Semantik für ein bestimmtes Teilmodell, wie z.B. ein "Digitales Typenschild", verbindlich vor. Ein Hersteller, der die Daten seiner Pumpe digital bereitstellen möchte, "füllt" dieses Template aus und erzeugt damit eine Teilmodell-Instanz. Dieser Mechanismus stellt sicher, dass alle digitalen Typenschilder, unabhängig vom Hersteller, die gleiche Struktur aufweisen und somit maschinell interoperabel sind. (Bader, 2019)

Die Sicherheit und die Kontrolle über die bereitgestellten Daten sind im industriellen Kontext von höchster Priorität. Die AAS-Spezifikation trägt dem Rechnung, indem sie ein Sicherheitskonzept integriert, das auf **Attribute Based Access Control (ABAC)** basiert. Dieses Modell ermöglicht es, sehr granulare Zugriffsrechte zu definieren und somit präzise zu steuern, welcher Partner welche Informationen innerhalb einer Verwaltungsschale einsehen oder verändern darf. (Bader, 2019)

Der letztendliche Verwendungszweck der Verwaltungsschale ist die Schaffung einer durchgängigen, digitalen Interoperabilität über den gesamten Lebenszyklus eines Assets hinweg – von der Planung und dem Engineering über den Betrieb und die Wartung bis hin zum Recycling. Indem sie eine einheitliche, semantisch reichhaltige und maschinenlesbare Datengrundlage bietet, ermöglicht die AAS den nahtlosen Informationsaustausch zwischen verschiedenen Unternehmen einer Wertschöpfungskette und wird so zum zentralen Wegbereiter für die flexiblen und datengetriebenen Geschäftsmodelle der Industrie 4.0.

2.4 Der Eclipse Dataspace Connector

Während die Verwaltungsschale das *Was* – also die standardisierte Struktur und Semantik – des Digitalen Zwillings definiert, adressieren Datenraumtechnologien das *Wie* des Datenaustauschs. Die fortschreitende Vernetzung im Rahmen von Industrie 4.0 erfordert einen unternehmensübergreifenden Datenaustausch, der jedoch auf einer fundamentalen Prämisse beruhen muss: der Datensouveränität. Jeder Teilnehmer eines Datenökosystems muss zu jeder Zeit die vollständige Kontrolle darüber behalten, wer auf seine Daten zugreift und unter welchen Bedingungen diese genutzt werden dürfen. Die technische Umsetzung solcher souveränen, dezentralen Datenräume wird maßgeblich durch Initiativen wie die **International Data Spaces Association (IDSA)** (International Data Spaces Association, 2024) vorangetrieben.

Der **EDC** ist ein Open-Source-Framework, das als technisches Kernstück zur Realisierung solcher Datenräume dient. (Eclipse Foundation, 2024a) Er implementiert die Protokolle und Mechanismen, die für einen sicheren und souveränen Datenaustausch zwischen Teilnehmern notwendig sind, ohne dass eine zentrale Plattform erforderlich wäre. Die Architektur des **EDC** basiert auf der strikten Trennung von zwei Ebenen: der Kontroll- und der Datenebene (*Control Plane* und *Data Plane*) (Eclipse Foundation, 2024b).

Auf der Kontrollebene finden alle vorbereitenden und administrativen Prozesse statt. Hierzu gehört das Aushandeln von Verträgen. Ein datennutzender Teilnehmer (Konsument) fragt die verfügbaren Datenangebote eines Datenanbieters ab. Jedes Angebot ist mit einer maschinenlesbaren Nutzungsrichtlinie (*Usage Policy*) verknüpft. In einem automatisierten Verhandlungsprozess einigen sich die Konnektoren beider Teilnehmer auf einen digitalen Vertrag, der die genauen Nutzungsbedingungen festlegt. Erst nach erfolgreichem Vertragsabschluss wird die Datenebene für den eigentlichen Transfer aktiviert.

Dieser Mechanismus stellt sicher, dass kein Datentransfer ohne explizite, regelbasierte Zustimmung erfolgt, wodurch die Datensouveränität des Anbieters technisch durchgesetzt wird. Für das in dieser Arbeit untersuchte Szenario ist der **EDC** somit die Schlüsseltechnologie, um den Austausch von Verwaltungsschalen – als Träger der wertvollen Pumpen- und Abatement-Daten – zwischen Halbleiterfabrik und Equipmenthersteller zu ermöglichen. Die **AAS** definiert den Inhalt und die Struktur des auszutauschenden Digitalen Zwillings, während der **EDC** den sicheren und kontrollierten "Transportcontainer" für diesen Austausch bereitstellt und somit die souveräne Zusammenarbeit in einem verteilten industriellen Ökosystem realisiert.

3 Konzeption

3.1 Zielarchitektur

Die Grundlage für den souveränen und standardisierten Austausch von Equipmentdaten bildet eine konzeptionelle Zielarchitektur, welche die informationstechnische Kapselung der physischen Assets und deren kontrollierte Freigabe in einem unternehmensübergreifenden Datenraum beschreibt. Diese Architektur, dargestellt in **Abbildung 1**, gliedert das Gesamtsystem in drei logische Domänen, die den Informationsfluss vom Entstehungsort der Daten bis zu ihrer finalen Verwertung abbilden.

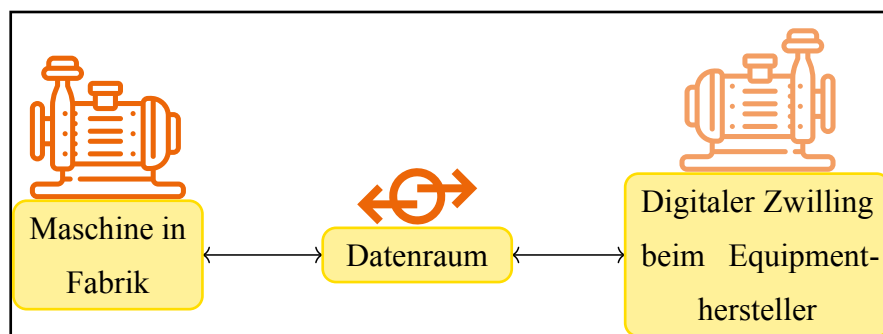


Abbildung 1: Vereinfachte Architektur der Datenübertragung

Der Prozess hat seinen Ursprung in der Domäne des Datenanbieters, in diesem Kontext der Halbleiterfabrik. Hier werden die physikalischen Assets – die Vakuumpumpe und das Abatement-System – durch eine datengenerierende Simulation repräsentiert, welche die prozessrelevanten Druckwerte dynamisch erzeugt. (**Abbildung 2**, „Simulation“)

Diese Livedaten werden in eine servicebasierte Laufzeitumgebung überführt, die für die Verwaltung der Digitalen Zwillinge zuständig ist. (**Abbildung 2**, Nr. 1)

Eine solche Umgebung stellt die Kernfunktionalitäten wie einen Speicher der AAS-Instanzen und ihrer Teilmodelle sowie Services, die das Auffinden dieser Verwaltungsschalen im Netzwerk des Anbieters ermöglicht. (**Abbildung 2**, „Verwaltungsschale“)

Die informationstechnische Brücke zu externen Partnern wird durch die Domäne des Datenraums geschlagen. Für die Realisierung dieses souveränen Datenaustauschs betreiben beide Geschäftspartner eine Instanz eines Konnektors, wie ihn beispielsweise der **EDC** darstellt. (**Abbildung 2**, „Connector Kontroll-/Datenebene“)

Dessen Architektur sieht eine strikte Trennung von Kontroll- und Datenebene vor. Auf der Kontrollebene werden zunächst die Zugriffs- und Nutzungsbedingungen in Form von Richtlinien erstellt. Dies geschieht auf Seiten der Verwaltungsschale. (**Abbildung 2**, Nr. 2)

Nach dem erstellen eines Assets, im anwendungsfall dieser Ausarbeitung eine REST-Endpoint Datenreferenz, am Connector kann der Client über seinen Connector nach einem Angebotskatalog fragen und einen digitalen Vertrag aushandeln. (**Abbildung 2**, Nr. 3)

Erst nach einem erfolgreichen Abschluss dieses Vertrags wird die Datenebene aktiviert, um den eigentlichen Transfer der Verwaltungsschale sicher und gemäß der zuvor vereinbarten Richtlinien durchzuführen. (**Abbildung 2**, Nr. 4 und 5)

In der Domäne des Datennutzers, dem Equipmenthersteller, empfängt schlussendlich eine anwenderspezifische Client-Anwendung (**Abbildung 2**, „Client“) die autorisierten AAS-Daten. Diese Applikation ist für die fachliche Analyse der übermittelten Informationen zuständig, um aus den Druckwerten wertvolle Erkenntnisse für die prädiktive Wartung zu gewinnen und entsprechende Handlungsempfehlungen abzuleiten. Diese architektonische Trennung gewährleistet auf konzeptioneller Ebene, dass die Halbleiterfabrik die volle Souveränität über ihre Daten behält, während der Equipmenthersteller auf standardisierte und autorisierte Weise auf die für ihn relevanten Informationen zugreifen kann.

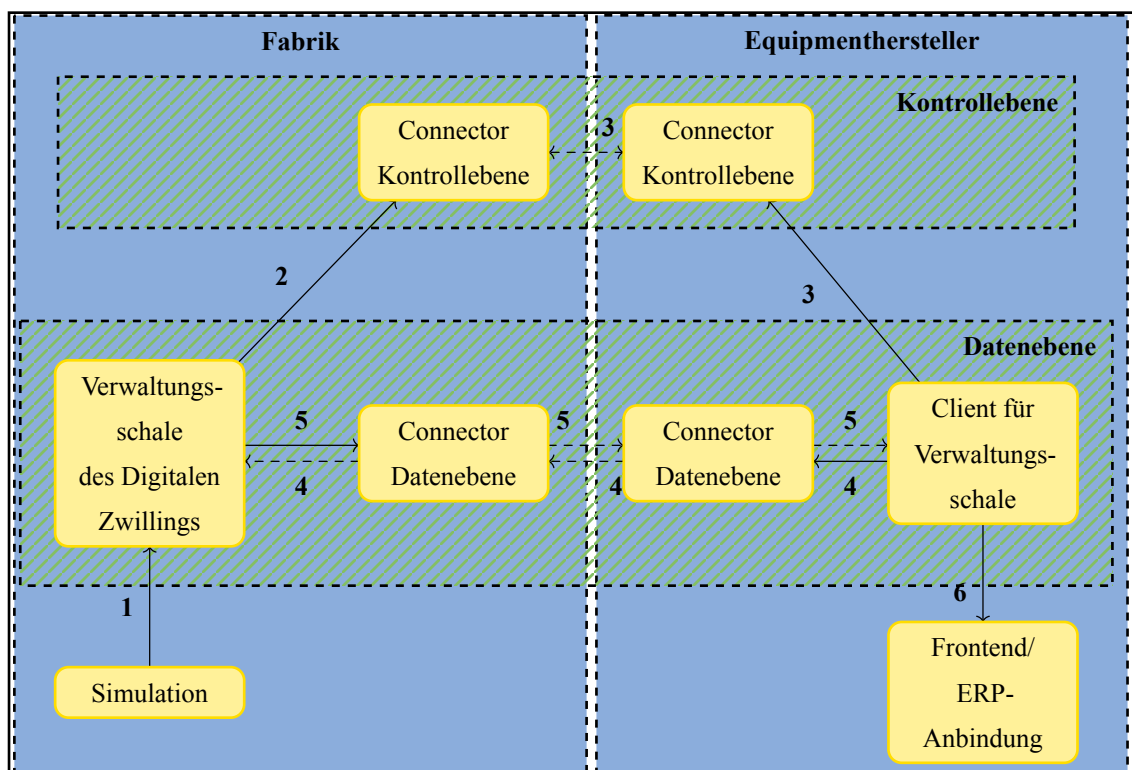


Abbildung 2: Datenfluss über den EDC-Datenraum

3.2 Herleitung des Simulationsansatzes

Die Implementierung einer prädiktiven Wartungsstrategie basiert fundamental auf der Analyse von Daten, die den "Gesundheitszustand" eines Systems widerspiegeln. Im betrachteten Anwendungsfall der Halbleiter-Sub-Fab ist die Effizienz des Abgastransports von der Prozesskammer zum Abatement-System von kritischer Bedeutung. Dieses System (vgl. **Abbildung 3**), bestehend aus Pumpe, Rohrleitung und Abatement-System, bildet eine physikalisch gekoppelte Einheit, deren Zustand maßgeblich durch die herrschenden Druckverhältnisse bestimmt wird. Eine Abweichung der Druckdifferenz zwischen dem Pumpenausgang und dem Abatement-Eingang von einem definierten optimalen Betriebszustand ist ein starker Indikator für eine beginnende Anomalie.

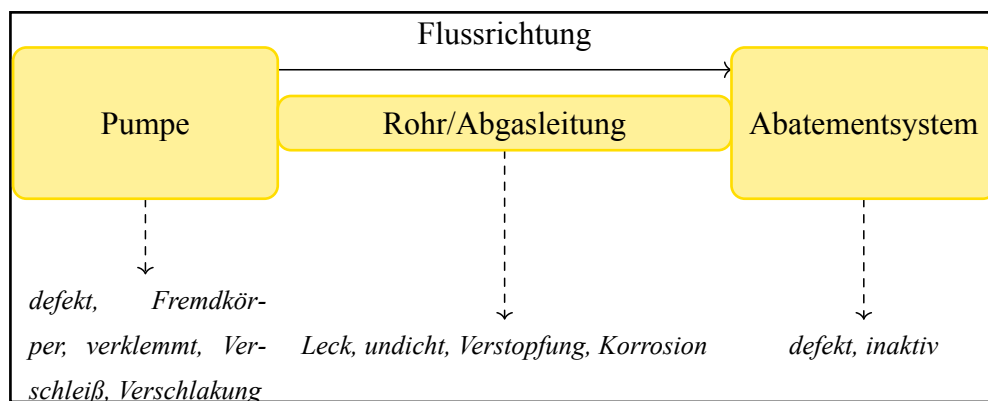


Abbildung 3: Modellierung Pumpen-Abatement-System

Da in der realen Produktionsumgebung keine Livedaten zur Verfügung stehen und das gezielte Herbeiführen von Fehlfunktionen für Testzwecke ausgeschlossen ist, ist die Simulation der physikalischen Prozesse die einzig gangbare Methode, um eine validierte Datengrundlage zu schaffen. Eine solche Simulation ermöglicht es, kontrolliert und reproduzierbar Daten für eine Vielzahl von Betriebs- und Fehlerszenarien zu generieren. Diese Daten sind unerlässlich, um später KI-gestützte Modelle zur Anomalieerkennung zu trainieren und zu validieren. Die zu entwickelnde Simulation muss das dynamische Verhalten des gesamten Abgaspfades realitätsnah abbilden. Kernanforderung ist die Generierung der beiden zentralen Messgrößen, dem Ist-Druck am Ausgang der Pumpe und dem Ist-Druck am Eingang des Abatements.

Basierend auf den physikalischen Gegebenheiten und möglichen Fehlerquellen, die in **Abbildung 3** skizziert sind, lassen sich verschiedene Wartungsszenarien ableiten, die von der Simulation abgebildet werden müssen. Dazu gehören der optimale Normalbetrieb,

eine allmähliche Rohrverschmutzung, eine plötzliche Verstopfung, diverse Formen des Pumpenverschleißes oder -defekts sowie Leckagen im System. Die Korrelationen zwischen den Druckverhältnissen und den daraus resultierenden Fehlerbildern lassen sich systematisch in einer Matrix zusammenfassen, wie sie in **Tabelle 1** dargestellt ist.

p_{ist}^A / p_{ist}^P	über Toleranzwert	innerhalb Toleranzwert	unterhalb Toleranzwert	kein Druck / Negativdruck
über Toleranzwert	Verstopfung	Abatement-Rückstau	Abatement-Rückstau & Pumpenfehler	Pumpenausfall mit Rückstau
innerhalb Toleranzwert	Verschmutzung	Optimaler Betrieb	Leckage / Pumpenfehler	Rohrbruch
unterhalb Toleranzwert	Rohr verstopft	Leckage	Pumpendefekt / Verschleiß	Systemfehler / Inaktivität
kein Druck / Negativdruck	Vollständiger Rohrverschluss	Sensorfehler	Pumpendefekt (Kein Druck)	Pumpe inaktiv / Verklemmt

Tabelle 1: Matrix über Druckverhältnisse und mögliche Szenarien

Aus dieser Systematik werden zentrale wissenschaftliche Hypothesen abgeleitet, welche die Simulationsdaten verifizieren sollen. Es wird postuliert, dass ein schleichender Anstieg der Druckdifferenz zwischen den beiden Messpunkten bei gleichzeitig erhöhtem Druck am Pumpenausgang auf eine progressive Rohrverschmutzung hindeutet. Im Gegensatz dazu wird ein simultaner Druckabfall an beiden Punkten unter den Sollwert als Indikator für einen fortschreitenden Pumpenverschleiß oder einen akuten Defekt angenommen. Eine weitere Hypothese besagt, dass ein signifikanter Druckabfall am Abatement bei gleichzeitig nur leicht verändertem Druck an der Pumpe auf eine Leckage zwischen den beiden Messpunkten hinweist. Die Simulation muss daher so konzipiert sein, dass sie genau diese Muster erzeugen kann. Die generierten Zeitreihendaten für die in **Tabelle 2** aufgeführten Szenarien bilden die Grundlage, um Algorithmen zu entwickeln,

Use-Case	Druck Pumpe [mBar]	Druck Abatement [mBar]	Pumpe Toleranz	Abatement Toleranz
Verstopfung	130	130	über	über
Abatement-Rückstau	110	130	innerhalb	über
Abatement-Rückstau & Pumpenfehler	70	130	unter	über
Pumpenausfall mit Rückstau	0	130	kein	über
Verschmutzung	130	110	über	innerhalb
Optimaler Betrieb	100	100	innerhalb	innerhalb
Leckage / Pumpenfehler	70	100	unter	innerhalb
Rohrbruch	0	100	kein	innerhalb
Rohr verstopft	130	70	über	unter
Leckage	110	70	innerhalb	unter
Pumpendefekt / Verschleiß	70	70	unter	unter
Systemfehler / Inaktivität	0	70	kein	unter
Vollständiger Rohrverschluss	130	0	über	kein
Sensorfehler	100	0	innerhalb	kein
Pumpendefekt (Kein Druck)	70	0	unter	kein
Pumpe inaktiv / Verklemmt	0	0	kein	kein

Tabelle 2: Use-Cases für die Simulation

die eine Klassifizierung dieser unterschiedlichen Fehlerbilder ermöglichen und somit eine gezielte Wartungsempfehlung ableiten können.

3.3 Auswahl des Implementierungs-Frameworks: Eclipse BaSyx

Um die Konzepte der Verwaltungsschale praktisch umzusetzen, bedarf es einer Middleware, die eine standardkonforme Realisierung der Digitalen Zwillinge sowie der notwendigen Infrastrukturdienste ermöglicht. Im Rahmen dieser Arbeit fiel die Wahl auf **Eclipse BaSyx**, da es sich hierbei um eine der ersten und zugleich die offizielle Open-Source-Referenzimplementierung für die Verwaltungsschale handelt, die unter dem Dach der Eclipse Foundation entwickelt wird. (Eclipse Foundation, 2024c)

Das Projekt wird maßgeblich vom Fraunhofer-Institut für Experimentelles Software Engineering (IESE) koordiniert und von einem breiten Netzwerk aus Industrie- und Forschungspartnern unterstützt und genutzt. (Eclipse Foundation, 2024d)

Die Entscheidung für dieses Framework gründet sich auf mehreren zentralen Vorteilen. Als Referenzimplementierung stellt es eine hohe Konformität mit den aktuellen Spezi-

fikationen der IDTA sicher. Die Bereitstellung als Open-Source-Software unter einer industriefreundlichen Lizenz senkt die Einstiegshürden erheblich und erlaubt auch eine Nutzung im kommerziellen Kontext. (Eclipse Foundation, 2017)

Des Weiteren bietet BaSyx **Software development kits (SDKs)** für gängige Programmiersprachen wie Java, C# und Python, was eine hohe Flexibilität bei der Anwendungsentwicklung gewährleistet. (Eclipse Foundation, 2024e)

Eclipse BaSyx stellt eine modulare Middleware zur Verfügung, die alle notwendigen Komponenten für den Aufbau einer **AAS**-basierten Infrastruktur als einzelne, containerisierte Microservices bereitstellt. Diese Dienste können kombiniert werden, um eine vollständige Laufzeitumgebung für Digitale Zwillinge zu schaffen. (Eclipse Foundation, 2023)

Der **AAS Server**, oft auch als **Repository** bezeichnet, ist die Kernkomponente für das Hosting der eigentlichen **AAS**- und Teilmodell-Instanzen. Er stellt eine standardisierte API (z.B. HTTP/REST) bereit, über die auf die Inhalte des Digitalen Zwillings zugegriffen, diese erstellt, gelesen, aktualisiert oder gelöscht werden können.

Die **AAS Registry** fungiert als zentrales Verzeichnis oder "Telefonbuch" für die Verwaltungsschalen. Jede **AAS**-Instanz im System wird hier mit ihrem eindeutigen Identifikator und dem Netzwerk-Endpunkt ihres **AAS** Servers registriert. Anwendungen können die Registry abfragen, um dynamisch herauszufinden, unter welcher Adresse eine bestimmte Verwaltungsschale erreichbar ist.

Analog zur **AAS** Registry dient die **Submodel Registry** der Registrierung und dem Auffinden von einzelnen Teilmodellen. Dies unterstützt eine dezentrale Architektur, in der Teilmodelle von unterschiedlichen Servern bereitgestellt und dynamisch zu einer Gesamt-**AAS** aggregiert werden können.

Aufbauend auf den Registries ermöglicht der **Discovery**-Service eine semantische Suche über den Bestand der Verwaltungsschalen. Anstatt eine **AAS** über ihre eindeutige ID zu suchen, können Anwendungen hier beispielsweise alle Assets abfragen, die über ein bestimmtes Teilmodell (z.B. "Technische Daten") oder eine spezifische Eigenschaft verfügen.

Die **AAS Environment**-Komponente fasst schließlich die Gesamtheit aller **AAS**-bezogenen Elemente – also die **AAS** selbst, ihre Teilmodelle und die zugehörigen semantischen Definitionen (Concept Descriptions) – zusammen und stellt sie über eine einzige Schnitt-

stelle als kohärente Einheit bereit, was insbesondere den Austausch von vollständigen Digitalen Zwillingen vereinfacht.

Obwohl alternative Open-Source-Implementierungen existieren, die sich teils durch eine einfachere Inbetriebnahme auszeichnen, wurde Eclipse BaSyx aufgrund seiner umfassenden Modularität und der vollständigen Abdeckung des AAS-Standards als überlegen für die in dieser Arbeit geforderte Realisierung bewertet. Die modulare Architektur von BaSyx erlaubt es, die einzelnen Infrastrukturkomponenten gezielt aufzusetzen und deren Zusammenspiel im Kontext des Eclipse Dataspace Connectors detailliert zu untersuchen, was eine zentrale Anforderung dieser wissenschaftlichen Arbeit ist.

3.4 Entwurf des AAS-Modells

Die informationstechnische Abbildung des Sub-Fab-Systems erfordert ein durchdachtes und strukturiertes Modell der beteiligten Assets, das deren physische und logische Beziehungen widerspiegelt. Der hier verfolgte Entwurfsansatz basiert auf einer hierarchischen Komposition von Verwaltungsschalen, die eine **Parent-Child**-Beziehung etabliert. Dieser Aufbau ermöglicht es, das Gesamtsystem als eine logische Einheit zu betrachten und gleichzeitig die einzelnen Komponenten als eigenständige Digitale Zwillinge zu verwalten.

Das Wurzelement der Modellierung bildet eine übergeordnete Verwaltungsschale, beispielhaft als „SemiconductorX“ bezeichnet. Diese AAS fungiert als logische Klammer (Parent) für das gesamte zu betrachtende Abgassystem. Unterhalb dieser Hülle sind die Verwaltungsschalen für die eigentlichen physischen Komponenten als untergeordnete Elemente (Children) angesiedelt: die „**IndustrialExhaustPump**“ und das „**Abatement-System**“. Jede dieser beiden Verwaltungsschalen stellt einen eigenständigen Digitalen Zwilling des jeweiligen Assets dar und wird durch eine Reihe von **Teilmodellen** beschrieben, welche die verschiedenen Aspekte des Assets kapseln.

Ein zentrales Ziel dieses Entwurfs ist die Gewährleistung von Interoperabilität und Standardkonformität. Aus diesem Grund wird, wo immer möglich, auf die Verwendung von standardisierten **Teilmodell-Templates** zurückgegriffen. (Admin Shell IO, 2024) Templates sind vordefinierte, standardisierte Schemata für Teilmodelle, die deren Struktur, Elemente und semantische Bedeutung festlegen und somit eine herstellerübergreifende, einheitliche Beschreibung von Aspekten ermöglichen. Für die grundlegende Identifikation wird in allen drei Verwaltungsschalen das etablierte Template „**Digital Nameplate**“

verwendet, um wesentliche Hersteller- und Typinformationen bereitzustellen. Zur Beschreibung der technischen Schnittstellen und Kommunikationsprotokolle wird für die Pumpe und das Abatement-System das Teilmodell „**TechnicalData**“ eingesetzt, welches auf dem standardisierten Template „**Asset Interfaces Description**“ basiert. Dies stellt sicher, dass die Art und Weise, wie auf die Daten der Assets zugegriffen werden kann, einheitlich und maschinenlesbar spezifiziert ist.

Den Kern des anwendungsspezifischen Datenmodells bildet jedoch das eigens für diese Arbeit konzipierte Teilmodell „**OperationalData**“. Da für die Erfassung von Live-Betriebsdaten kein universelles Standard-Template existiert, das die spezifischen Anforderungen dieses Anwendungsfalls abdeckt, wird hier ein domänenspezifisches Modell entworfen. Dieses Teilmodell enthält die für die prädiktive Wartungsanalyse entscheidenden Eigenschaften. Für die „IndustrialExhaustPump“ ist dies die Eigenschaft „**ActualPressure**“, welche den Ist-Druck an ihrem Ausgang repräsentiert. Korrespondierend dazu enthält das „OperationalData“-Teilmodell des „AbatementSystem“ ebenfalls eine Eigenschaft namens „**ActualPressure**“, die den Druck an seinem Eingang abbildet. Die semantische Verknüpfung und der Abgleich genau dieser beiden Werte bilden die datentechnische Grundlage für die spätere Analyse von Rohrverschmutzungen und Pumpendefekten.

Die Struktur der Verwaltungsschale ist in **Anhang A** beschrieben.

3.5 Konzeption des souveränen Datenraums

Um die informationstechnische Lücke zwischen dem Fabrikbetreiber (Datenanbieter) und dem Equipmenthersteller (Datennutzer) zu schließen, bedarf es einer Architektur, die den Austausch der generierten Equipmentdaten sicher und richtlinienbasiert ermöglicht. Der Kern dieses Konzepts ist die Etablierung eines dezentralen Datenraums unter Einsatz des **EDC**.

Anders als bei herkömmlichen, tunnelbasierten Netzwerklösungen (wie etwa einem VPN), bei denen Systeme auf Netzwerkebene durchgereicht werden, fungiert der **EDC** als souveräner Daten-Proxy. Das bedeutet, dass die anbietende Domäne ihre Verwaltungsschalen nicht direkt exponiert. Stattdessen wird die Kommunikation in eine Kontrollebene und eine Datenebene aufgetrennt.

Konzeptionell sieht der lokale Aufbau zwei voneinander isolierte Konnektor-Instanzen vor: einen Provider-Konnektor auf Seiten der Halbleiterfabrik und einen Consumer-

Konnektor auf Seiten des Equipmentherstellers. Der Datenaustauschprozess ist streng vertragsbasiert und durchläuft mehrere konzeptionelle Phasen:

Katalogbereitstellung und -abfrage: Der Provider-Konnektor aggregiert die verfügbaren Teilmodelle der AAS (spezifisch das `OperationalData`-Modell mit den Druckwerten) zu einem Datenkatalog. Jedes Datenangebot ist zwingend an eine maschinenlesbare Nutzungsrichtlinie geknüpft.

Vertragsaushandlung: Der Consumer-Konnektor fragt diesen Katalog ab und initiiert eine automatisierte Aushandlung. Stimmen die Anforderungen des Nutzers mit den Richtlinien des Anbieters überein, wird ein bindendes Abkommen generiert.

Autorisierter Transfer: Erst nach erfolgreichem Vertragsabschluss tauschen die Kontrolllebenen kryptografische Token aus. Die Datenebene des Datennutzers erhält daraufhin eine gesicherte Endpunkt-Referenz, über die der tatsächliche **Hypertext Transfer Protocol (HTTP)**-Datenstrom der AAS-Druckwerte abgerufen werden kann.

Durch dieses Proxy-Pattern wird sichergestellt, dass die Datenhoheit jederzeit beim Fabrikbetreiber verbleibt, während der Equipmenthersteller einen genormten, sicheren Zugriffspunkt für seine Analysedienste erhält.

3.6 Konzeption der Client-Anwendung und Visualisierung

Die konzeptionelle Kette des souveränen Datenaustauschs endet in der Domäne des Datennutzers. Um die bereitgestellten Informationen der Verwaltungsschale domänenspezifisch verwerten zu können, wird eine dedizierte Client-Anwendung als informationstechnischer Endpunkt entworfen. Der Architekturansatz dieses Clients ist modular und evolutionär aufgebaut: Zur initialen Validierung der reinen Kommunikations- und Ausleselogik ist das System so konzipiert, dass es zunächst direkt mit der AAS-Laufzeitumgebung interagieren kann. In der finalen Ausbaustufe der Architektur wird diese direkte Verbindung durch den **EDC** als obligatorischen Proxy substituiert, um den souveränen Datenaustausch zu erzwingen.

Der konzeptionelle Workflow des Clients im vollständig integrierten Datenraum gliedert sich in vier logische Phasen: Vertragsaushandlung, Datenabruf, Auswertung und Persistierung.

Zunächst interagiert der Client als aktiver Konsument mit dem **EDC**. Er fragt den bereitgestellten Angebotskatalog des Datenanbieters ab und initiiert die Aushandlung der

Nutzungsbedingungen. Nach der erfolgreichen Akzeptanz des digitalen Vertrages erhält der Client einen temporären Zugangsschlüssel, der ihn für den nachfolgenden Datentransfer autorisiert.

Mit diesem Schlüssel erfolgt der eigentliche Datenabruf. Die konzeptionelle Herausforderung dieses Schrittes liegt in der korrekten informationstechnischen Adressierung: Um die dynamischen Druckwerte punktgenau abzufragen, muss der Client die exakten, eindeutigen Identifikatoren (Submodel-IDs) der Ziel-Assets – der Vakuumpumpe und des Abatement-Systems – kennen. Nur mit diesen Referenzen kann die Anfragelogik die relevanten Parameter aus der Hierarchie der Verwaltungsschale zielgerichtet extrahieren. Nach dem erfolgreichen Abruf der Livedaten findet die analytische Auswertung direkt im Client statt. Anstelle eines ressourcenintensiven Modells des maschinellen Lernens sieht das Konzept eine deterministische, schwellwertbasierte Analysekomponente vor. Basierend auf der zuvor definierten Simulationsmatrix (vgl. **Tabelle 1**) vergleicht die Logik die abgerufenen Druckdifferenzen kontinuierlich mit definierten Toleranzbändern. Überschreitet beispielsweise der Druck an der Pumpe einen Schwellwert bei gleichzeitigem Druckabfall am Abatement, klassifiziert das System dieses Ereignis automatisch als Anomalie, etwa als Rohrverstopfung.

Um diese abgeleiteten Zustandsbewertungen sowie die rohen Druckverläufe für das Wartungspersonal nutzbar zu machen, schließt das Konzept mit der methodischen Trennung von Datenverarbeitung und Präsentation ab. Der Client persistiert die abgerufenen und ausgewerteten Datenpunkte fortlaufend in einem strukturierten, generischen Speicherformat. Ein eigenständiges Visualisierungs-Frontend greift anschließend rein lesend auf diesen Speicher zu und transformiert die abstrakten Zeitreihen in anschauliche Graphen. Diese strikte Entkopplung gewährleistet ein performantes Condition Monitoring: Das Frontend fungiert lediglich als visuelles Dashboard, während die gesamte komplexe Geschäftslogik, die Vertragsaushandlung und die Auswertung sicher im Client gekapselt bleiben.

Darüber hinaus ist die Architektur des Clients gezielt darauf ausgelegt, eine nahtlose Integration in die bestehende Unternehmens-IT des Equipmentherstellers, insbesondere in übergeordnete **Enterprise Resource Planning (ERP)**-Systeme, zu ermöglichen. Der konzeptionelle Mehrwert dieses Ansatzes liegt in der Überwindung proprietärer Insellösungen: Anstatt für jeden Fabrikbetreiber individuelle und wartungsintensive Schnittstellen pflegen zu müssen, fungiert der Client als standardisierter Adapter. Er übersetzt die

normierten **AAS**-Daten aus dem souveränen Datenraum in das interne Datenformat des Herstellers.

3.7 Architektonische Erweiterbarkeit für KI-gestützte Analyseverfahren

Diese umfassen insbesondere die Anbindung externer Machine-Learning-Dienste an den etablierten Datenraum, um die avisierte KI-gestützte Analyse auf Basis der geschaffenen Infrastruktur vollständig zu realisieren. Ein zentrales Motiv bei der Konzeption dieser Datenpipeline ist die Schaffung einer zukunftssicheren und skalierbaren Infrastruktur, die perspektivisch auch hochkomplexe Auswertungsmethoden unterstützt. Die in der ursprünglichen Zielsetzung avisierte KI zur Mustererkennung von Fehlerfällen wird in diesem Architekturkonzept bewusst als entkoppelte, nachgelagerte Instanz betrachtet. Der Fokus liegt auf der Etablierung einer robusten, standardisierten Datenbasis, die als technologischer Wegbereiter für derartige fortgeschrittene Analyseverfahren fungiert.

Durch die konsequente Nutzung der **AAS** als normiertes semantisches Informationsmodell und des **EDC** als souveränen Transportweg entsteht eine hochgradig generische Schnittstellenarchitektur. Diese strikte Standardisierung bedingt, dass eine zukünftige KI-Anwendung nicht als proprietäres, monolithisches Modul tief in den lokalen Client integriert werden muss. Vielmehr kann ein solches System – sei es ein lokaler Machine-Learning-Dienst oder eine cloudbasierte Analyseplattform – als eigenständiger, weiterer Konsument innerhalb des Datenraums agieren. Eine solche Anwendung ist in der Lage, die historischen und echtzeitnahen Druckwerte über etablierte Web-Standards (wie etwa **Representational State Transfer (REST)**-Schnittstellen) verlustfrei und vertraglich abgesichert abzurufen.

Für den informationstechnischen Nachweis der Pipeline im Rahmen dieser Ausarbeitung ist die im Client implementierte deterministische Schwellwertanalyse vollkommen ausreichend, um die grundsätzliche informationstechnische Machbarkeit der prädiktiven Wartung zu belegen. Externe KI-Dienste können durch die generische Natur des **EDC** jederzeit nahtlos an den souveränen Datenraum angebunden werden, ohne dass tiefgreifende strukturelle Anpassungen an der bestehenden Infrastruktur der Halbleiterfabrik oder des primären Clients vorgenommen werden müssen.

4 Implementierung

4.1 Aufbau der AAS-Laufzeitumgebung

Die serverseitige Grundlage für die Bereitstellung der Digitalen Zwillinge wurde unter Verwendung der offiziellen **“Off-the-Shelf“-Komponenten** von Eclipse BaSyx realisiert. (Eclipse Foundation, 2023) Dieser Ansatz nutzt von den Entwicklern bereitgestellte, vorgefertigte Docker-Images, was den Aufbau einer standardkonformen und reproduzierbaren Systemumgebung erheblich vereinfacht. Für die in dieser Arbeit benötigte Kernfunktionalität wurden dabei insbesondere die Images für den **AAS-Server (Repository)**, die **AAS Registry** sowie den übergeordneten **AAS Environment-Service** verwendet.

Die Orchestrierung und Konfiguration dieser containerisierten Dienste erfolgte mittels einer **“docker-compose“-Datei**. Als Ausgangspunkt dienten hierfür die von der BaSyx-Community bereitgestellten Beispielkonfigurationen (Eclipse Foundation, 2024f), welche für die spezifischen Anforderungen dieses Projekts adaptiert wurden. Diese Anpassungen umfassten unter anderem die Netzwerkkonfiguration, um eine reibungslose Kommunikation zwischen den einzelnen Diensten sicherzustellen. Die resultierende, für diese Arbeit verwendete **“docker-compose“-Konfiguration** ist im **Anhang B** detailliert dokumentiert.

Nach dem erfolgreichen Start der Laufzeitumgebung erfolgt die gesamte Interaktion mit den Verwaltungsschalen über eine standardisierte HTTP/REST-Schnittstelle, die von den BaSyx-Diensten bereitgestellt wird. Die explorative Analyse der API ergab, dass mittels GET-Anfragen sowohl das Auffinden von Verwaltungsschalen über die Registry als auch das gezielte Auslesen von AAS, Teilmodellen und einzelnen Eigenschaften möglich ist. Für die dynamische Aktualisierung der Digitalen Zwillinge, wie sie später durch den Simulator erfolgt, kommen **“PUT“-Anfragen** zum Einsatz, mit denen die Werte einzelner Eigenschaften (Properties) überschrieben werden können. Eine wesentliche technische Erkenntnis bei der Arbeit mit der API war, dass Identifikatoren für den Transfer Base64-kodiert sein müssen, um eine URI-konforme Übertragung zu gewährleisten. Besonders entwicklerfreundlich ist, dass der Environment-Service eine interaktive Weboberfläche bereitstellt, auf der die gesamte API-Definition live abgefragt und getestet werden kann.

4.2 Modellierung und Erstellung der Verwaltungsschalen

Auf Basis der in **Kapitel 3.4** konzipierten hierarchischen Struktur (vgl. **Anhang Anhang A**) wurden die konkreten Verwaltungsschalen für das Abgassystem erstellt. Ein

zentrales Kriterium bei der Modellierung war die maximale Wiederverwendung von standardisierten Bausteinen, um die Interoperabilität zu gewährleisten. Daher wurden für Aspekte, die einer branchenweiten Standardisierung unterliegen, offizielle Teilmodell-Templates aus dem öffentlichen Git-Repository der **IDTA** verwendet (Admin Shell IO, 2024).

Für die grundlegende Identifikation aller Assets kam das Template **“Digital Nameplate”** zum Einsatz. Dieses weit verbreitete Teilmodell dient der Bereitstellung von Kerninformationen, die typischerweise auf einem physischen Typenschild zu finden sind, wie beispielsweise Herstellername (`ManufacturerName`), Produktbezeichnung (`ManufacturerProductDesignation`) und Seriennummer (`SerialNumber`). Durch die Nutzung dieses standardisierten Templates wird sichergestellt, dass jede Komponente im System ihre Stammdaten in einer einheitlichen, maschinenlesbaren Form präsentiert.

Da für die Erfassung der dynamischen Betriebsdaten kein universelles Standard-Template existiert, das die spezifischen Anforderungen dieses Anwendungsfalls abdeckt, wurde das anwendungsspezifische Teilmodell **“OperationalData”** eigens für diese Arbeit konzipiert und erstellt. Dieses Teilmodell ist der zentrale Baustein für die spätere Analyse und enthält die für die prädiktive Wartung entscheidenden Eigenschaften (`Properties`). In der Verwaltungsschale der Pumpe (`IndustrialExhaustPump`) und des Abatement-Systems (`AbatementSystem`) wurde jeweils eine Eigenschaft namens **“Actual Pressure”** implementiert. Diese repräsentiert den momentanen Ist-Druck am jeweiligen Messpunkt und dient als direkter Eingang für die Simulationsdaten, ohne dabei einen historischen Verlauf zu speichern.

Die Erstellung der finalen `.aasx`-Paketdateien, welche die konfigurierten Verwaltungsschalen mit ihren standardisierten und anwendungsspezifischen Teilmodellen enthalten, erfolgte mit dem **AASX Package Explorer** (Eclipse Foundation, 2024g). Dieses Werkzeug ermöglicht die grafische Komposition der verschiedenen Teilmodelle, die Konfiguration der spezifischen Eigenschaftswerte (z.B. das Eintragen der Seriennummern) und die Bündelung aller zugehörigen Informationen in einer einzigen, austauschbaren Datei. Ein solches AASX-Paket repräsentiert den vollständigen, in sich geschlossenen Digitalen Zwilling eines Assets und bildet die Datengrundlage für den späteren Transfer durch den Datenraum. (vgl. **Abbildung 4**)

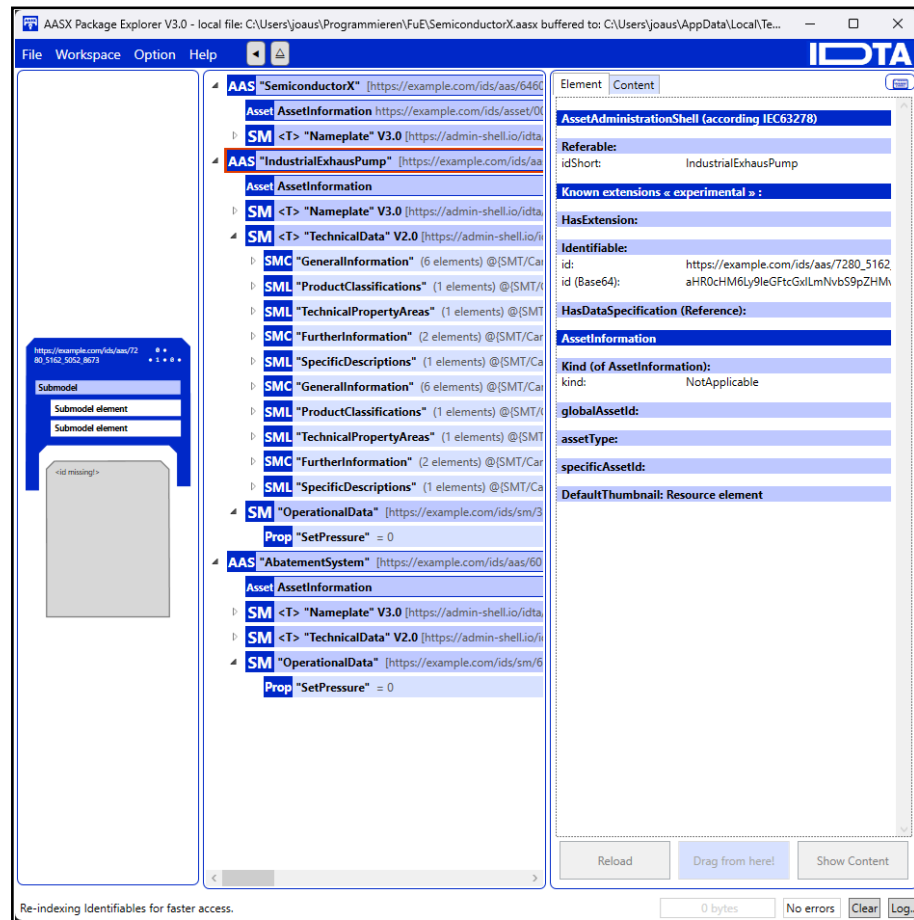


Abbildung 4: Bearbeiten der Verwaltungsschale mit dem AASX-Packageexplorer

4.3 Realisierung und Anbindung des Drucksimulators

Die Generierung der für die Analyse notwendigen Livedaten erfolgt durch einen eigens für diese Arbeit entwickelten, prozessbasierten Simulator. Als technologische Basis wurde die Programmiersprache **Python** in Kombination mit der Bibliothek **SimPy** gewählt. Python bietet durch sein reichhaltiges Ökosystem an Bibliotheken eine hervorragende Grundlage für wissenschaftliche und datenintensive Anwendungen, während SimPy als prozessbasiertes Framework für diskrete Ereignissimulationen besonders geeignet ist, um das zeitliche Verhalten von Systemen, wie den schleichenden Anstieg einer Rohrverschmutzung oder plötzliche Defekte, realitätsnah abzubilden. Der Simulator ist architektonisch in zwei parallel laufende Prozesse aufgeteilt, die mittels Multithreading realisiert wurden: ein Prozess ist für die GUI und die Nutzerinteraktion zuständig, der andere für die kontinuierliche Generierung der Druckwerte und die Kommunikation mit der AAS-Infrastruktur.

Die Kernfunktionalität des Simulators ist die Erzeugung der Druckwerte für die Pumpe und das Abatement-System. Um abrupte Sprünge zu vermeiden und ein realistisches, träges Systemverhalten nachzubilden, werden die Ist-Druckwerte iterativ an die Soll-Werte angenähert. Dies simuliert den langsamen Druckauf- oder -abbau in einem physischen System. Über die grafische Benutzeroberfläche kann der Anwender gezielt zwischen den zuvor in der Szenarien-Matrix definierten Wartungsfällen (vgl. **Tabelle 1**) wählen oder manuell spezifische Drucksollwerte eingeben, um individuelle Tests durchzuführen. So können gezielt Daten für alle relevanten Zustände – vom Optimalbetrieb bis zum vollständigen Systemausfall – erzeugt werden.

Die Anbindung des Simulators an die in **Kapitel 4.1** aufgebaute AAS-Laufzeitumgebung erfolgt über deren REST-API. In einem kontinuierlichen Prozess übermittelt der Simulator die aktuell generierten Ist-Druckwerte mittels einer HTTP-‘PATCH’-Anfrage (eine spezifischere Form der ‘POST’-Anfrage zur Aktualisierung eines Teils einer Ressource) an das entsprechende ‘Property’-Element im ‘OperationalData’-Teilmodell der jeweiligen Verwaltungsschale. Dieser Ansatz wurde bewusst gewählt, da er das typische Verhalten von IoT-fähigen Sensoren oder Mikrocontrollern nachbildet, die ihre Messwerte ebenfalls über etablierte Netzwerkprotokolle an übergeordnete Systeme senden. Der Simulator agiert somit als valider und realitätsnaher Stellvertreter für die physischen Assets in der Zielarchitektur.

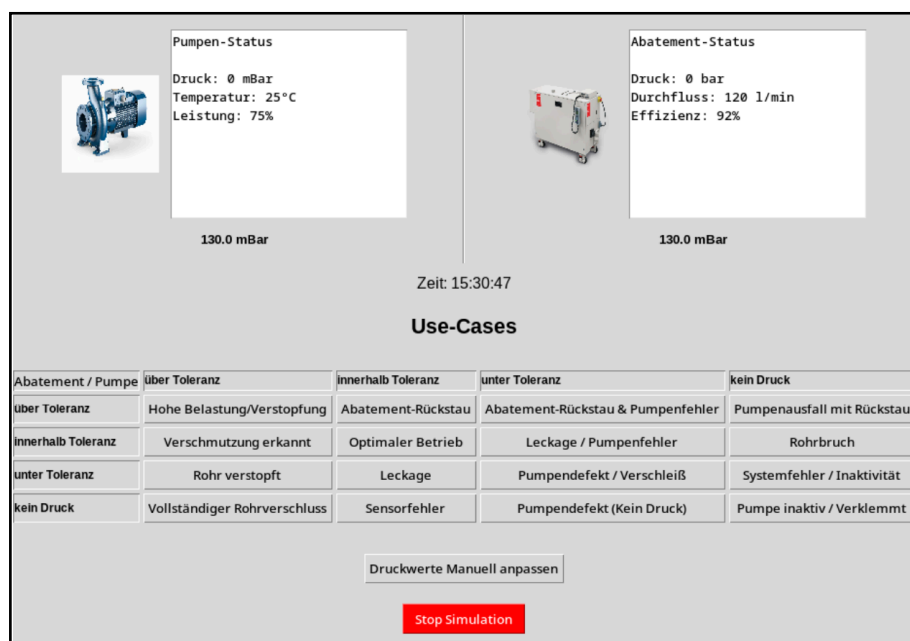


Abbildung 5: Python Drucksimulator GUI

Die **Abbildung 5** zeigt die GUI des Python-Drucksimulators. Über Knopfdruck lassen sich die verschiedenen Use-Cases wie in **Tabelle 2** beschrieben bedienen.

4.4 Bereitstellung der lokalen EDC-Infrastruktur

Die praktische Realisierung des souveränen Datenraums erfolgte durch die lokale Instanziierung des **EDC**. Um eine realitätsnahe Trennung der Domänen abzubilden, wurden zwei separate Konnektor-Instanzen – ein Provider-Konnektor und ein Consumer-Konnektor – auf Basis der offiziellen **EDC**-Samples kompiliert und bereitgestellt. (Eclipse Foundation, 2026) Die spezifische Konfiguration der jeweiligen Laufzeitumgebungen, wie etwa die Definition der Netzwerk-Ports und der anzubindenden Backend-Dienste, wurde über dedizierte Konfigurationsdateien (`provider-config.properties` und `consumer-config.properties`) realisiert. Zur Gewährleistung einer reproduzierbaren und plattformunabhängigen Ausführung wurde die gesamte Infrastruktur mittels Docker-Compose orchestriert.

Die Validierung der Kontrollebene (Control Plane) und der fehlerfreien Konnektivität zwischen den beiden Domänen wurde iterativ mittels der **Application Programming Interface (API)**-Entwicklungsplattform Postman durchgeführt. Durch eine eigens erstellte Postman-Collection konnten die essenziellen **HTTP**-Aufrufe zur Erstellung von Datenangeboten auf Provider-Seite sowie die Abfrage des Katalogs auf Consumer-Seite manuell ausgelöst und verifiziert werden. Dieser semi-automatisierte Testansatz stellte sicher, dass die kryptografische und logische Infrastruktur des Datenraums funktionsfähig ist, bevor die Client-Applikation angebunden wurde.

Die Postman-Collection und die EDC-configurations Dateien sowie das Docker-Compose script sind auf dem Gitlab der HTW-Dresden zu finden unter `Wintersemester_2025-26`. (Rudolph, 2026)

4.5 Die Client-Applikation

Den informationstechnischen Abschluss der Datenpipeline bildet eine objektorientierte Konsolenanwendung. Diese wurde in Java (Version 17) unter Verwendung des Build-Management-Tools Maven (`de.htw_dresden.jrudolph:aas_consumer_client`) realisiert. Die interne Softwarearchitektur ist in die logischen Pakete `configuration`, `processing` und `util` gegliedert und folgt strikt etablierten Entwurfsmustern, um eine hohe Wartbarkeit und Skalierbarkeit zu gewährleisten.

Die zentrale Steuerung und Konfiguration der Anwendung erfolgt dynamisch über Umgebungsvariablen. Diese Parameter – wie beispielsweise Verbindungs-URLs und das gewählte Abrufverfahren – werden zur Laufzeit in einem unveränderlichen Datenobjekt, dem `EndpointRecord`, gekapselt. Der Einsatz von Java Records garantiert hierbei eine threadsichere und speichereffiziente Haltung der Konfigurationsdaten.

Abhängig von der übergebenen Umgebungsvariable instanziiert das System über das Factory/Builder-Pattern unterschiedliche Strategien zur Datenbeschaffung. Für Entwicklungs- und Validierungszwecke kann die Applikation eine direkte Anbindung an die BaSyx-Laufzeitumgebung (`BaSyxPressureProvider`) nutzen. Im produktiven Zielszenario wird jedoch dynamisch auf den `EDCPressureProvider` umgeschaltet, welcher native **HTTP**-Requests an den souveränen Datenraum richtet und den vertraglich ausgehandelten Autorisierungsschlüssel für den Datenabruf injizieren kann.

Der kontinuierliche Abruf der Sensordaten wird im Hintergrund durch die Klasse `PressureWatcher` orchestriert, welche als eigenständiger Thread ausgeführt wird. Um die Datenbeschaffung von der anschließenden Verwertung informationstechnisch zu entkoppeln, wurde ein ereignisgesteuerter Ansatz gewählt. Sobald der `PressureWatcher` einen neuen validen Datensatz (`PressureRecord`) empfängt, benachrichtigt er alle registrierten Abonnenten, die das `PressureListener`-Interface implementieren.

In dieser konkreten Implementierung sind zwei primäre Listener registriert:

Der `PressureAnalyser`: Diese Komponente fungiert als Aufpasser und logische Schnittstelle für die Auswertung der Druckdifferenzen. Durch die generische Architektur dient diese Klasse als informationstechnischer Ankerpunkt, an dem in zukünftigen Ausbaustufen komplexe Modelle zur Mustererkennung oder Algorithmen der künstlichen Intelligenz nahtlos und ohne tiefgreifende Refactorings des Clients integriert werden können.

Der `CsvWriter`: Diese Komponente übernimmt die Persistierung der extrahierten Daten. Bei jedem Eintreffen neuer Werte schreibt der Writer die Druckdaten der Pumpe und des Abatement-Systems, angereichert mit einem exakten Zeitstempel, in den lokalen Speicher. Die Entscheidung für das **Comma Separated Value (CSV)**-Format fiel hierbei bewusst: Es bietet eine hohe maschinelle Kompatibilität für nachgelagerte Dashboards, ist ressourcenschonend und bleibt gleichzeitig direkt durch den Menschen (bspw. das Wartungspersonal) lesbar und interpretierbar.

Der Einstiegspunkt der Applikation (`Application.java`) demonstriert diese saubere Trennung der Zuständigkeiten, indem er die Komponenten über ihre jeweiligen Builder instanziiert, die Listener registriert und den asynchronen Überwachungsprozess startet.

Der Quellcode ist wiederzufinden im GitLab der HTW Dresden unter „WinterSemester_25-26/aas_client/aas_consumer_client“. (Rudolph, 2026)

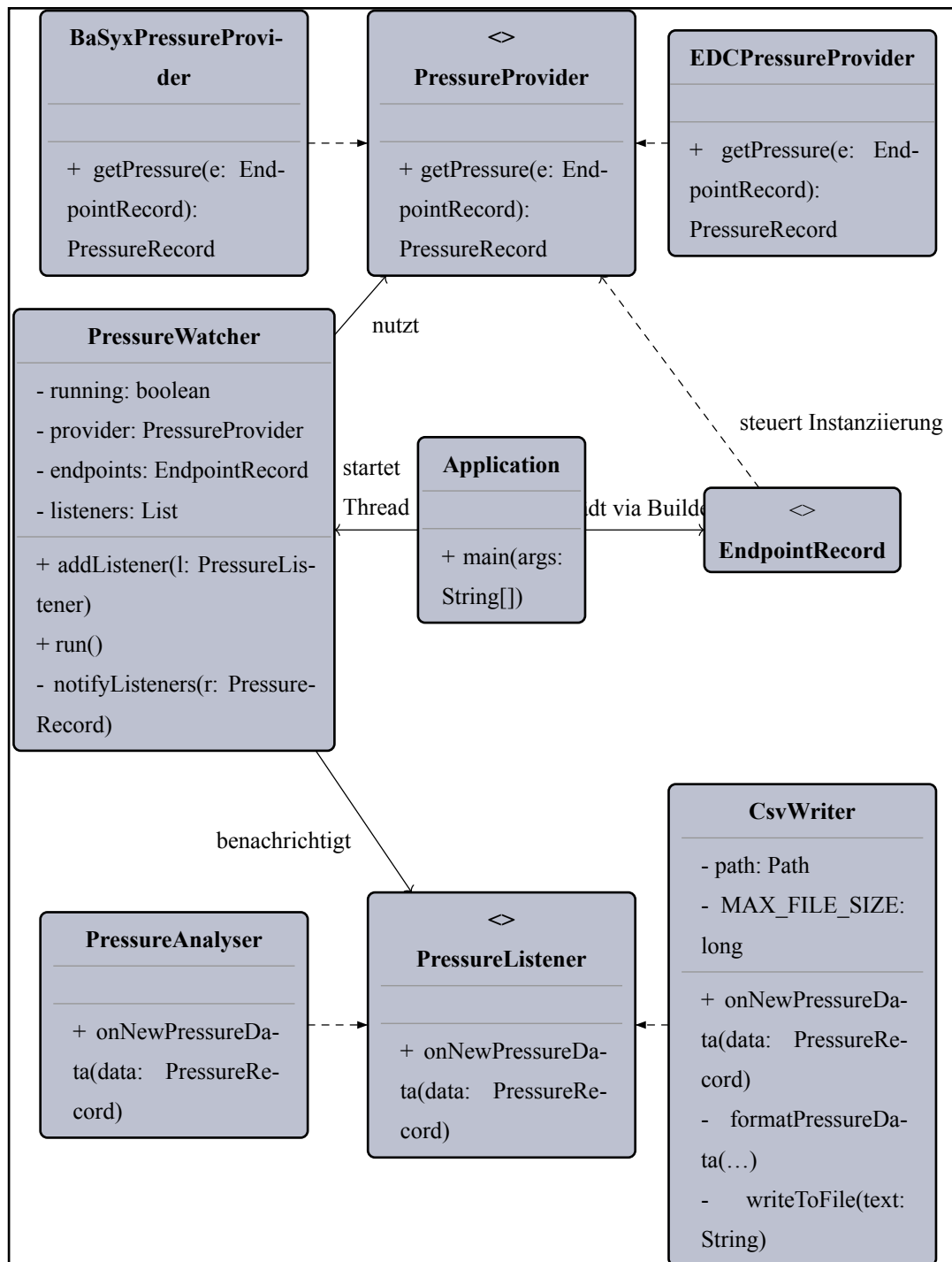


Abbildung 6: Klassendiagramm der Client-Applikation

4.5.1 Autorisierung und Vertragsaushandlung

Um als legitimer Datennutzer im souveränen Datenraum agieren zu können, muss der Client den vertraglich gesicherten Abrufmechanismus implementieren. Anstatt das proprietäre BaSyx-SDK für direkte Anfragen zu nutzen, generiert der Client native **HTTP-Requests**. Der Prozess erfolgt zweistufig: Zunächst ruft die Applikation über den lokalen Consumer-Konnektor die **Endpoint Data Reference (EDR)** ab. Die **Java Script Object Notation (JSON)**-Antwort dieses Aufrufs enthält einen temporären Autorisierungsschlüssel (*authorization*).

Dieser Schlüssel wird vom Client extrahiert, im Speicher vorgehalten und für alle nachfolgenden Abfragen in den **HTTP-Header** der GET-Requests injiziert. Mit dieser Autorisierung fragt der Client periodisch die spezifischen **AAS-Endpunkte** für die Druckwerte der Vakuumpumpe und des Abatement-Systems ab. Die aus dem **JSON-Payload** der Verwaltungsschale extrahierten Werte werden unmittelbar in strukturierte Laufzeitobjekte (*PressureRecord*) überführt. Innerhalb der Applikation greift anschließend der *PressureAnalyser*, welcher die abgerufenen Druckdifferenzen anhand deterministischer Schwellwerte evaluiert, um Anomalien wie Rohrverstopfungen algorithmisch zu detektieren.

4.5.2 Datenpersistierung

Für die nachgelagerte historische Analyse und die visuelle Aufbereitung müssen die flüchtigen *PressureRecord*-Objekte persistent gespeichert werden. Dies übernimmt die Komponente *CsvWriter*, welche ebenfalls als Observer in den Datenfluss eingehängt ist. Bei jedem Eintreffen neuer Sensordaten transformiert diese Klasse die Druckwerte von Pumpe und Abatement in einen kommagetrennten Datensatz. Um die zeitliche Chronologie zu wahren, wird jedem Eintrag ein präziser System-Zeitstempel (*Timestamp*) hinzugefügt. Die resultierende **CSV-Datei** wird fortlaufend in einem dedizierten lokalen Verzeichnis aggregiert.

4.5.3 Visualisierung

Die Präsentationsschicht des Systems wurde konzeptionell strikt von der Datenbeschaffung getrennt. Als Visualisierungs-Frontend kommt die Open-Source-Plattform Grafana zum Einsatz, welche als eigenständiger Docker-Container parallel zum Client betrieben

wird. Die informationstechnische Kopplung zwischen dem Java-Client und Grafana erfolgt über ein Shared Volume auf Basis des Host-Dateisystems.

Grafana greift über ein spezialisiertes **CSV**-Plugin lesend auf die vom Client generierte Datei zu. In der Weboberfläche von Grafana wurden spezifische Dashboards konfiguriert, welche die tabellarischen Zeitreihendaten in dynamische, leicht interpretierbare Liniendiagramme übersetzen. Diese Architektur ermöglicht ein echtzeitnahes Condition Monitoring der simulierten Assets und schließt die Wirkkette vom Datengenerator über den souveränen Datenraum bis hin zum visuellen Endpunkt erfolgreich ab.

5 Tests und Ergebnisse

5.1 Automatisierte Bereitstellung des Demonstrators

Die Zusammenführung der einzelnen Domänen zu einer funktionsfähigen Gesamtarchitektur erfordert eine präzise Orchestrierung der verteilten Systemkomponenten. Um eine deterministische und plattformunabhängige Reproduzierbarkeit (evaluiert unter einer Linux-Umgebung) zu gewährleisten, wurde die gesamte Startroutine in einem zentralen Shell-Skript (`run_full_setup.sh`) automatisiert (vgl. **Anhang D**). Der zugehörige Quellcode sowie die Konfigurationsdateien werden im GitLab der HTW Dresden (Zweig „Wintersemester2025_26“, (Rudolph, 2026)) vorgehalten.

Dieses Skript fungiert als übergeordneter Orchestrator und startet die Microservices in einer sequenziellen Reihenfolge:

1. **Datenanbieter:** Initialisierung der AAS-Laufzeitumgebung via Docker Compose.
2. **Datengenerierung:** Start des Python-Drucksimulators in einer isolierten virtuellen Umgebung (`venv`). Das Skript implementiert hierbei eine automatische Vorabprüfung der benötigten Systemabhängigkeiten (wie `simpy` und `requests`), um Laufzeitfehler frühzeitig abzufangen.
3. **Datenraum:** Bereitstellung des Provider- und Consumer-Konnektors.
4. **Datennutzer:** Automatisierter Build-Prozess und Start des Java-basierten AAS-Clients sowie des Grafana-Dashboards.

5.2 Evaluation der souveränen Datenpipeline

Nach der erfolgreichen, fehlerfreien Ausführung des Orchestrierungs-Skripts wurde der Gesamtsystem-Workflow – vom Asset bis zum Dashboard – in einem integralen Systemtest evaluiert. Dieser Test durchläuft alle informationstechnischen Stationen und verifiziert das Zusammenspiel der im Wintersemester entwickelten Komponenten.

5.2.1 Datengenerierung und -bereitstellung

Der Testlauf beginnt in der grafischen Oberfläche des Python-Simulators. Durch die manuelle Triggerung eines spezifischen Wartungsszenarios (beispielsweise „Rohrverstopfung“) wird die Generierung veränderter Druckwerte initiiert. Wie in **Abbildung 7** auf der linken Seite dargestellt lässt die Benutzeroberfläche sowohl die Auswahl vorgefertigter Wartungsszenarien zu als auch manuelle Eingabe von Werten.

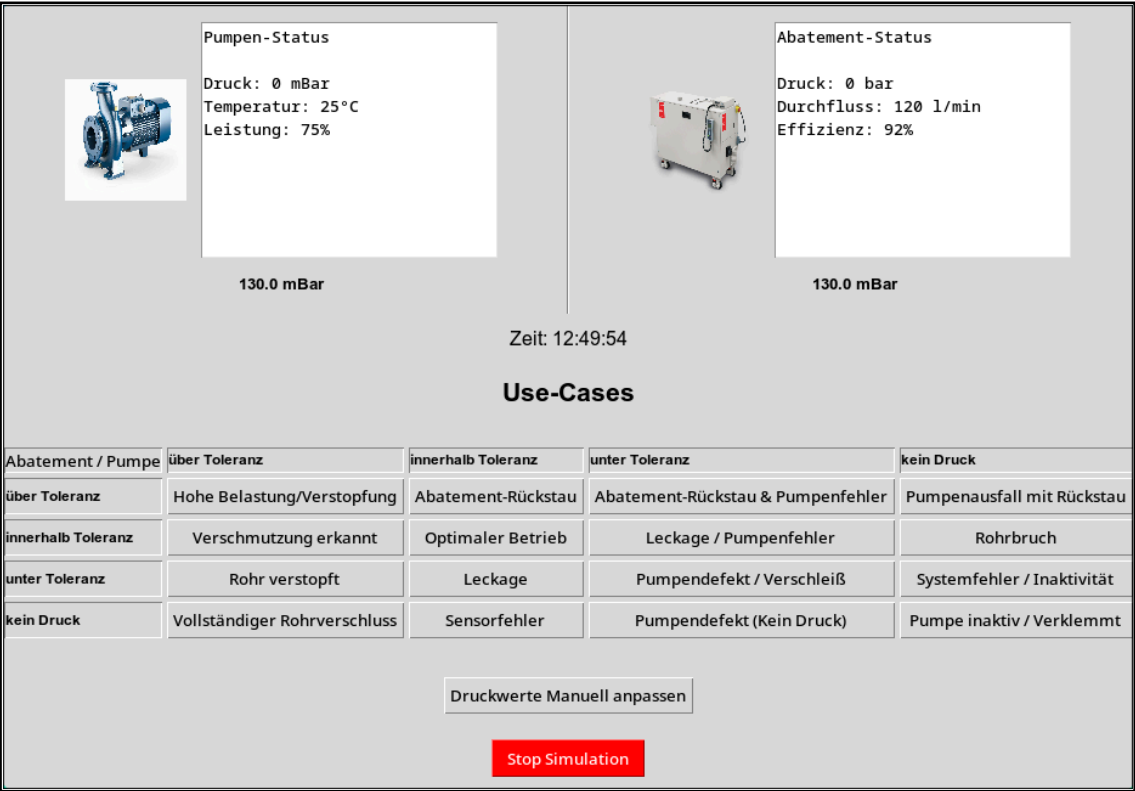


Abbildung 7: Benutzeroberfläche des Simulators

Die erfolgreiche Übermittlung via HTTP-PATCH an den lokalen BaSyx-Server lässt sich in der AAS-Web-UI (localhost:3000) verifizieren. Die Verwaltungsschale spiegelt die simulierten Druckdifferenzen in Echtzeit wider. Wie in **Abbildung 8** dargestellt, werden die Druckwerte direkt für die richtigen Submodeltemplates eingetragen. Ebenso wird die Struktur der Verwaltungsschale über die grafische Ansicht der Weboberfläche deutlich.

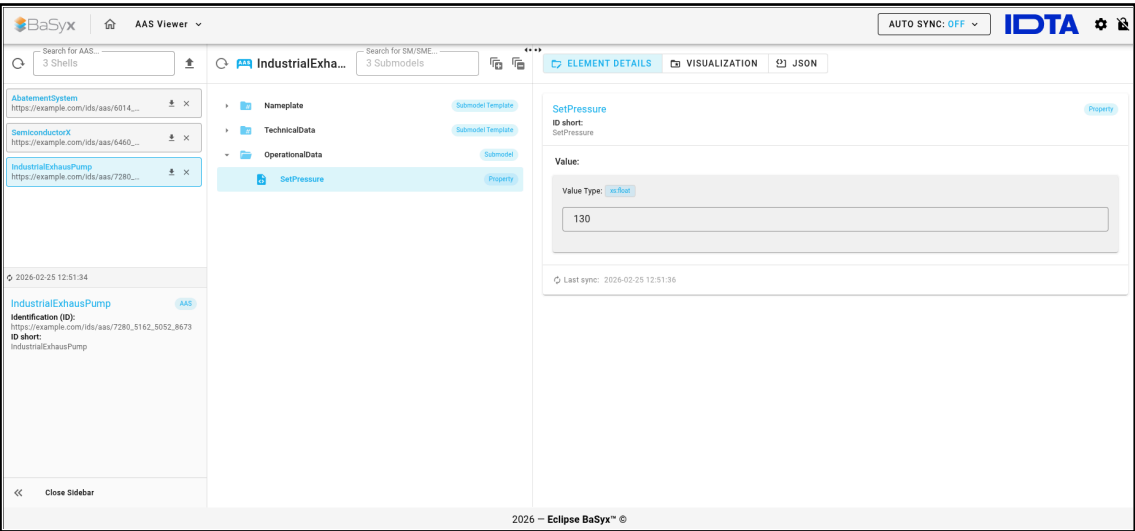


Abbildung 8: Webansicht der Verwaltungsschale

5.2.2 Vertragsaushandlung und Datentransfer

Der kritische Pfad dieses Systemtests ist die Überwindung der Domänengrenze. Der Java-Client initiiert vollautomatisiert die Vertragsaushandlung über den lokalen **EDC**-Consumer-Konnektor. Die Analyse der Docker-Logs des Konnektors bestätigt den erfolgreichen Austausch der kryptografischen Token. Der Client nutzt den erhaltenen Zugangsschlüssel, um die Live-Daten der **AAS** über den **EDC**-Proxy abzufragen. Der souveräne Datenraum blockiert in diesem Testaufbau gezielt alle unautorisierten Direktanfragen und lässt ausschließlich Zugriffe mit gültigem Token passieren, was die Einhaltung der Datensouveränität informationstechnisch belegt.

In **Anhang E** ist ein Auszug aus dem log der Client-Anwendung zu finden, welche mit dem Befehl `docker logs aas_consumer_client -f` ausgegeben werden können.

5.2.3 Auswertung, Persistierung und Visualisierung:

In der Endpunkt-Domäne validiert der Test die korrekte Verarbeitung der JSON-Antworten. Der im Java-Client integrierte `CsvWriter` extrahiert die ankommenden Druckwerte und schreibt diese iterativ, versehen mit einem korrekten Zeitstempel, in die lokale `pressure_data.csv`. Den finalen visuellen Nachweis des Workflows liefert das Grafana-Dashboard (`localhost:3001`). Die Web-Oberfläche liest die **CSV**-Datei erfolgreich über das angebundene Volume ein und transformiert die Datenpunkte in dynamische Liniendiagramme. In **Abbildung 9** zu sehen ist auf der linken Seite der historische Plot und auf der rechten Seite eine tabellarische Ansicht der Druckwerte.

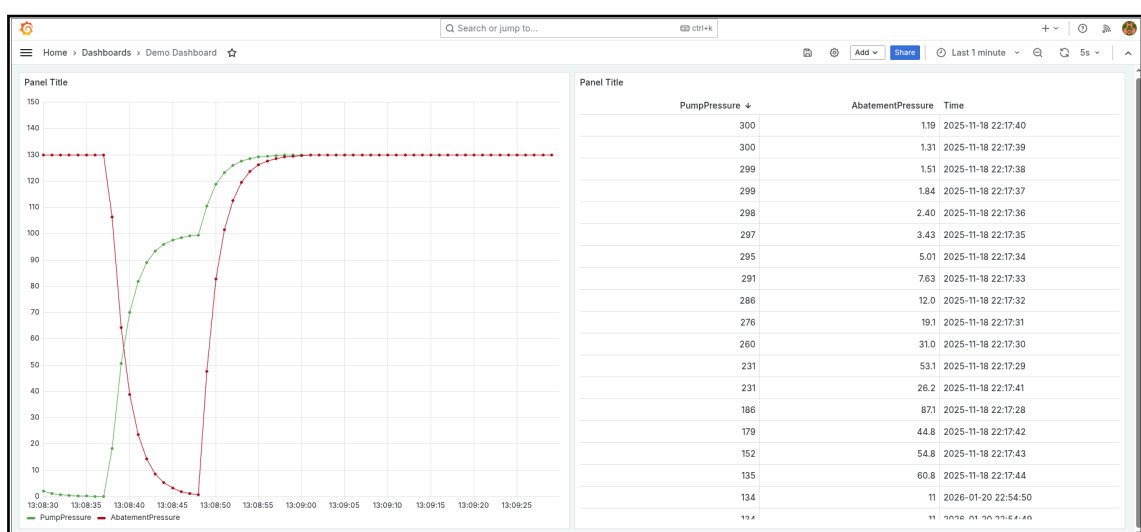


Abbildung 9: Grafana - Visualisierung der Druckwerte

6 Fazit und Ausblick

6.1 Zusammenfassung der Ergebnisse

Die vorliegende Arbeit adressierte die Herausforderung der ineffizienten, reaktiven Wartung von prozesskritischen Anlagen in der Sub-Fab von Halbleiterfabriken. Um fehleranfällige, manuelle Vor-Ort-Ablesungen zu überwinden, wurde ein fundiertes Konzept für einen standardisierten und datensouveränen Informationsaustausch entwickelt.

Im Rahmen der praktischen Umsetzung wurde zunächst eine voll funktionsfähige AAS-Laufzeitumgebung auf Basis von Eclipse BaSyx implementiert und durch einen prozessbasierten Python-Drucksimulator ergänzt. Dieser Simulator modelliert valide Livedaten für diverse Wartungsszenarien, von der allmählichen Rohrverschmutzung bis hin zum akuten Pumpenausfall. Die informationstechnische Überbrückung der Domänengrenzen erfolgte anschließend durch die Bereitstellung einer lokalen EDC-Infrastruktur. Eine eigens entwickelte Client-Anwendung übernimmt dabei die kontinuierliche Vertragsaushandlung, den gesicherten Datenabruf, die Persistierung in einem maschinenlesbaren CSV-Format sowie die Visualisierung über Grafana. Damit wurde die zentrale Forschungsfrage der Arbeit positiv beantwortet: Durch die Kombination aus genormter Verwaltungsschale und dem EDC als souveränem Datenproxy wird eine hochgradig skalierbare Infrastruktur geschaffen, die eine proaktive, unternehmensübergreifende Fernwartungsstrategie ermöglicht.

6.2 Kritische Reflexion und methodische Einordnung

Bezüglich der in der ursprünglichen Aufgabenstellung geforderten Analyse der Druckdifferenzen zur Anomalieerkennung wurde im Rahmen dieser Arbeit ein funktionaler Proof of Concept in Form einer deterministischen, schwellwertbasierten Auswertung implementiert. Die ursprüngliche Zielsetzung einer KI-gestützten Mustererkennung wurde architektonisch bewusst so adaptiert, dass die standardisierte Pipeline als technologischer Wegbereiter fungiert.

Anstatt ein monolithisches Machine-Learning-Modell isoliert in den Client zu integrieren, lag der Fokus auf der informationstechnischen Fundierung und Absicherung der verteilten Datenpipeline. Die implementierte deterministische Auswertung dient hierbei als essenzielle Baseline-Validierung. Sie erbringt den Nachweis, dass die durch den Simulator generierten Fehlerzustände latenzfrei über den EDC-Proxy übertragen und regelbasiert

klassifiziert werden können. Diese methodische Vorgehensweise ist ein zwingender Zwischenschritt, da sie erst die notwendige, strukturierte und validierte Datengrundlage für fortgeschrittene Verfahren des maschinellen Lernens schafft.

6.3 Ausblick und zukünftige Forschungsansätze

Für zukünftige Arbeiten bietet dieser evaluierte Demonstrator eine ideale Ausgangsbasis. Ein logischer nächster Schritt ist der Abgleich der generierten Simulationsdaten mit Messwerten einer realen Pumpe in Zusammenarbeit mit einem Praxispartner. Zudem kann die entwickelte Client-Anwendung um die tatsächliche Anbindung an ein produktives ERP-System des Herstellers erweitert werden.

Darauf aufbauend sollte die Integration der avisierten KI-gestützten Analyseverfahren erfolgen. Für die Klassifizierung der komplexen, schleichenden Anomalien – wie etwa einer progressiven Rohrverschmutzung – empfiehlt sich perspektivisch der Einsatz von etablierten überwachten Lernverfahren, beispielsweise Random Forests. Um darüber hinaus eine holistische Vorhersage auf Basis der kontinuierlichen Zeitreihen und Druckhistorien zu realisieren, bietet sich die Integration rekurrenter neuronaler Netze, insbesondere Long Short-Term Memory Architekturen, an. Aufgrund der generischen Konzeption der entwickelten Pipeline können diese fortgeschrittenen KI-Modelle zukünftig als eigenständige, souveräne Datenkonsumenten in den EDC-Datenraum eingebunden werden, ohne tiefgreifende strukturelle Anpassungen an der bestehenden Infrastruktur vornehmen zu müssen.

Literatur

- Admin Shell IO. (2024,). *Submodel Templates*. <https://github.com/admin-shell-io/submodel-templates>
- Asset Administration Shell Specification - Part 1: Metamodel*. (o. J.).
- Bader, S. (2019). *Details of the Asset Administration Shell - Part 1*.
- Bosch. (2021, Juni 7). *Fabrik der Zukunft: Halbleiterwerk Dresden*. <https://www.bosch.com/de/stories/fabrik-der-zukunft-halbleiterwerk-dresden/>
- Eclipse Foundation. (2017, Dezember 21). *Eclipse BaSyx Projektseite*. <https://projects.eclipse.org/projects/dt.basyx>
- Eclipse Foundation. (2023,). *Eclipse BaSyx Wiki*. <https://wiki.basyx.org/en/latest>
- Eclipse Foundation. (2024c,). *Eclipse BaSyx*. <https://eclipse.dev/basyx/>
- Eclipse Foundation. (2024e,). *Eclipse BaSyx auf GitHub*. <https://github.com/eclipse-basyx>
- Eclipse Foundation. (2024f,). *Eclipse BaSyx Java Server SDK*. <https://github.com/eclipse-basyx/basyx-java-server-sdk#>
- Eclipse Foundation. (2024a,). *Eclipse Data Space Connector*. <https://www.eclipse.org/edc/>
- Eclipse Foundation. (2024b,). *Eclipse Data Space Connector: Architektur*. <https://www.eclipse.org/edc/docs/developer/architecture/>
- Eclipse Foundation. (2024g,). *Package Explorer*. <https://github.com/eclipse-aaspe/package-explorer>
- Eclipse Foundation. (2024d,). *Über Eclipse BaSyx*. <https://eclipse.dev/basyx/about/>
- Eclipse Foundation. (2026,). *Eclipse EDC Samples*. <https://github.com/eclipse-edc/Samples>
- Hilscher. (2023, März 23). *Die Halbleiterfertigung und ihre Maschinen*. <https://www.hilscher.com/de/der-hilscher-blog/blog-die-halbleiterfertigung-und-ihre-maschinen>
- International Data Spaces Association. (2024,). *International Data Spaces*. <https://internationaldataspaces.org/>
- Krauß, J., Schmetz, A., Fitzner, A., Ackermann, T., Pouls, K. B., Hülsmann, T.-H., Roth, D., Gehring, J., Hamacher, N. C., Jaspers, W., Mohring, L., Rube, N., Tübke, J., Kies, A. D., Kreppein, A., Abramowski, J.-P., Schmitt, R. H., Baum, C., Brandstetter,

- A., ... Kampker, A. (2023, Januar 9). *Der Digitale Zwilling in der Batteriezellfertigung*. <https://publica.fraunhofer.de/entities/publication/05cc0964-10f5-43fd-b8b5-6b4d05bad628>
- Nitzschner, J., & Krauß, R. (2024, November 28). *Netzwerk im Rampenlicht: Mikroelektronik, Software und Zulieferunternehmen arbeiten gemeinsam an der ressourceneffizienten Sub Fab*. <https://silicon-saxony.de/netzwerk-im-rampenlicht-mikroelektronik-software-und-zulieferunternehmen-arbeiten-gemeinsam-an-der-ressourceneffiziente-n-sub-fab/>
- Plattform Industrie 4.0. (2023,). *Hintergrund*. <https://www.plattform-i40.de/IP/Navigation/DE/Plattform/Hintergrund/hintergrund.html>
- Rudolph, J. (2026,). *FuE-SemiconductorX Quellcode*. <https://iversion.informatik.htw-dresden.de/s88598/fue-semiconductorx>
- Salari, A. (2019). *The Asset Administration Shell: Implementing digital twins for use in Industrie 4.0*. <https://www.plattform-i40.de/IP/Redaktion/EN/Downloads/Publikation/VWSiD%20V2.0.html>

A Struktureller Aufbau der Verwaltungsschale

- **Shell:** SemiconductorX (Parent)
 - **Sub:** Nameplate [Template-Git]
- **Shell:** IndustrialExhausPump (Child)
 - **Sub:** Nameplate [Template-Git]
 - **Sub:** TechnicalData [Template-Git]
 - **Sub:** OperationalData
 - **Prop:** ActualPressure
- **Shell:** AbatementSystem (Child)
 - **Sub:** Nameplate [Template-Git]
 - **Sub:** TechnicalData [Template-Git]
 - **Sub:** OperationalData
 - **Prop:** ActualPressure

B Docker-Compose für BaSyx Off-The-Shelf Komponenten

```
services:
  mongo:
    image: mongo:5.0.10
    # Provide mongo config
    restart: always
    environment:
      MONGO_INITDB_ROOT_USERNAME: mongoAdmin
      MONGO_INITDB_ROOT_PASSWORD: mongoPassword
    # Set health checks to wait until mongo has started
    #volumes:
    # - ./data-vol:/data/db
    healthcheck:
      test: mongo
      interval: 10s
      start_period: 5s
      retries: 5
    # Maps tcp port to host
    #ports:
    # - 27017:27017

  mqtt:
    image: eclipse-mosquitto:2.0.15
    ports:
      - 1884:1884
    volumes:
      - ./mosquitto:/mosquitto/config
    healthcheck:
      test: ["CMD-SHELL", mosquitto_sub -p 1884 -t 'topic' -C 1 -E -i probe
-W 3]
      interval: 5s
      retries: 3
      start_period: 1s
      timeout: 10s

  aas-env:
    image: eclipsebasyx/aas-environment:2.0.0-SNAPSHOT
    volumes:
      - ./aas-env.properties:/application/application.properties
```

```
- ./aas:/application/aas
ports:
  - 8081:8081
depends_on:
  mongo:
    condition: service_healthy
  mqtt:
    condition: service_healthy
  aas-registry:
    condition: service_healthy
  sm-registry:
    condition: service_healthy

aas-registry:
  image: eclipsebasyx/aas-registry-log-mongodb:2.0.0-SNAPSHOT
  ports:
    - 8082:8080
  volumes:
    - ./aas-registry.yml:/workspace/config/application.yml
  depends_on:
    mongo:
      condition: service_healthy

sm-registry:
  image: eclipsebasyx/submodel-registry-log-mongodb:2.0.0-SNAPSHOT
  ports:
    - 8083:8080
  volumes:
    - ./sm-registry.yml:/workspace/config/application.yml
  depends_on:
    mongo:
      condition: service_healthy

aas-discovery:
  image: eclipsebasyx/aas-discovery:2.0.0-SNAPSHOT
  ports:
    - 8084:8081
  volumes:
    - ./aas-discovery.properties :/application/application.properties
  depends_on:
```

```
    mongo:
      condition: service_healthy

basyx-client:
  build:
    context: ./basyx-client
    dockerfile: Dockerfile
  container_name: basyx-client
  env_file: ./basyx-client.env
  network_mode: host
  depends_on:
    aas-env:
      condition: service_healthy

aas-web-ui_v2:
  image: eclipsebasyx/aas-gui:SNAPSHOT
  container_name: aas-web-ui_v2
  ports:
    - "3000:3000"
  environment:
    AAS_REGISTRY_PATH: "http://localhost:8082/shell-descriptors"
    SUBMODEL_REGISTRY_PATH: "http://localhost:8083/submodel-descriptors"
    AAS_DISCOVERY_PATH: "http://localhost:8084/lookup/shells"
    AAS_REPO_PATH: "http://localhost:8081/shells"
    SUBMODEL_REPO_PATH: "http://localhost:8081/submodels"
    CD_REPO_PATH: "http://localhost:8081/concept-descriptions"
  restart: always
  depends_on:
    aas-env:
      condition: service_healthy
```

C Drucksimulator Python Script

```
import tkinter as tk
from tkinter import ttk
import simpy
import threading
import time
from datetime import datetime
import requests
from PIL import Image, ImageTk

# Use-Case Labels and Values

button_labels = [
    "Hohe Belastung/Verstopfung",
    "Abatement-Rückstau",
    "Abatement-Rückstau & Pumpenfehler",
    "Pumpenausfall mit Rückstau",
    "Verschmutzung erkannt",
    "Optimaler Betrieb",
    "Leckage / Pumpenfehler",
    "Rohrbruch",
    "Rohr verstopft",
    "Leckage",
    "Pumpendefekt / Verschleiß",
    "Systemfehler / Inaktivität",
    "Vollständiger Rohrverschluss",
    "Sensorfehler",
    "Pumpendefekt (Kein Druck)",
    "Pumpe inaktiv / Verklemmt"
]

use_case_values = {
    0: (130, 130),
    1: (110, 130),
    2: (70, 130),
    3: (0, 130),
    4: (130, 110),
    5: (100, 100),
    6: (70, 100),
```

```
    7: (0, 100),
    8: (130, 70),
    9: (110, 70),
   10: (70, 70),
   11: (0, 70),
   12: (130, 0),
   13: (100, 0),
   14: (70, 0),
   15: (0, 0)
}

header_labels = ["über Toleranz", "innerhalb Toleranz", "unter Toleranz",
"kein Druck"]

class SimulationApp:
    def __init__(self, root, env):
        self.root = root
        self.env = env
        # Control flag for shutting down
        self.running = True
        # Shared variables
        self.actual_pressure_pump = 0
        self.actual_pressure_abatement = 0
        self.target_pressure_pump = 0
        self.target_pressure_abatement = 0
        self.flag_pump = 0
        self.flag_abatement = 0
        self.use_cases(0)

        # REST-API
        # TODO: get the REST-API endpoints dynamicaly (stuck here till
        connection with aas-server is established)
        (self.pump_pressure_url, self.abatement_pressure_url) =
self.get_rest_api_endpoints()

        # Build GUI
        self.setup_gui()

        # Simulation and REST in seperate threads
        self.sim_thread = threading.Thread(target=self.run_simulation,
```



```
daemon=True)
    self.sim_thread.start()
    self.rest_thread = threading.Thread(target=self.rest_update,
daemon=True)
    self.rest_thread.start()

    # Register a Process for SimPy
    self.pump_proc = env.process(self.pump(env))

    # Schedule GUI Update
    self.root.after(100, self.update_gui)

    # Set Closing callback
    self.root.protocol("WM_DELETE_WINDOW", self.on_close)

    # ==> Everything from Simulation <==
    # Pump thread is for updating Values from the AAS
    def pump(self,env):
        factor = 0.01
        while self.running:
            self.actual_pressure_pump += (self.target_pressure_pump -
self.actual_pressure_pump) * factor
            self.actual_pressure_abatement += (self.target_pressure_abatement
- self.actual_pressure_abatement) * factor

            self.actual_pressure_pump = round(self.actual_pressure_pump, 3)
            self.actual_pressure_abatement =
round(self.actual_pressure_abatement, 3)

            # Näherungs-Toleranz definieren
            epsilon = 0.1

            # Wenn nahe genug: exakt setzen
            if abs(self.actual_pressure_pump - self.target_pressure_pump)
< epsilon:
                self.actual_pressure_pump = self.target_pressure_pump

                if abs(self.actual_pressure_abatement -
self.target_pressure_abatement) < epsilon:
```

```
        self.actual_pressure_abatement = self.target_pressure_abatement

        yield env.timeout(2)

def run_simulation(self):
    try:
        while self.running:
            self.env.run(until=self.env.now + 10) # Run in increments
            time.sleep(0.1)
    except Exception as e:
        print(f"Simulation error: {e}")
    finally:
        print("Simulation stopped")

def get_use_case_label(self, index):
    return button_labels[index] if index < len(button_labels) else
"Unbekannt"

def use_cases(self, val):
    (self.target_pressure_pump, self.target_pressure_abatement) =
use_case_values.get(val, (0, 0)) # Standardwerte für unbekannte Fälle

# ==> Everything for GUI <==
def setup_gui(self):
    main_frame = ttk.Frame(self.root)
    main_frame.pack(fill="both", expand=True)

    # left Side (Pumpe)
    self.left_frame = ttk.Frame(main_frame, padding="10")
    top_frame_pump = ttk.Frame(self.left_frame)
    # Picture Pumpe
    im = Image.open("images/pump.jpg")
    im = im.resize((100, 100), Image.NEAREST)
    tk_image = ImageTk.PhotoImage(im)
    self.pump_image = tk_image
    self.label_image_pump = tk.Label(top_frame_pump,
image=self.pump_image)
    self.label_image_pump.pack(side="left", padx=5)
    # Stats Pump
```

```
self.label_stats_pump = tk.Text(top_frame_pump, width=30, height=10,
wrap="word")
self.label_stats_pump.insert("1.0", "Pumpen-Status\n\nDruck: 0
mBar\nTemperatur: 25°C\nLeistung: 75%")
self.label_stats_pump.pack(side="left", padx=5)
top_frame_pump.pack(pady=5)
# Pressure Pump
self.label_pressure_pump = tk.Label(self.left_frame, text="0 mBar",
font=("Arial", 10, "bold"), anchor="center")
self.label_pressure_pump.pack(pady=5)
self.left_frame.pack(side="left", fill="both", expand=True)

# Separator
separator = ttk.Separator(main_frame, orient="vertical")
separator.pack(side="left", fill="y", padx=5)

# Right side (Abatement)
self.right_frame = ttk.Frame(main_frame, padding="10")
top_frame_abatement = ttk.Frame(self.right_frame)
# Picture Abatement
im = Image.open("images/abatement.png")
im = im.resize((100, 100), Image.NEAREST)
tk_image = ImageTk.PhotoImage(im)
self.image_abatement = tk_image
self.label_image_abatement = tk.Label(top_frame_abatement,
image=self.image_abatement)
self.label_image_abatement.pack(side="left", padx=5)
# Stats
self.label_stats_abatement = tk.Text(top_frame_abatement, width=30,
height=10, wrap="word")
self.label_stats_abatement.insert("1.0", "Abatement-Status\n\nDruck:
0 bar\nDurchfluss: 120 l/min\nEffizienz: 92%")
self.label_stats_abatement.pack(side="left", padx=5)
top_frame_abatement.pack(pady=5)
# Pressure Abatement
self.label_pressure_abatement = tk.Label(self.right_frame, text="0
mBar", font=("Arial", 10, "bold"), anchor="center")
self.label_pressure_abatement.pack(pady=5)
self.right_frame.pack(side="left", fill="both", expand=True)
```

```
# Time Display
self.time_frame = ttk.Frame(self.root)
self.time_frame.pack(fill="x", pady=10)
self.label_time = ttk.Label(self.time_frame, text="Zeit: 00:00:00",
font=("Arial", 12))
self.label_time.pack()

# Buttons to Trigger Use-Cases
title_frame = ttk.Frame(self.root)
title_frame.pack(pady=10)
    ttk.Label(title_frame, text="Use-Cases", font=('Arial', 16,
'bold')).pack() # Überschrift
self.button_frame = ttk.Frame(self.root)
self.button_frame.pack(pady=20)
self.create_buttons_with_grid(self.button_frame)

# Manuel Pressure Adjustment (PopUp)
popup_btn = ttk.Button(root, text=f"Druckwerte Manuell anpassen",
command=self.open_popup)
popup_btn.pack(pady=10)

# Button to Stop Simulation
    stop_button = tk.Button(root, text=f"Stop Simulation",
command=self.on_close, fg='white', bg='red')
    stop_button.pack(pady=10)

def create_buttons_with_grid(self, parent_frame):
    # Tabelle für Use-Cases mit Toleranz-Bezeichnungen
        ttk.Label(parent_frame, text="Abatement / Pumpe",
relief="ridge").grid(row=0, column=0, padx=2, pady=2, sticky="nsew")
        for col_idx, header_text in enumerate(header_labels):
            ttk.Label(parent_frame, text=header_text, font=('Arial', 9,
'bold'), relief="ridge") \
                .grid(row=0, column=col_idx + 1, padx=2, pady=2, sticky="nsew")
        for row_idx, side_text in enumerate(header_labels):
            ttk.Label(parent_frame, text=side_text, font=('Arial', 9,
'bold'), relief="ridge") \
                .grid(row=row_idx + 1, column=0, padx=2, pady=2, sticky="nsew")
    max_internal_columns = 4
```

```
row_offset = 1
column_offset = 1
# Buttons mit entprecheneden Labels für Use-Cases (in Tabelle
eintragen)
for i in range(0, len(button_labels)):
    button_relative_row = i // max_internal_columns
    button_relative_column = i % max_internal_columns
    grid_row = button_relative_row + row_offset
    grid_column = button_relative_column + column_offset
    btn = ttk.Button(parent_frame, text=self.get_use_case_label(i),
command=lambda i=i: self.use_cases(i))
    btn.grid(row=grid_row, column=grid_column, padx=2, pady=2,
sticky="nsew")

def open_popup(self):
    def popup_on_close():
        try:
            val1 = int(entry1.get())
            val2 = int(entry2.get())
            if 0 <= val1 <= 300 and 0 <= val2 <= 300:
                # print(f"\nEingegebene Werte: {val1}, {val2}\n")
                self.target_pressure_pump = val1
                self.target_pressure_abatement = val2
            else:
                print("Druckwerte out of bound!")
        except ValueError:
            print("Bitte gültige Zahlen eingeben.")
        popup.destroy()
    popup = tk.Toplevel(self.root)
    popup.title("Gewünschte Druckwerte eingeben")
    popup.geometry("300x200")
    tk.Label(popup, text="Druck Pumpe: [mBar]").pack(pady=5)
    entry1 = tk.Entry(popup)
    entry1.pack(pady=5)
    tk.Label(popup, text="Druck Abatement: [mBar]").pack(pady=5)
    entry2 = tk.Entry(popup)
    entry2.pack(pady=5)
    close_btn = ttk.Button(popup, text=f"Übernehmen & Schließen",
command=popup_on_close)
    close_btn.pack(pady=10)
```

```
# Updating Textboxes and values to Display
def update_gui(self):
    if not self.running:
        self.root.quit()
        return

    # Update GUI elements with the current state of the simulation
    self.label_pressure_pump.config(text =
f"{self.actual_pressure_pump:.1f} mBar")
    self.label_pressure_abatement.config(text =
f"{self.actual_pressure_abatement:.1f} mBar")
    self.label_time.config(text=f"Zeit: {datetime.now().strftime('%H:
%M:%S')}")
    self.root.after(200, self.update_gui) # Schedule next GUI update

def on_close(self):
    self.running = False
    self.root.quit()

# ==> Everything for REST-API <==
def get_rest_api_endpoints(self):
    pump_pressure_url = "http://localhost:
8081/submodels/ aHR0cHM6Ly9leGFtcGxlLmNvbS9pZHMvc20vMzI1NF81MTYyXzUwNTJf
MTk5NQ==/submodel-elements/SetPressure/$value"
    abatement_pressure_url = "http://localhost:
8081/submodels/ aHR0cHM6Ly9leGFtcGxlLmNvbS9pZHMvc20vNjE2NF81MTYyXzUwNTJf
NDE3Nw==/submodel-elements/SetPressure/$value"
    return (pump_pressure_url, abatement_pressure_url)

def rest_update(self):
    headers = {"accept": "application/json", "Content-Type": "application/
json"}
    while self.running:
        try:
            # Pump Pressure
            pump_response = requests.patch(self.pump_pressure_url,
headers=headers, data=f'"{self.actual_pressure_pump}"')
```

```
        if pump_response.status_code == 204:
            print(f"Pump pressure set to {self.actual_pressure_pump}
mBar")
        else:
            print(f"Failed to set pump pressure:
{pump_response.status_code} - {pump_response.text}")

        # Abatement Pressure
        abatement_response
= requests.patch(self.abatement_pressure_url, headers=headers,
data=f'"{self.actual_pressure_abatement}"')
        if abatement_response.status_code == 204:
            print(f"Abatement pressure set to
{self.actual_pressure_abatement} mBar")
        else:
            print(f"Failed to set abatement pressure:
{abatement_response.status_code} - {abatement_response.text}")

    except requests.exceptions.RequestException as e:
        print(f"REST API error: {e}")

    time.sleep(1)

# ==> Setup Environment <==
root = tk.Tk() # Tkinter
env = simpy.Environment() # SimPy
app = SimulationApp(root, env) # Simulation App

root.mainloop()
try:
    root.mainloop()
finally:
    app.running = False

print("Finished?")
```

D Setupscript zum starten des Demonstrators

```
#!/bin/bash

# BaSyx AAS Services starten
echo ">>> Starte BaSyx AAS Services..."
cd ./basyx_aas_services
docker compose up -d
cd ..

# Simulator Starten
echo ">>> Starte den Pumpensimulator"
cd ./python_simulator/
source ./venv/bin/activate

echo ">>> Prüfe Python-Abhängigkeiten..."
python3 -c "import simpy, requests, PIL, tkinter" 2>/dev/null
if [ $? -ne 0 ]; then
    echo "FEHLER: Abhängigkeiten fehlen!"
    exit 1
fi
pip show Pillow >/dev/null 2>&1 || echo "Warnung: Pillow nicht lokal
installiert."

python3 drucksimulator.py >simulator.log 2>&1 &
cd ..

# EDC starten
echo ">>> Starte EDC..."
cd ./EDC
docker compose up -d
cd ..

# AAS Client bauen und starten
echo ">>> Baue und starte AAS Client..."
cd ./aas_client
docker compose up -d --build
cd ..

echo ">>> Alle Setups wurden verarbeitet."
```



```
docker ps --format "{{.Names}}"
```

```
# Websites aufrufen+ "/submodel-elements/SetPressure/$value"
```

```
echo " "
```

```
echo "Hier gehts zu den Websites:"
```

```
echo "BaSyx AAS-Package Viewer: http://localhost:3000/"
```

```
echo "Grafana: http://localhost:3001/"
```

E Logs des AAS Clients

```
13:40:22.771 [Thread-0] INFO de.htw_dresden.jrudolph.aas_consumer_client.processing.PressureWatcher -- ==> Starting AAS-Client
13:40:22.773 [Thread-0] INFO de.htw_dresden.jrudolph.aas_consumer_client.processing.EdcNegotiationService -- Starte EDC-Vertragsaushandlung für Asset: aas-server
13:40:22.846 [Thread-0] INFO de.htw_dresden.jrudolph.aas_consumer_client.processing.EdcNegotiationService -- Katalog abgerufen. Offer-ID: YWFzLWNvbnRyYWN0:YWFzLXNlcnZlcnZlcg==:MGVkb0RmNDAtMmJiYS00ZjU4LWIyYzYtYzc4ZDZjMjk0ZTU0
13:40:22.854 [Thread-0] INFO de.htw_dresden.jrudolph.aas_consumer_client.processing.EdcNegotiationService -- Vertragsverhandlung initiiert. Negotiation-ID: df1cceb0-fb1b-4b36-9b18-a038fe7986c3
13:40:24.862 [Thread-0] INFO de.htw_dresden.jrudolph.aas_consumer_client.processing.EdcNegotiationService -- Vertrag erfolgreich geschlossen. Agreement-ID: 0be9c602-c9b3-4092-9c54-a84470b7e102
13:40:24.868 [Thread-0] INFO de.htw_dresden.jrudolph.aas_consumer_client.processing.EdcNegotiationService -- Transfer initiiert. Transfer-ID: 849e0a0a-a3d1-48ee-97fd-10f8c5530d02
13:40:26.876 [Thread-0] INFO de.htw_dresden.jrudolph.aas_consumer_client.processing.EdcNegotiationService -- Transfer ist im Status STARTED. EDR ist bereit.
13:40:26.879 [Thread-0] INFO de.htw_dresden.jrudolph.aas_consumer_client.processing.EdcNegotiationService -- Erfolgreich Authorization-Key aus EDR extrahiert!
13:40:26.880 [Thread-0] INFO de.htw_dresden.jrudolph.aas_consumer_client.processing.CsvWriter -- ==> Kombiniere Kombiniere
13:40:26.888 [Thread-0] INFO de.htw_dresden.jrudolph.aas_consumer_client.processing.CsvWriter -- ==> new Data to CSV-File: 0.00;0.00;2026-02-25 13:40:26

13:40:27.890 [Thread-0] INFO de.htw_dresden.jrudolph.aas_consumer_client.processing.pressure.EDCPressureProvider -- http://localhost:19291/public/submodels/aHR0cHM6Ly9leGFtcGxlLmNvbS9pZHMvc20vMzI1NF81MTYyXzUwNTJfMTk5NQ/submodel-elements/SetPressure/$value
13:40:27.896 [Thread-0] INFO de.htw_dresden.jrudolph.aas_consumer_client.processing.CsvWriter -- ==> Kombiniere Kombiniere
13:40:27.897 [Thread-0] INFO de.htw_dresden.jrudolph.aas_consumer_client.processing.CsvWriter -- ==> new Data to CSV-File: 130.00;130.00;2026-02-25 13:40:27

13:40:28.897 [Thread-0] INFO de.htw_dresden.jrudolph.aas_consumer_client.processing.pressure.EDCPressureProvider -- http://localhost:19291/public/submodels/aHR0cHM6Ly9leGFtcGxlLmNvbS9pZHMvc20vMzI1NF81MTYyXzUwNTJfMTk5NQ/submodel-elements/SetPressure/$value
13:40:28.902 [Thread-0] INFO de.htw_dresden.jrudolph.aas_consumer_client.processing.CsvWriter -- ==> Kombiniere Kombiniere
13:40:28.902 [Thread-0] INFO de.htw_dresden.jrudolph.aas_consumer_client.processing.CsvWriter -- ==> new Data to CSV-File: 130.00;130.00;2026-02-25 13:40:28
```