# Konzept zur Modularisierung von Komponenten des Digitalen Prozesszwillings

**Concept for the Modularization of Components of the Digital Process Twin**

Bachelorarbeit von Robert Knobloch (Matrikelnummer: 2367563)
Tag der Einreichung: 8. Oktober 2022

Betreuer: Vladimir Kutscher
Betreuer: Christian Plesker

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Maschinenbau

Fachgebiet Datenverarbeitung in der Konstruktion

Bachelor Thesis

für

Robert Knobloch [Matr. 2367563]

TECHNISCHE UNIVERSITÄT DARMSTADT

# Konzept zur Modularisierung von Komponenten des Digitalen Prozesszwillings

*Concept for modularization of components of the digital process twin*

Im Forschungsprojekt Space Factory 4.0 wurden Konzepte einer In-Orbit-Produktion modularer Satelliten auf Basis von Industrie 4.0 erstellt. Das Projekt AI-In-Orbit-Factory erweitert die Ergebnisse von Space Factory 4.0 um einen Digitalen Prozesszwilling zur autonomen Steuerung und Simulation der Prozessabläufe in der In-Orbit Factory. Dazu wurde ein Digitaler Prozesszwilling entwickelt und mit einer autonome Montageplanung ausgestattet.

Im Rahmen dieser Bachelorarbeit soll aufbauend auf der beschriebenen Grundlage das Konzept des Digitalen Prozesszwillings mittels der Software-Container Technologie flexibilisiert werden. Dadurch können die einzelnen Module des Digitalen Prozesszwillings flexibel miteinander verknüpft und unabhängig von der ausführenden Ressource eingesetzt werden. Damit lässt sich das Konzept auf neue Systeme und Anwendungsfälle übertragen. Zudem verbessert dies die Skalierbarkeit, indem die einzelnen Software-Container auf externe Ressourcen ausgelagert werden können. Das entwickelte Konzept soll in einem konkreten Anwendungsfall im Kontext der In-Orbit Fabrik umgesetzt werden. Die prototypische Implementierung dient der Verifikation und Validierung des Konzepts.

Arbeitspakete:

- Recherche zum Stand der Technik und Forschung
- Auswahl einer geeigneten Container-Technologie
- Ableiten von Anforderungen an die Methodik
- Entwicklung eines Konzepts zur Flexibilisierung des Digitalen Prozesszwillings mittels der Software-Container Technologie
- Prototypische Implementierung
- Verifikation und Validierung
- Dokumentation und Präsentation der Ergebnisse

Tag der Ausgabe: 21.04.2022

Betreuer: Christian Plesker, M.Sc. und Vladimir Kutscher, M.Sc.

Prof. Dr.-Ing. R. Anderl

Fachgebiet Datenverarbeitung in der Konstruktion

Department of Computer Integrated Design

Prof. Dr.-Ing. Reiner Anderl

Otto-Berndt-Straße 2
64287 Darmstadt

Christian Plesker, M.Sc.
plesker@dik.tu-darmstadt.de
Vladimir Kutscher, M.Sc.
kutscher@dik.tu-darmstadt.de

Tel. +49 6151 16 -21868
Fax +49 6151 16 - 21793
anderl@dik.tu-darmstadt.de

Datum
21.04.2022

## Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Robert Knobloch, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 7. Oktober 2022

Robert Knobloch

# Zusammenfassung

Digitale Prozesszwillinge (DPTs) sind vernetzte Software-Systeme im Industrie-4.0-Kontext, die Produktionsprozesse digital abbilden, überwachen und steuern. Damit können Produktivitätssteigerungen erzielt und Wertschöpfungsketten flexibilisiert werden. Allerdings kann der Betrieb von DPTs zu sehr hohen Anforderungen an die ausführende Rechenressource führen. Zu diesen Anforderungen gehören eine hohe und variable benötigte Rechenleistung, Zuverlässigkeit und Ausfallsicherheit. Diese können mit monolithischen DPT-Architekturen nicht immer ausreichend erfüllt werden. Außerdem sind monolithische DPTs schwer wartbar, da einzelne Teile nur umständlich ersetzt werden können. Wenige wissenschaftliche Arbeiten befassten sich aber bisher mit nicht-monolithischen Digitalen Zwillingen.

Der Aufbau eines DPT in modularer statt monolithischer Architektur verspricht eine längere Lebensdauer durch bessere Wartbarkeit und Austauschbarkeit der DPT-Komponenten. Außerdem lassen sich die Module mittels Software-Virtualisierung durch Container von der physischen Hardware trennen und so flexibel zwischen beliebigen Hosts bewegen. Diese Portabilität erlaubt die horizontale Skalierung des DPT, Ausfalltoleranz sowie eine Auslagerung von Modulen in die Cloud. Austauschbarkeit und Wiederverwertbarkeit der Module sowie Portabilität der Container-Technologie ermöglichen zudem ein herstellerübergreifendes Ökosystem für den DPT.

In dieser Arbeit werden die Komponenten eines DPT modularisiert und containerisiert. Es wird eine Architektur für diesen containerisierten, modularisierten DPT (CMDPT) geschaffen, welche insbesondere die benötigte Infrastruktur für den Betrieb spezifiziert. Das Konzept wird in einer prototypischen Implementierung umgesetzt. Die Validierung des Konzepts zeigt, dass der CMDPT neue Anwendungsgebiete für den DPT schafft, die zugrundeliegenden Technologien aber in Hinsicht Echtzeit und Latenz noch nicht ausgereift sind.

# Abstract

Digital Process Twins (DPTs) are connected software systems in the industry 4.0 context, which digitally represent, monitor and control production processes. This technology promises productivity gains and can improve the flexibility of supply chains. On the other hand, DPTs are setting difficult-to-achieve requirements for the executing computing resource, such as high and variable performance, reliability and fault tolerance. These requirements might not be satisfyingly met by a monolithic DPT architecture. Monolithic DPTs are also more difficult to maintain because components can not be easily replaced. Few scientific works have yet proposed explicitly non-monolithic digital twins.

A modular DPT promises a longer lifespan through improved maintainability and the ability to replace components. The modules are detached from the physical hardware using container software virtualization. This way the modules can be quickly and reliably migrated between hosts. This portability enables the horizontal scaling of the modules and the moving of modules to the cloud. Horizontally scaled modules also gain fault tolerance to a degree. The exchangeability and reusability of the modules enable a producer-independent module ecosystem.

In this thesis, the components of a DPT are modularized and containerized. An architecture for this containerized, modularized DPT (CMDPT) is created, which provides the required infrastructure for its operation. This concept is realized in a prototypical implementation which demonstrates the characteristics of the CMDPT. The validation of the concept shows new areas of applications for the DPT, but also notes that underlying technologies might not yet have the required real-time and latency capabilities.

# Abkürzungsverzeichnis

**CMDPT** Containerized Modular Digital Process Twin

**DPT** Digital Process Twin

**DT** Digital Twin

**HMI** Human-Machine Interface

**IF-DTiM** Implementation Framework of Digital Twins for Intelligent Manufacturing

**MMI** Machine-Machine Interface

**MOM** Message-Oriented Middleware

**OCI** Open Container Initiative

**SoA** Service-orientierte Architektur

**VM** Virtuelle Maschine

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Inhaltsverzeichnis

# 1. Einführung

Eine neue Welle von Veränderungen erfasst zurzeit die Industrie. Diese wird angeführt vom Konzept Industrie 4.0, das die Integration aller Assets eines Unternehmens in einen virtuellen Raum vorsieht. Dieser Raum ist für alle Ebenen in der Wertschöpfungskette, vom Produkt über die Maschinen bis zum Management, zugänglich. Dadurch lässt sich ein produzierendes Unternehmen komplett neu strukturieren, Prozesse können betriebsweit vollautomatisiert, flexibilisiert und mittels maschineller Intelligenz optimiert werden. „Die Deutsche Industrie […] sieht großen Nutzen und riesige Chancen durch Industrie 4.0“ [Abe+15, S. 3]. Good-Practice-Beispiele zeigen, dass auch einfache Lösungen schon besonders bei kleinen und mittleren Unternehmen die Effizienz in der Produktion enorm steigern können [Abe+15, S. 3].

Die Wandlung zu Industrie 4.0 erfordert die digitale Repräsentation physischer Assets, und damit für jedes Asset ein virtuelles Echtzeitabbild, den Digital Twin (DT). Das virtuelle Echtzeitabbild, verteilte Intelligenz und die schnelle Vernetzung und Konfiguration der Komponenten sind drei von sieben Basismerkmalen der Umsetzung von Industrie 4.0, die die Firma Bosch in einer Studie ausfindig machte [Han17, S. 131]. Das in dieser Arbeit vorgeschlagene Konzept betrifft alle drei dieser Merkmale.

Dafür wird sich auf die Industrie-4.0-Technologie Digital Process Twin (DPT) fokussiert. Der DPT findet Verwendung als Echtzeitabbild und Manager eines Produktionsprozesses, und ist daher besonders relevant für die Industrie. Das Konzept des DPT ist besonders vielversprechend für die Verbesserung von Effizienz, Zuverlässigkeit und Sicherheit in der Produktion. Der DPT bedarf jedoch stetiger Weiterentwicklung, um in der Praxis die hohe Hürde zu überschreiten, bei der Betriebe dessen Einsatz beschließen.

## 1.1. Motivation

Digitale (Prozess-)Zwillinge können in ausgereifter Form sehr anspruchsvolle und komplizierte Technologien sein, an die höchste Anforderungen gestellt werden. Ein Problem bei einer monolithischen Architektur des DPT ist, dass der DPT-Entwickler in allen Aspekten und für den gesamten Lebenszyklus diesen Anforderungen gerecht werden muss. Es ist sehr schwer, Teile eines monolithisch programmierten DPT separat zu entwickeln oder wiederzuverwerten, da dessen Teile voneinander abhängig und stark integriert sind. Die Standardisierung einer facettenreichen und sich wandelnden Technologie wie dem DPT stellt sich zudem als sehr schwierig heraus. Hersteller oder Forscher tendieren dazu, ihre eigenen, miteinander inkompatiblen Architekturen zu entwickeln. Dies fördert zwar die Innovation in der frühen Phase, bremst aber eine Weiterentwicklung aus.

Auch ist die oft enge Bindung der Software eines DPT an die Hardware ein Problem. Kommt es zu Problemen mit der Hardware, ist der DPT direkt betroffen und kann nicht flexibel auf andere Hardware migriert werden. Die Hardware-Last kann zudem stark schwanken und so zu unnötiger Ressourcenbelegung oder Überlastung führen. Ersteres ist aus wirtschaftlichen Gründen unerwünscht, da die reservierte Rechenleistung anderweitig genutzt werden könnte. Eine Überlastung kann bei kritischen Anwendungen zu erheblichen Verzögerungen und ungewünschtem Verhalten führen.

Die Software-Containertechnologie entkoppelt die Software zu einem gewissen Grad von der physischen Hardware. Das erleichtert die Installation, Migration und den Betrieb von einer großen Anzahl an Software-Paketen. Die Container-Technologie kommt gerade im Distributed Computing heutzutage sehr stark zum Einsatz, um durch Virtualisierung der Software die flexible Skalierbarkeit der reservierten Rechenleistung zu ermöglichen und den Betrieb in der Cloud zu vereinfachen [Sus20] [Wat20]. Deswegen könnte die Container-Technologie ein nützliches Werkzeug sein, um monolithische Architekturen von DPTs aufzubrechen.

## 1.2. Zielsetzung

Das Ziel dieser Arbeit ist die Schaffung eines Konzeptes zur Anwendung des Prinzips der Modularität auf den Digitalen Prozesszwilling unter Verwendung der Containertechnologie. Dafür soll eine System-Architektur erarbeitet werden. Eine Verifikation und Validierung des Konzeptes soll über eine prototypische Implementierung erfolgen.

## 1.3. Methodik und Struktur

Die Ausarbeitung des Konzeptes folgt induktiv sowie deduktiv auf der Grundlage vorhandener Forschung. Verschiedene in der Literatur größtenteils separat betrachtete Technologien wie der Digitale Zwilling, die Container-Technologie und die Softwaremodularisierung sollen hierbei auf ihr Zusammenspiel untersucht werden, um mögliche Synergieeffekte zu nutzen. Deren Erkenntnisse werden induktiv im Konzept verarbeitet. Der Teil „Stand der Technik“ (**Kapitel 2**) erläutert diese Technologien, darunter den DPT, die Software-Virtualisierung, sowie die Modularisierung von Software mit deren Anwendungsfällen und einhergehenden Vor- und Nachteilen.

Die Arbeiten, die diese Brücke zwischen DT und Container-Technologie schon teilweise geschlagen haben ([Ala+20], [Bor+17], [Hun+22]) werden ebenfalls untersucht. In keiner dieser Arbeiten wird eine Containerisierung DT-interner Module vorgeschlagen, wie sie in der vorliegenden Arbeit am DPT vorgenommen wird. Es fehlt in den genannten Veröffentlichungen zudem eine ausführliche Charakterisierung des modularen Aufbaus von DT-Systemen. Der Beitrag der Arbeiten zum Konzept wird deswegen deduktiv abgeleitet. Die Arbeiten können außerdem wertvolle Hinweise für die Praxis liefern.

Das Konzept (**Kapitel 3**) wird von einer Auswahl und Entwicklung geeigneter Anforderungen an den Containerized Modular Digital Process Twin (CMDPT) angeführt. Es folgt eine Erläuterung der Charakteristika, welche die Synergie der untersuchten Technologien auszeichnet. Dabei wird auch auf auftretende Herausforderungen eingegangen, die teilweise schon in anderen Forschungsarbeiten qualitativ und quantitativ ermittelt wurden. Die Charakteristika stellen das grundlegende Prinzip des Konzeptes dar. Anschließend wird eine konkrete Architektur für den CMDPT entwickelt. Dabei werden Auswahlen an geeigneten Implementationen der verwendeten Technologien getroffen.

Als Proof of Concept wird in **Kapitel 4** ein Teil des CMDPT in einer ersten skizzenhaften Ausführung auf das Projekt AI-in-Orbit-Factory angewandt und die dabei ermittelten Vorteile und auftretenden Schwierigkeiten dokumentiert. Diese prototypische Implementierung dient dann in **Kapitel 5** als Grundlage für die Verifikation der Anforderungen an das Konzept. Dort wird das Konzept anschließend auch anhand realistischer Anwendungsfälle validiert.

Im abschließenden **Kapitel 6** wird ein Ausblick auf die Weiterentwicklung des DPT vor dem Hintergrund dieses Konzeptes gegeben und die Gebiete mit Forschungsbedarf ermittelt.

# 2. Stand der Technik

Das Konzept des CMDPT und dessen prototypische Implementierung vereinen eine Vielzahl an Technologien. Daher wird in diesem Kapitel diesbezüglich eine Übersicht über den Stand der Technik gegeben. Behandelt werden insbesondere DT und DPT, die Container-Technologie und Software-Modularität. Das Kapitel schließt mit einer Betrachtung bestehender wissenschaftlicher Arbeiten an der Schnittstelle von DT und Container-Technologie.

## 2.1. Der Digitale Zwilling

Der Digital Twin (DT) erweitert ein physisches Asset um ein möglichst identisches virtuelles Objekt. Das Prinzip wurde von Michael Grieves im Jahr 2002 vorgeschlagen [GV17, S. 93]. Ein DT zeichnet sich vor allem durch einen bidirektionalen Datenfluss zwischen dem virtuellen Modell (Digitaler Zwilling) und dem reellen Gegenstand (physischer Zwilling) aus [GV17, S. 93], wie zu erkennen in Abbildung 2.1. Diese Kopplung geschieht durch „Sensoren, Datenbanken, Smart Devices und Prozessmanagement-Systeme“ (aus dem engl. nach [DTB22, S. 3]). Dadurch soll der Zustand und das Verhalten des physischen Zwillings während des gesamten Lebenszyklus simuliert und vorhergesagt werden [NE22, S. 42]. Die Daten, die das Abbild des physischen Zwillings formen, werden im *Digitalen Schatten* gespeichert, einer Teilmenge des DT [Sta+20, S. 47] Die Berechnungen des DT dienen wiederum der Einflussnahme auf den physischen Zwilling.

Der Einsatz eines DT hat in allen Phasen des Lebenszyklus vielseitige Vorteile für den physischen Zwilling und dient als Grundlage für ein weites Gebiet an Funktionalitäten [NE22, S. 43]. Das „Urmodell“ [Sta+20, S. 48] eines DT, der *Digitale Master*, kann schon am Anfang des Lebenszyklus im iterativen Entwicklungsprozess eingesetzt werden, um das Bauen von teuren Prototypen zumindest teilweise zu ersetzen [GV17, S. 100, 112] [Sta+20, S. 48]. Dabei entstehen sogenannte *Digitale Prototypen* [Sta+20, S. 48].

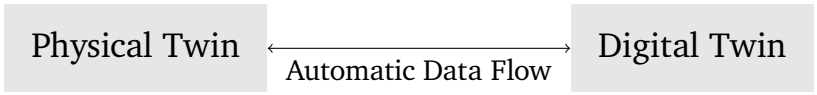


Abbildung 2.1.: Allgemeines Prinzip des digitalen Zwillings, angelehnt an [Kri+18, S. 1017]

In dieser frühen Phase erlaubt der DT das effiziente Aufdecken von unerwünschtem und unerwartetem Systemverhalten. Auch während des Betriebs können unerwartete Situationen mit der Hilfe des DT vorhergesagt und bewältigt werden. Dadurch kann die Zuverlässigkeit des Systems erheblich verbessert werden [GV17, S. 103].

Es gibt auch Herausforderungen bei der Einführung von DTs, wie Grieves [GV17, S. 108, 109] sie selbst anführt:

1. Der DT braucht für dessen volle Funktionstüchtigkeit Modelle der Verhaltensweisen des physischen Zwillings in allen verschiedenen technischen Disziplinen. Maschinenbauer, Elektrotechniker, Informatiker, Physiker etc. müssen von vornherein ein einheitliches virtuelles Modell entwickeln und ihre Informationen über ein Produkt zusammenführen. Die häufige Fachbarriere zwischen ihnen stellt ein Hindernis dar.
2. Der Umfang und die Tiefe der möglichen Simulationen ist aufgrund des mangelnden Verständnisses der realen Welt und der Rechenleistung und Datenübertragungsgeschwindigkeit aktueller Technik limitiert. Diskrepanzen zwischen den Zwillingen sind daher unvermeidbar.

2019 erweitert Grieves die Liste an Herausforderungen [Gri19, S. 29, 30]:

3. Die Cyber-Sicherheit ist ein ernstzunehmendes Thema, besonders beim Einsatz des DT während des Betriebs. Digitale Zwillinge müssen sich selbst vor Cyberangriffen schützen können.
4. Die Datenflut, die durch die Anforderung an präzise Abbildung des physischen Systems entsteht, muss zu nutzbaren Informationen zusammengefasst werden.

5. Die vielen verschiedenen Implementationen des DT von unterschiedlichen Herstellern sollen miteinander kompatibel sein. Es braucht daher eine Standardisierung.

Die akademische Forschung über Digitale Zwillinge nahm in den letzten fünf Jahren stark zu. Untersucht wurde dabei unter anderem der Einsatz in der Produktion, Lagerhausmanagement, Asset Management in der Schifffahrt und im Katastrophenschutz [DTB22, S. 3, 4]. Dennoch ist die Entwicklung in einer noch frühen Phase: Eine 2018 durchgeführte Studie zum DT kategorisiert 55 Prozent der Literatur dazu als „Konzept“ [Kri+18, S. 1018].

In der umfassenden Vision sollen Digitale Zwillinge auf den verschiedenen Ebenen des Produktionsprozesses kooperieren. Diese Multi-DT Systeme sind jedoch noch kaum erforscht und die Realisierbarkeit unklar [BB22, S. 11].

## 2.2. Der Digitale Prozesszwilling

Der DPT ist eine angepasste Version des DT auf Prozessebene und übernimmt zusätzlich zu bekannten DT-Funktionen umfängliche Aufgaben im Produktionsprozess. In dieser Arbeit wird die Definition des DPT von Kempf et al. [Kem+21, S. 7] übernommen: „Der Digitale Prozesszwilling orchestriert die Produktion einer Produktinstanz durch das Zuordnen der relevanten Produktkomponenten, das Instanziieren des Digitalen Produktzwillings als Auftakt des physischen Produktionsprozesses und das Bereitstellen von Informationen für die Digitalen Zwillinge der Produktionsmaschinen“ [Zitat aus dem Englischen].

Der DPT divergiert von der Definition des allgemeinen DT vor allem durch die Abwesenheit eines konkret greifbaren physischen Zwillings. Der Produktionsprozess nimmt dessen Rolle ein. Ein DPT braucht daher auch keine eigenen Aktoren oder Sensoren. Stattdessen steht der DPT als Akteur und Manager mehrerer DTs, vor allem von Produktions- und Produktzwillingen, im Vordergrund.

Es wird von den Autoren [Kem+21, S. 7] insbesondere betont, dass der DPT keine klassische Automatisierungshierarchie realisieren soll. Stattdessen stellt der DPT eine autonome Einheit dar, die selbstständig unter Bezugnahme der Informationen „von oben“ (Auftrag) und „von unten“ (Produktionsmittel) Entscheidungen trifft.

Abbildung 2.2 zeigt die von Kempf et al. [Kem+21, S. 8] vorgeschlagene Architektur eines DPT mit vier Ebenen. Die oberste Ebene ist die Mensch-Maschine-Schnittstelle. Darunter folgt die Daten-Ebene, in der eine vollständige Selbstbeschreibung des DPT,

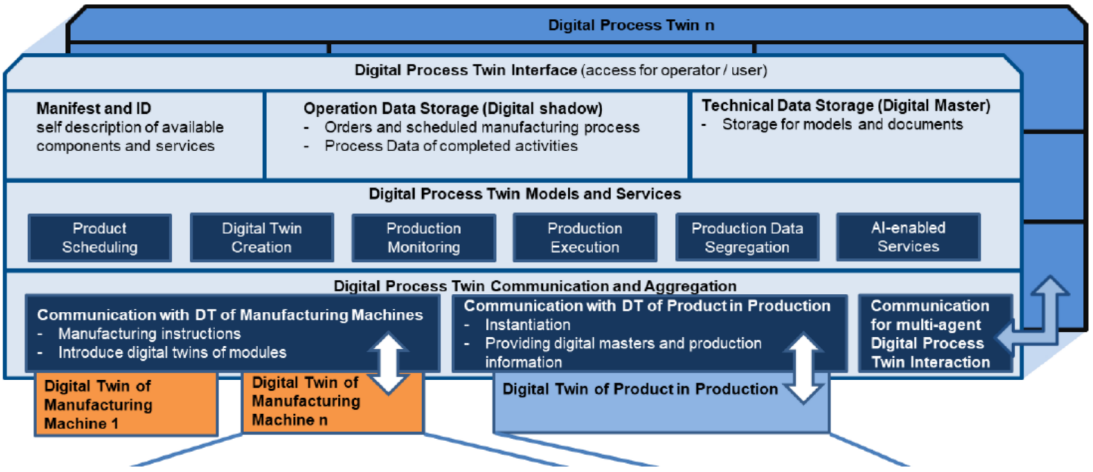


Abbildung 2.2.: Der Digitale Prozesszwilling (DPT). Quelle: [Kem+21, S. 8]

sowie jeweils ein Datenspeicher für den Prozessbetrieb (Operation Data Storage) und technische Modelle und Dokumente (Technical Data Storage) enthalten sind. Die dritte Ebene enthält sämtliche nicht anderweitig kategorisierten Funktionalitäten („Models and Services“) des DPT, wie die Produktionsplanung und -überwachung, die Erstellung der Produkt-DTs und weiteres. Diese Ebene ist beliebig erweiterbar. Die unterste Ebene ist die Schnittstelle zu den relevanten Digitalen Zwillingen des Prozesses, hauptsächlich den DTs der Produktionsmaschinen, der Produkte und anderen DPTs (Maschine-Maschine Schnittstelle).

## 2.3. Software-Virtualisierung mit Containern

In diesem Abschnitt wird zuerst die Software-Virtualisierung anhand der etablierten Technik der Virtuellen Maschinen erläutert. VMs sind nicht zwingend relevant für das vorgeschlagene Konzept, aber wichtig für das Verständnis der Container-Technologie. Zuerst wird deren Funktionsweise vorgestellt, danach folgt ein Überblick über Features von modernen Container-Engines wie Union-Mount, Repositories, Volumes und Networking.

### 2.3.1. Virtuelle Maschinen

Die Software-Containertechnologie existiert vor allem im Kontrast zu der Virtualisierung von Software über Hypervisoren. Virtualisierung im Allgemeinen strebt die Emanzipation der Software von der Computer-Hardware und dem Betriebssystem an. Dafür wird die Software auf einem virtuellen Computer betrieben, der mithilfe eines Hypervisors auf einem physischen Rechner (Host) laufen kann. Der Hypervisor teilt die Ressourcen des Hostsystems auf ggf. mehrere dieser virtuellen Maschinen auf [Red20].

Die Virtuelle Maschinen betreiben ein komplettes Gast-Betriebssystem auf dem Host-Computer, sodass ein signifikanter Overhead entsteht. Dieser bindet Systemressourcen, führt zu einem sehr langsamen Startprozess [Pah15, S. 25], und schränkt dadurch die Mobilität der VM zwischen mehreren Hosts ein.

Die Gast-Betriebssysteme bieten eine vollwertige Rechenumgebung für die zu betreibenden Applikationen. Zusammen gebündelt wird die gesamte Software einer VM als Image gespeichert. Aufgrund des großen Overhead jeder einzeln virtualisierten Software, lohnt es sich nicht, Microservice-Architekturen (siehe Kap. 2.8) aufzubauen. Modularität lässt sich durch diese Technologie daher schwer erreichen.

Die Vorteile von Virtualisierung sind bessere Ressourceneffizienz, erhöhte Sicherheit durch Isolation der VMs und schnelle Einsatzbereitschaft der Software auf beliebigen Hosts [ES21, S. 180]. Der Einsatz von VMs findet sich vor allem in der Infrastructure-as-a-Service (IaaS) Cloud, da mit VMs den Nutzern die volle Kontrolle über die geliehene Hardware gegeben werden kann.

### 2.3.2. Funktionsweise der Container-Technologie und Unterschiede zu VMs

Die Container-Technologie ist eine neuere Virtualisierungstechnik und eliminiert den Gebrauch des Gast-Betriebssystems. Stattdessen teilen Container sich den Systemkernel des Hosts, siehe Abb. 2.3. Die Aufgaben des Hypervisors werden in diesem Fall vom Kernel selbst übernommen: Die Isolierung des Dateisystems und der Prozesse des Containers vom Host wird über Linux-Namespaces implementiert, die Zuteilung und Verwaltung der Host-Ressourcen mit cgroups (Linux Control Groups) [Pah15, S. 25].

Ein Container-Image enthält lediglich die Anwendung mit ihren Abhängigkeiten. Dadurch werden die Größe der Images und die rein zum Betrieb der virtualisierten Software benötigte Rechenleistung reduziert, welches besonders beim Betreiben einer großen Zahl an virtuellen Umgebungen relevant ist. Die Anzahl an separaten Applikationen in einem

Container wird Granularität genannt [Pah15, S. 27], bei Microservice-Architekturen (siehe Kap. 2.8) beträgt diese Eins.

Felter et al. [Fel+15, S. 172] bemaßen in ihren Untersuchungen die Performance von Docker, einer Container-Engine (siehe Kap. 2.6), gegenüber KVM, einer VM Infrastruktur, als immer besser oder gleich. Außerdem seien Virtuelle Maschinen schlechter für den Einsatz bei hohen Datenübertragungsraten oder Anwendungen mit der Anforderung nach niedrigen Latenzen geeignet. Gerade diese Umstände sind aber relevant im Fall einer Verwendung im Industrie-4.0-Kontext.

Da die Isolation der Kernel der verschiedenen virtuellen Umgebungen von der Host-Maschine nicht mehr gegeben ist, wird die Systemsicherheit bei Containern jedoch schwächer eingeschätzt als bei VMs [KS17, S. 176]. Das muss bei kritischen Applikationen beachtet werden.

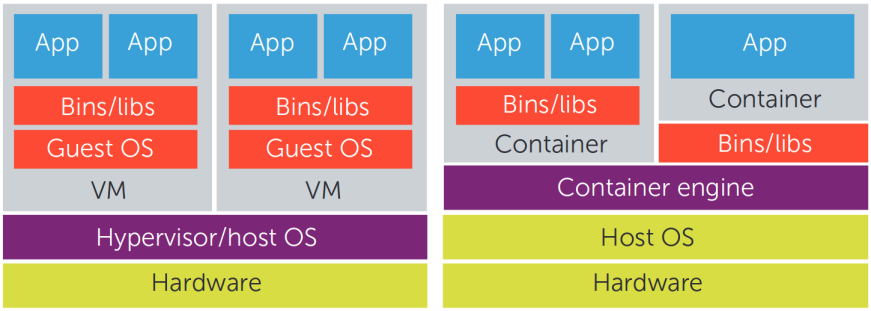


Abbildung 2.3.: Software-Virtualisierungsarchitekturen, links VM, rechts Container. © [2015] IEEE. Quelle: [Pah15, S. 25]

### 2.3.3. Container Engines, Runtimes, Managers und Orchestators

Der Betrieb von Containern wird auf verschiedenen Abstraktionsebenen von unterschiedlichen Programmen ermöglicht. Die *Container Runtime* ist das Programm, das auf niedrigster Ebene für den Aufbau und Betrieb der Container zuständig ist [Doc17]. Der *Container Manager* instruiert die Container Runtime. Der Container Manager ist ein „Satz an APIs, um auf einfache Art und Weise den gesamten Lebenszyklus eines Containers zu verwalten“ [Emi19, S. 224] [aus dem Englischen]. Ein *Container Orchestrator* wiederum wird benötigt, um große Mengen an Containern auf einer großen Anzahl an Hosts zu

verwalten. Der Orchestrator ist unabdingbar für den Betrieb von verteilten Applikationen auf einem Rechen-Cluster (siehe Kap. 2.4). Jede unabhängige Recheneinheit im Cluster benötigt zusätzlich zum Orchestrator noch eine eigene Container-Runtime. *Container Engine* ist ein loser Begriff, der meist eine größere Sammlung an Programmen für den Betrieb von Containern in verschiedenen Einsatzmöglichkeiten bezeichnet. Oft fällt der Begriff mit dem Begriff des Container-Managers zusammen, kann aber auch wie im Fall der Docker-Engine zusätzlich einen Orchestrator beinhalten [Doc22f].

Der de facto Industriestandard-Orchestrator ist Kubernetes [Pah15, S. 30] [Hun+22, S. 3] und der de facto Standard-Container-Manager Docker [Pah15, S. 30]. Docker verwendet die Container-Runtime containerd, die dadurch ebenfalls zum Industriestandard wurde. Kubernetes und containerd sind die in ihrem Feld von der Cloud Native Computing Foundation (CNCF) am meisten empfohlenen Programme, da diese lang erprobt und erfolgreich sind [CNC22]. Die CNCF ist Teil der Linux Foundation.

### 2.3.4. Images und Union-Mount

Eine Besonderheit der Containerarchitektur bei modernen Container-Runtimes wie containerd ist der Aufbau von Images in Schichten. Dafür kommt ein Union-Mount zum Einsatz, welcher mehrere separate Dateisysteme als ein einziges lesbares Dateisystem zugänglich macht. Eine oder mehrere schreibgeschützte Image-Schichten bilden die Basis, auf der die schreibfähige Container-Schicht mit der Applikation sitzt, die wiederum als Image-Schicht eines anderen Containers dienen kann [Pah15, S. 26], siehe Abbildung 2.4.

Ein beispielhafter Aufbau (Abbildung 2.4) eines Container-Stacks wäre ein Ubuntu Base-Image (1. Schicht), auf dem der Emacs-Texteditor (2. Schicht) und ein Apache Webserver (3. Schicht) aufgesetzt wird. Darauf sitzt die oberste Schicht, die schreibbar ist und die Applikation enthält. Das erstellte Container-Image dieser Applikation könnte wiederum genutzt werden, um eine Schicht hinzuzufügen, welche die Applikation nutzt und erweitert. Dieser neue Container enthält dann selbstverständlich auch alle Schichten des alten Containers. Dieses System hat den Vorteil der Wiederverwendbarkeit der Image-Schichten.

### 2.3.5. Volumes und Networking

Da Container keinen beständigen Datenspeicher haben, weil Laufzeitdaten nach der Zerstörung eines Containers mit verloren gehen, existieren unabhängig der Container sog. Daten-Volumes. Das sind von der Container-Engine verwaltete Speicherelemente. Volumes

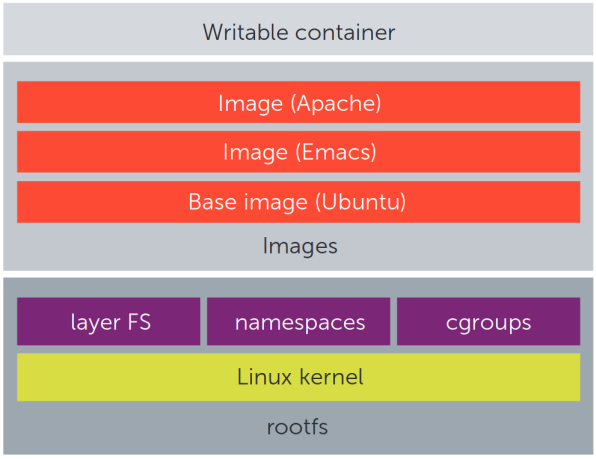


Abbildung 2.4.: Container-Image Architektur anhand des Beispiels eines Apache Webservers. © [2015] IEEE. Quelle: [Pah15, S. 26]

sind nicht Teil des Union-Mount Dateisystems, können von mehreren Containern benutzt werden und unabhängig von denen existieren [Pah15, S. 27]. Die Isolation des Dateisystems kann aber auch bewusst durch die Verwendung von Bind-Mounts durchbrochen werden [Doc22i]. Dabei wird ein Teil des Host-Speichers in den Speicher des Containers projiziert. Das ist vor allem während der Entwicklung praktisch, negiert aber viele Vorteile der Container-Technologie, wie die Unabhängigkeit von der ausführenden Ressource, die Portabilität und die Sicherheit eines isolierten Dateisystems.

Um Container zu vernetzen, können entweder deren Ports an die des Hosts geknüpft (mapping) oder Container untereinander vernetzt werden (linking) [Pah15, S. 27]. Letzteres wird meist über die von Container-Managern erstellten Netzwerke realisiert, an denen die Container teilhaben können.

### 2.3.6. Open Container Initiative

Die Open Container Initiative (OCI) ist ein Projekt der Linux Foundation mit dem Ziel der Standardisierung von Container Images und Container Runtimes [Ope20]. Der OCI-Standard hat sich aufgrund der Mitarbeit und Konformität der gängigsten Container-Engine Docker etablieren können.

## 2.4. Container-Orchestrierung im Container-Cluster

Eine Applikation, die containerisiert in einer Microservice-Architektur (siehe Kap. 2.8) vorliegt, kann durch Verwendung eines Orchestrators auf ganzen Host-Pools betrieben werden. Dafür bedarf es jedoch eines sehr hohen Grades an Organisation und Kooperation zwischen den Hosts, im Cluster *Nodes* genannt. Diese wird durch den Orchestrator erreicht. Der Microservice kann skaliert werden, indem der Orchestrator mehrere identische Container erstellt und die eingehenden Anfragen über Load-Balancing aufteilt. Die resultierende Computing-Plattform, gezeigt in Abb. 2.5, wird *Cluster* genannt. Die einzelnen Programmeinheiten im Cluster werden *Services* genannt.

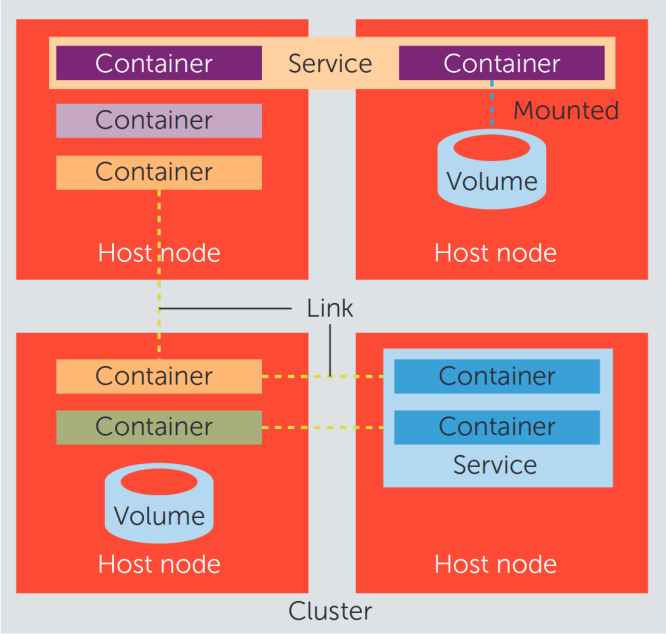


Abbildung 2.5.: Container-Cluster Architektur, Orchestrator nicht gezeigt. © [2015] IEEE. Quelle: [Pah15, S. 27]

Ein Orchestrator soll u. a. folgende Fähigkeiten haben [Kha17, S. 44]:

- Planung und Zustandsmanagement des Clusters,
- Sicherstellen von Ausfalltoleranz im Cluster,

- Sicherstellen der Sicherheit des Clusters,
- Vereinfachung der Vernetzung innerhalb des Clusters,
- Überwachung des Cluster-Zustands.

Der verbreitetste aktive Open-Source Orchestrator ist Kubernetes (siehe 2.3.3). Als Alternative dazu dienen vor allem HashiCorp Nomad, Apache Mesos zusammen mit Marathon, und Docker Swarm. Docker Swarm ist in der Docker-Engine mitenthalten.

## 2.5. Echtzeit-Container

*Echtzeit* beinhaltet keine Aussage über die Schnelligkeit von Berechnungen und Übertragungen, sondern deren zeitlichen Determinismus [Str+20, S. 3]. Stuhár et al. [Str+20, S. 3] unterscheiden zwischen Systemen, die bei Verfehlen einer Deadline lediglich Fehlverhalten aufweisen („soft real-time“) und denen, die ausfallen („firm real-time“) oder schwere Konsequenzen mit sich ziehen („hard real-time“).

Hinze untersucht in [Hin+19] und [Hin+18] den Einsatz von containerisierter Echtzeit-Software in Industriesteuerungen. Ziel ist die Regelung eines Roboters mittels Simulation und Machine Learning [Hin+19, S. 65, 66]. Das Ziel dringt damit in Forschungsbereiche des DT ein. Diese Regelung muss zeitlich determinierte Feedback-Loops aufweisen und benötigt daher Echtzeitfähigkeit. Das ist eine besondere Herausforderung, wenn die Software in einzelne Teile zerlegt wird, die miteinander kommunizieren müssen.

Der *PREEMPT-RT* Patch für den Linux-Kernel ermöglicht Echtzeitanwendungen unter Linux [TMV18, S. 3]. Da Container sich den Host-Kernel teilen, erben sie diese Fähigkeit. Der Betrieb von Simulationen in Containern mit diesem Patch ermöglicht bei genauer Kenntnis des Rechensystems eine bestimmbare obere Grenze für Latenzen [Hin+18, S. 6]. Genauere Untersuchungen dazu fehlen jedoch noch größtenteils.

Dieses Fehlen von weitergehender Forschung zu Echtzeit-Container kritisieren Stuhár et al. [Str+20, S. 7, 8]. Es gebe zudem keine Werkzeuge für den Einsatz von Echtzeit-Containern, keine Forschung zu Echtzeit-Container-Kommunikation und keine Sicherheitsanalysen. Demnach ließe sich ein verlässlicher Einsatz von Echtzeit-Containern noch nicht garantieren.

## 2.6. Docker

Docker ist eine Open-Source Container-Engine und der de facto Industriestandard für Container-Manager [Pah15, S. 30]. Docker ist ein Werkzeug, mit dem der gesamte Lebenszyklus eines Containers verwaltet werden kann. Die Engine baut auf der eigenen Container-Runtime containerd auf und enthält zusätzlich einen Container-Orchestrator, Docker Swarm (siehe 2.3.3). Docker ist verfügbar für Linux, Windows und Mac OS. Auf die grundlegende Funktionsweise von Containern wird in 2.3 eingegangen. Für ausführliche und aktuelle Erläuterungen zur Engine wird auf die Dokumentation von Docker verwiesen.

### 2.6.1. Daemon und Client

Auf dem Container-Host läuft der Docker Daemon. Dieser ist ein Server-Prozess mit REST-API, der Anweisungen vom Docker-Client, dem User-Interface, annimmt. Der Daemon nutzt containerd für die Low-Level Arbeit mit den OCI-Images. [Doc22a]

### 2.6.2. Erstellen von Images mittels Dockerfile

Die Anweisungen zum Erstellen von Images werden in der sogenannten Dockerfile festgehalten. Jedes Image erhält eine separate Dockerfile.

Grundsätzlich ist der Ausgangspunkt eines jeden Images ein Base-Image, das zuerst angegeben wird. Im Fall eines Python-Programms würde daher in der Dockerfile zuerst das Python-Image spezifiziert. Danach wird das Programm in die schreibbare Ebene des Containers kopiert, die nötigen Bibliotheken z. B. mittels des Python Package Managers (pip) installiert und der Ausgangsbefehl für den Container bei Containerstart angegeben. Dieser ist in diesem Fall dann der Python-Befehl mit der Angabe des Pfades des Programmes. [Doc22c]

Die Dockerfile wird von Docker gelesen und das entsprechende Image gebaut und gespeichert. Jede Anweisung der Dockerfile erstellt eine neue Image-Schicht. Bei Bedarf kann das Image dann auf eine externe Registry geladen werden.

### 2.6.3. Arbeiten mit Containern

Der Docker Client beinhaltet CLI-Befehle (Command Line Interface) für alle Aspekte des Container-Managements, darunter sind das Starten und Stoppen von Containern, Networking, Logging, Port-Forwarding, Monitoring und weitere. Der Nutzer ist aber nicht zwingend auf den Client angewiesen, da auch andere Programme auf die API des Docker Daemon zugreifen können. So kann beispielsweise das Docker SDK für Python zur Entwicklung eines solchen Programmes verwendet werden.

### 2.6.4. Docker-Compose

Das Tool Docker-Compose ermöglicht einen benutzerfreundlicheren Umgang mit Applikationen, die aus mehreren Containern bestehen, als das manuelle Verwalten mit einzelnen Client-Befehlen. In einer YAML Datei wird die Zusammensetzung der Applikation, Vernetzung der Container, das Port-Forwarding, zugehörige Volumes und weiteres beschrieben. Die Datei trägt standardmäßig den Namen „compose.yaml“. Durch die dortige Spezifizierung kann das Deployment von Multi-Container-Applikationen automatisiert werden.

Anschließend dient diese compose.yaml dem Docker-Compose Werkzeug dazu, den beschrieben Zustand der Multi-Container Applikation durch Anweisung des Docker Daemons zu erreichen. Zudem bietet das Command Line Interface von Docker-Compose viele Funktionen, die den gesamten Lebenszyklus dieser Multi-Container Applikation abdecken.

### 2.6.5. Docker Swarm

Docker Swarm ist der in der Docker Engine enthaltene Container-Orchestrator. Dadurch wird das Deployment von Multi-Container Applikationen auf Computer-Clustern ermöglicht (siehe Kap. 2.4). Auf allen Teilen eines Swarm-Clusters muss der Docker Daemon aktiv sein.

Eine Instanz der Docker Engine im Cluster wird *Node* genannt. Nodes können die Rolle eines *Managers* oder eines *Workers* einnehmen, aber auch beide gleichzeitig. Die Manager arbeiten kontinuierlichen an der Erreichung des vom Nutzer gewünschten Cluster-Zustands. Gibt es mehrere Manager, einigen sie sich untereinander mit einem Konsens-Algorithmus über den Zustand des Clusters. Die Worker bekommen Arbeit von den Managern zugeteilt und führen diese aus. [Doc22f]

Der Nutzer definiert Applikationen, die er auf dem Cluster betreiben will, als *Services*. Zu einem Service gehört ein Container-Image sowie Definitionen zu der Vernetzung mit anderen Services, geöffnete Ports, CPU- und Speicherbeschränkungen, Anzahl an der Replika-Tasks etc. [Doc22d]

Arbeit wird in Form von *Tasks* verteilt. Ein Task ist eine Anweisung zur Ausführung eines Containers, die der Orchestrator auf Grundlage des gewünschten Cluster-Zustands erstellt. Tasks können immer Services zugeordnet werden. Ein Service wird skaliert, indem zusätzliche Tasks generiert und verteilt werden. [Doc22d]

## 2.7. Software-Modularität

Die Eigenschaften eines Software-Systems werden zum großen Teil von dessen Architektur bestimmt. Verschiedene Komponenten eines Systems zu modularisieren geht mit Vor- und Nachteilen einher, die sich meistens analog auch bei der Gestaltung von physischen Systemen finden lassen.

Nach Balzert [Bal09, S. 41] ist ein Modul eine funktionelle Einheit begrenzten Umfangs, die in sich abgeschlossen und unabhängig vom Kontext entwickelt und getestet werden kann. Die Schnittstellen des Moduls nach außen sollen klar definiert sein.

### Vorteile

Ein großer Vorteil der Nutzung von Modulen ist deren Wiederverwendbarkeit. Hochwertige Module, die sich in der Praxis schon bewiesen haben, können auch in einem neuen Projekt Verwendung finden. Dadurch kann dessen Qualität unter großen Einsparungen von Zeit und Ressourcen gesichert werden [VFN20, S. 3]. Modulare Systemteile lassen sich leichter ersetzen und erweitern, besser warten und überprüfen und können in großen Teams zu einer produktiveren Zusammenarbeit führen [Bal09, S. 42]. Die Austauschbarkeit fördert außerdem die Standardisierung der Komponenten, da verschiedene Ausführungen eines Moduls mit dem Gesamtsystem kompatibel sein müssen, um gleichermaßen einsetzbar zu sein.

Modularisieren bedeutet vor allem abstrahieren [Bal09, S. 42]. Ein modularer Ansatz fordert vom Entwickler ein abstrakteres Denken, welches zu einer klareren Struktur der

Software führen kann. Bei monolithischer (aus einem Stück bestehender) Software hingegen kann eine fehlende Trennung der Funktionalitäten zu erschwerter Verständlichkeit führen.

#### Nachteile

Falls eine Abstrahierung aber gar nicht nötig oder hilfreich ist, kann eine vor allem zu fein granulierte Modularität jedoch unnötige Komplexität und Rechenaufwand bedeuten. Deshalb kann bei kleinen, experimentelleren Systemen, die nicht längere Zeit verwendet werden sollen, eine monolithische Softwarearchitektur den Entwicklungsaufwand senken und überflüssige Rechenkosten vermeiden.

Modularisierung wirft außerdem neue Herausforderungen auf. Besonders in der Domäne der verteilten Systeme führt sie zu der Notwendigkeit von schnellem, Durchsatz-starkem und zuverlässigem Datentransfer innerhalb eines Systems, welcher nicht immer gegeben ist oder garantiert werden kann.

#### Granularität

Wie in Abschnitt 2.3 schon angeschnitten ist die Granularität der modularen Architektur einer der definierenden Kennwerte. Kleinere Module verstärken die Vorteile der Modularität, wie die Wiederverwertbarkeit, aber auch deren Nachteile, wie die Belastung der Datenleitungen [VFN20, S. 11].

## 2.8. Service-orientierte Architektur

Die Service-orientierte Architektur (SoA) ist eine Art und Weise, die Funktionalitäten eines Programmes zugänglich zu machen. „SoA ermöglichen es, vormals monolithische, komplexe Softwaresysteme zu modularisieren.“ [C M+06, S. 12] Ein Service in der SoA zeichnet sich durch folgende Eigenschaften aus [C M+06, S. 12, 13]:

1. Der Zugang zu den Funktionalitäten des Services erfolgt über festgelegte Schnittstellen.
2. Der Service-Provider ist agnostisch gegenüber der Identität des Service-Konsumenten.

3. Der Service-Konsument kennt nicht die konkrete Implementierung der Services.

### 2.8.1. Microservices

Der Begriff *Microservice* bezeichnet den Baustein einer Architektur, in der die atomaren Funktionen einer Applikation in einzelne, unabhängige und zustandslose Services aufgeteilt werden. Die Microservice-Architektur kann daher als ein spezifischer Einsatz des Prinzips der Software-Modularität (siehe Kap. 2.7) betrachtet werden. Die Transformation hin zu einer Microservice-Architektur geschieht vor allem mit der Absicht, die Applikation zu skalieren, indem die Instanzen einzelner Microservices vervielfältigt werden [Nis+22]. Da diese klein sind, ist die feine Einstellbarkeit der Skalierung möglich. Einzelne Funktionalitäten können so abhängig von der Verteilung der Belastung innerhalb einer Applikation skaliert werden [Nis+22].

## 2.9. Message-oriented Middleware

Message-Oriented Middleware (MOM) ist Software zur Kommunikation in verteilten Systemen. MOM wird zwischen heterogene Teilnehmer eines verteilten Systems geschaltet, um einheitlich und zuverlässig Nachrichten austauschen zu können [Yon+19, S. 88]. Die Fähigkeiten von MOM umfassen die Komposition einer Nachricht sowie deren Handhabung über den gesamten Lebenszyklus. Sie umfasst zudem meist Sicherheitsmechanismen, Transaktionsgarantien, Monitoring und abhängig von der konkreten Software weitere Funktionalitäten [Yon+19, S. 89].

Der große Vorteil von MOM ist die lose Kopplung der Teilnehmer. Ein Client bringt eine Nachricht an die MOM an, die sich um den gesamten Transport kümmert [Cur05, S. 5]. Oft wird ein *Broker* eingesetzt, ein separates Programm, das zwischen Absender und Empfänger verortet ist und die Nachrichten weiterleitet. Dadurch muss jede Seite nur den Broker, nicht den Absender oder Empfänger einer Nachricht kennen. MOM stellt unabhängig vom Einsatz eines Brokers eine neutrale Ebene zwischen Absender und Empfänger dar, sodass diese sich nicht strikt aufeinander abstimmen müssen [Cur05, S. 5].

Zwei der Zustellungsmodi von MOMs sind Point-to-Point und Publish-Subscribe. Bei Point-to-Point werden Nachrichten von einem Absender zu einem explizit genannten Empfänger gesandt. Publish-Subscribe zeichnet sich dagegen dadurch aus, dass vom Absender

(Produzent) eine Nachricht mit einem bestimmten *Topic* produziert und veröffentlicht wird (publish). Mehrere Konsumenten können dieses Topic abonnieren (subscribe) und empfangen dann entsprechende Nachrichten. [Yon+19, S. 90]

## 2.10. ZeroMQ

ZeroMQ ist ein Messaging-Framework, das keine Broker benötigt (brokerless architecture) und spezifisch für verteilte Systeme entwickelt ist [Zer22a]. Durch die Vermeidung eines Brokers können Ressourcen eingespart, Bottlenecks/Single Point of Failures verhindert und der Nachrichtenweg verkürzt werden. Dadurch wird ein hoher Durchsatz und niedrige Latenz erzielt [Yon+19, S. 95], jedoch Flexibilität eingebüßt, da Kommunikationspartner sich gegenseitig kennen müssen und nicht nur einen zentralen Broker.

ZeroMQ führt in Untersuchungen von Sommer et al. [Som+18, S. 1221] sowohl in Latenz als auch Durchsatz gegenüber Apache Kafka, AMQP und MQTT. Da es jedoch in seinem Umfang minimal ist, fällt es bezüglich zusätzlicher Funktionalität hinter den anderen zurück [Som+18, S. 1222].

Auch wenn ZeroMQ aufgrund des fehlenden Brokers streng genommen nicht als vollwertige MOM gezählt werden kann [Som+18, S. 1220], erfüllt es dieselben Aufgaben. Die Nutzung der API von ZeroMQ erfolgt unabhängig von der Transportart (TCP, inter-process, Websocket etc.). Diese wird während der Laufzeit des Programmes ausgewählt [Zer22a].

Tasci et al. sprechen ZeroMQ aufgrund niedriger Latenzzeiten eine prinzipielle Eignung für Echtzeitsysteme zu [TMV18, S. 5]. Die Entwickler von ZeroMQ konnten zeigen, dass bei Verwendung auf einem Linux-Betriebssystem mit Real-Time Kernel (siehe Kap. 2.5) Latenzspitzen vermieden werden können [Zer22b]. Es konnten im Test Latenzen über 75 Mikrosekunden ausgeschlossen werden. Dieser Determinismus ist unabdingbar für den Betrieb in Echtzeit.

## 2.11. Bestehende Konzepte zur Containerisierung von Digitalen Zwillingen

Wie in 1.3 erwähnt, wurde in einigen wenigen wissenschaftlichen Arbeiten schon die Containerisierung von allgemeinen DT-Strukturen behandelt.

### 2.11.1. Der Digitale Zwilling im Cloud- und Fog-Computing

Borodulin et al. [Bor+17] schlagen die Migration des DT in die Cloud vor. Dies sei eine Lösung für die Anforderung an Performance und Flexibilität des DT. Ein detailliertes Konzept stellen sie in ihrem Paper nicht vor.

Alternativ zum Betrieb des DT in der Cloud steht der Einsatz in einer Fog-Computing-Umgebung. Fog-Computing ist ein Konzept, welches zwischen den Smart-Devices (Sensoren, Maschinen, genannt *Edge*) und der *Cloud* mehrere Ebenen mit sogenannten *Fog*-Nodes ansiedelt [Ior+18]. Diese sind geografisch verteilte aber zusammenarbeitende Recheneinheiten, die näher an den Smart-Devices sind als die Cloud-Server. Fog-Nodes verarbeiten von Smart-Devices kommende Daten und senden nur nicht-Latenz-kritische Daten weiter an die Cloud. Das verringert die Latenz und erhöht den möglichen Durchsatz. [Ior+18, S. 2]

Alaasam et al. [Ala+20] schlagen den Betrieb eines DT in solch einer Fog-Computing-Umgebung vor. Ihr Augenmerk liegt auf der Verarbeitung von Event-Streams, also kontinuierlichen Strömen an Daten von z. B. Sensoren, durch den DT in der Fog-Umgebung. Verarbeitet werden die Datenströme mit Apache Kafka, einer speziell dafür zugeschnittene Software. Untersucht wird dabei vor allem das Verhalten während der Migration des DT zwischen Hosts in der Fog-Umgebung. Diese Migration ist eine wichtige Fähigkeit, um Ausfallsicherheit zu gewährleisten. Das Paper beschreibt vor allem die Schwierigkeit, die der Betrieb von Zustands-behafteten Services in der Fog-Umgebung bereitet.

### 2.11.2. IF-DTiM

Die Entwickler des Implementation Framework of Digital Twins for Intelligent Manufacturing (IF-DTiM) schlagen die Containerisierung von DT-Applikationen mittels Docker und den Betrieb im Cluster mit Kubernetes vor [Hun+22]. Das Konzept geht auch darüber hinaus auf Aspekte der Künstlichen Intelligenz, Virtuellen Realität und weitere im Rahmen dieses Konzeptes ein. Es wurde gezeigt, dass durch die Containerisierung eine größere Anzahl verschiedener „Intelligent Services“ und Cloud-Services im Plug-and-Play Stil eingebaut werden können.

Die in sich abgeschlossenen Images, das schnelle Deployment, Fehlertoleranz, Autoscaling und Load Balancing sind weitere genannte Vorteile [Hun+22, S. 15].

Die Kommunikation läuft ähnlich wie bei [Ala+20] (siehe 2.11.1) über die Event-Streaming-Plattform Apache Kafka. Im Best Effort konnten mit dem Kafka-Cluster zuungunsten der

Fehlertoleranz Latenzen von 1,13ms erreicht werden [Hun+22, S. 13], sie bewegt sich damit gerade noch im von Hinze hinnehmbaren Bereich von Latenzen in Steuerungssystemen in der Industrie (siehe Kap. 2.5).

Anstatt jedoch den DT selbst zu Modularisieren, werden die intelligenten Applikationen aus dem DT herausgetrennt und containerisiert [Hun+22, S. 4]. Diese rechenaufwändigen Services laufen in einem Container-Cluster in der Cloud, während die DTs in einem Container-Cluster an der Edge, also in geografischer Nähe der Maschinen betrieben werden (siehe 2.11.1). Das Konzept schlägt nicht explizit die Containerisierung einzelner interner Teile des DT vor, es wird jedoch angemerkt, dass der Betrieb von DT-Applikationen in Microservice-Architektur möglich sei [Hun+22, S. 7].

# 3. Konzept zur Modularisierung des Digitalen Prozesszwillings

Ausgehend vom Stand der Technik aus Kapitel 2 werden in diesem Kapitel Anforderungen an das Konzept eines Containerized Modular Digital Process Twin (CMDPT) entwickelt und dessen Charakteristika erläutert. Die Charakteristika sind wichtig für die Begründung der CMDPT-Architektur, die in zwei unterschiedlichen, aber aufeinander aufbauenden Konzeptionierungen ausgearbeitet wird. Die Konzeptionierung des CMDPT im Cluster (Kap. 3.4) birgt einen signifikanten Mehrwert zu der auf einem einzelnen Host (Kap. 3.3), ist jedoch mit einer erhöhten Komplexität und größerem Hardware- und Rechenaufwand verbunden. Entsprechend unterscheiden sich die Anwendungsfälle.

## 3.1. Anforderungen

Zu Beginn der Forschungsarbeit stand das Ziel einer modularen Architektur des DPT und damit verbundener Funktionsweisen. Aus diesem Ziel, den Definitionen von DPT (Kap. 2.2) und Modul (Kap. 2.7), sowie dem nach weiterführenden Überlegungen entwickelten gewünschten Nutzen des Konzeptes ergeben sich folgende Anforderungen. Diese dienen als Grundlage für die zielgerichtete Entwicklung des Konzeptes des CMDPT in Kapitel 3.3.

Folgende Anforderungen ergeben sich aus der zu dieser Arbeit führenden Zielsetzung eines modularen DPT mit bestimmter Funktionsweise:

1. Modulare Architektur des DPT,
2. Fähigkeit der flexiblen Verknüpfung der Module,
3. Unabhängigkeit der Module von der ausführenden Ressource,
4. Auslagerungsfähigkeit von Modulen auf externe Ressourcen.

Aus der Definition des DPT (Kap. 2.2) ergeben sich folgende Anforderungen:

5. Existenz von jeweils eigenen Modulen für (siehe Abb. 2.2 in Kap. 2.2):
    a) Digital Process Twin Interface, hier Human-Machine Interface (HMI) genannt,
    b) Data Storage, bestehend aus Manifest and ID, Operational Data Storage und Technical Data Storage,
    c) Digital Process Twin Models and Services, hier Functionalities genannt,
    d) Digital Process Twin Communication and Aggregation, hier Machine-Machine Interface (MMI) genannt.
6. Alle nicht unter 5. aufgezählten möglichen Erweiterungen des DPT müssen sich unkompliziert Modularisieren lassen.

Aus der Definition eines Moduls (Kap. 2.7) ergeben sich folgende Anforderungen:

7. Kontextunabhängigkeit der Module,
8. Eigenständige Existenz der Module,
9. Existenz einheitlicher Schnittstellen der Module,
10. Kommunikationsfähigkeit der Module untereinander und nach außen.

Um einen signifikanten Mehrwert zu bieten, werden vom Autor selbst folgende Anforderungen gestellt (*in kursiv folgt die Begründung*):

11. Es gibt keine nicht-containerisierten Module des DPT, *um vollständige Portabilität zu gewährleisten.*
12. Die Module sollen als Microservice ausgelegt und daher nur lose gekoppelt sein, *um die Skalierbarkeit in einer SoA zu gewährleisten (siehe Kap. 2.8).*
13. Die Module des DPT sind bezüglich ihrer zugewiesenen Ressourcen über mehrere Hosts hinweg skalierbar, *eine Erweiterung der Anforderung 4, um stets die nötigen Ressourcen bereitzustellen, aber auch keine Rechenressourcen zu verschwenden.*
14. Es gibt einen zentralen Speicherort für alle Container-Images *für einen praktischen Entwicklungsprozess und unkompliziertes Deployment.*

## 3.2. Charakteristika

Im folgenden Abschnitt sollen die Prinzipien der Containerisierung und Modularisierung des DPT erläutert werden. Zusammen ergibt dies die gewünschte charakteristische Beschreibung eines Containerized Modular Digital Process Twin (CMDPT).

### 3.2.1. Modularität des DPT

Die Modularisierung eines DPT führt zu einer signifikanten Änderung von dessen Systemeigenschaften. Diese werden im Folgenden beschrieben, noch unbeachtet der Containerisierung der Module. Daraus folgen die Gründe für die Modularisierung des DPT, bzw. auch allgemeiner des DT.

#### Reduktion der Kompliziertheit

Digitale Zwillinge, vor allem im ausgereiften Stadium, sind komplizierte Systeme. Das Verhalten von komplizierten Systemen ist berechenbar, doch die Anzahl der Komponenten ist groß und die Funktionsweise ist nur für Experten wirklich zugänglich [SM11, S. 68]. Eine Modularisierung des DT schafft klare Abstraktionsebenen (siehe Kap. 2.7), die allen Anwendern beim Verständnis des Systems helfen können. Ein Physiker, der die Simulationen für den DT entwickelt, muss sich so nur mit der standardisierten Schnittstelle des DT für Simulationsdaten auseinandersetzen, und kann den Rest vernachlässigen.

#### Arbeitsteilung

Eine vollständige Arbeitsteilung für die verschiedenen Teile des DPT kann nur mit Modulen stattfinden. Firma X und Firma Y könnten durch den Einsatz von Modulen unabhängig voneinander Module A und B für den DPT entwickeln, ohne voneinander oder der spezifischen Realisierung des DPT abhängig zu sein. Dies erlaubt Spezialisierung und ist vorteilhaft für die langfristige Weiterentwicklung des DPT. Die Einbindung von allgemeinen Container-Modulen wie Datenbanken als generische Images in den DPT ist schon jetzt mit geringem Zeitaufwand und Wissen möglich (siehe Implementierung, Kapitel 4). Die Entwickler solcher Images müssen keinerlei Bezug zum DPT haben.

#### Austauschbarkeit

Da Module kontextunabhängig sind und über festgelegte Schnittstellen verfügen (2.7) sind sie leicht austauschbar. Der Lebenszyklus der Module ist nicht mehr von dem des DPT abhängig. So kann ein DPT auf undefinierte Zeit betrieben werden. Nach und nach können einzelne Module durch neuere ersetzt werden.

#### Wiederverwertung, Wartung, Erweiterung

Die einzige Voraussetzung für den Betrieb eines Moduls in einem beliebigen Umfeld ist, dass dessen Schnittstellen kompatibel sind. Gibt es eine Standardisierung, ist dies gegeben.

Dadurch können Module in neuen Projekten wiederverwertet und ohne Probleme erweitert werden. Außerdem wirkt sich die Modularisierung positiv auf die Wartbarkeit des DPT aus. Ein Spezialist bräuchte für die Wartung eines Moduls nur das spezifische Modul zu betrachten und könnte den Kontext vernachlässigen. Zumal kann Wechselwirkung, also ein Fehler eines Moduls aufgrund eines anderen Moduls, ausgeschlossen werden, insofern dieser nicht über eingehende Nachrichten zustande kam. Dies erleichtert die Fehlersuche im System enorm.

#### Standardisierung

Die Standardisierung der Module wäre für Modulentwickler und deren Kunden von großem Interesse. Das fördert die Standardisierung auf diesem neuen Gebiet. Schon der Erfinder des DT, Michael Grieves, sah Standardisierung bzw. Interoperabilität als ein wichtiges Vorhaben [Gri19, S. 30, 31] (siehe auch Kap. 2.1). Das Schichtensystem der modernen Container-Images ermöglicht zudem das Entwickeln von Modul-Basen, also Base-Images, die als Grundlage für eine Standardisierung in der Industrie 4.0 dienen könnten.

### 3.2.2. Modularität durch Container

Wie in 2.3 gezeigt, zeichnen sich Container durch gute Performance und kompakte Images in der Software-Virtualisierung aus. Ein vollwertiges DPT-Modul kann massiv von der

Flexibilität und Portabilität der Virtualisierung und der Einfachheit in Konfiguration und Betrieb der Containertechnologie profitieren. Jedoch treten durch diese Synergie auch neue Herausforderungen auf. Im Folgenden soll genauer auf Gründe für und Probleme mit der Modularisierung des DPT *durch Container* eingegangen werden.

### Kontextunabhängigkeit

Um als vollkommene Module nach Balzert (siehe Kap. 2.7) gelten zu können, müssen die DPT-Module unabhängig vom Kontext funktionieren und in sich abgeschlossen sein (Anforderung 7). Dies soll durch Containerisierung der Module erreicht werden. Ein Grund für die Virtualisierung ist die Unabhängigkeit von der Hardware und Isolation der Software (siehe Kap. 2.3). Der einzige benötigte Kontext für den Betrieb eines Containers ist die Container-Runtime und der Host-Kernel. Da diese in diesem Konzept als über alle Hosts homogener Kontext angesehen werden, kann ein Container-Modul wenn auch in einer schwachen Formulierung als kontextunabhängig gelten. Es soll zudem keine Abhängigkeiten (z. B. Bibliotheken) geben, die das Modul benötigt und die nicht im Container-Image enthalten sind. Das hat den Vorteil, dass die Installation eines DPT-Moduls sehr einfach und an keine Rahmenbedingungen geknüpft ist außer der Existenz der Container-Runtime.

### Portabilität

Ein Programm, das virtualisiert gespeichert ist, benötigt keinen individuellen Installationsprozess. Durch die Isolation des Programmes vom Hostsystem und anderen Containern können außerdem keine Software-Konflikte entstehen. Das Programmverhalten sollte auf jedem Host identisch sein, da alle benötigten Abhängigkeiten mitgeliefert werden. Es kann also beispielsweise nicht sein, dass ein System-Update oder die neue Version einer Bibliothek oder SDK die Software untauglich macht, es sei denn, diese wird bewusst im Image aktualisiert. Die Startzeit eines Containers beträgt zudem üblicherweise nur wenige Sekunden und die Imagegrößen kompakter Programme sind unter 100 MB. Die Images können zügig heruntergeladen und die Container zeitnah gestartet werden. Das alles lässt sich als sehr gute Portabilität zusammenfassen.

Diese Portabilität greift einige Probleme an, die vor allem in der Forschung präsent sind. Boettiger [Boe15, S. 72] fasst die Probleme zusammen, die eine große Herausforderung für

die Reproduzierbarkeit in der Forschung darstellen: Die Schwierigkeit, exakte Software-Bedingungen zu replizieren

1. da Code oft eine unüberschaubare Anzahl an Abhängigkeiten besitzt,
2. wenn der Installationsprozess der Software schlecht dokumentiert ist,
3. die Abhängigkeiten regelmäßige Updates erhalten („code rot“).

Der DPT ist noch größtenteils ein Forschungsthema, die Schwierigkeiten können aber auch auf die Industrie übertragen werden. Die Portabilität von Containern verhindert diese Probleme, wenn Forscher ihre Software als Images bereitstellen. Gleiches gilt analog auch für die Industrie-Praxis.

Zusätzlich könnten Forscher, die eine Arbeit replizieren wollen und nicht die nötige Rechenleistung zur Verfügung haben, Container leicht auf Cloud-Plattformen auslagern [Boe15, S. 75]. Dies ist die geforderte Auslagerungsfähigkeit der Module (Anforderung 4).

Die Portabilität der Container konnte sich im Projekt IF-DTiM (siehe 2.11.2) im DT-Kontext schon bewähren: Die Autoren merken das “standard and effortless software packaging and deployment” [Hun+22, S. 4] positiv an.

Aus der Portabilität lässt sich eine besondere Prädestination der Containertechnologie für die Modularisierung des DPT ableiten. Die Portabilität ermöglicht, dass Module, gleich wie beispielsweise ein Elektromotor oder eine Lagerung, ohne Probleme Hersteller- und Projektübergreifend unkompliziert und schnell eingesetzt und auch wieder ausgebaut werden können.

#### Skalierbarkeit und Hochverfügbarkeit

Eine Literaturstudie aus 2022 nennt die Skalierbarkeit des DT als in der Forschung betrachtete Problematik [BB22, S. 11]. Die Skalierbarkeit von auf Containerbasis-Architektur gebauten Systemen ist gerade zurzeit ein Hauptgrund für deren Einsatz in der Cloud-Computing-Industrie (siehe Kap. 1.1). Die Container-Engine kann einem Container System-Ressourcen dynamisch zuweisen oder entziehen. Damit wird die Ressourcennutzung optimiert und längerfristige Überlastung verhindert.

In Clustern (siehe Kap. 2.4) können Anwendungen auf Rechnernetzen betrieben werden, in denen die Anzahl und Verteilung der Container so gewählt wird, dass der Rechenaufwand aufgeteilt (Skalierbarkeit und Load-Balancing) und der Ausfall von Hosts kompensiert werden kann (Hochverfügbarkeit). Skalierbarkeit, Load-Balancing und Hochverfügbarkeit

der Cluster konnten im Projekt IF-DTiM (siehe 2.11.2) klare Vorteile für DT-Systeme darstellen [Hun+22, S. 15].

Für den Betrieb im Cluster oder in der Cloud ist für DTs die in 3.2.2 besprochene Echtzeitproblematik sowie die erhöhte Latenz kritisch. Dem Latenz-Problem der Cloud kann mit dem alternativen Betrieb des DT in einer Fog-Computing-Umgebung begegnet werden, siehe 2.11.1.

#### Probleme in der Datenübertragung

Auch wenn Container als Virtualisierungstechnik mit geringem Overhead verstanden werden, kann dieser beim Datentransfer nicht vernachlässigt werden. Kratzke [Kra17, S. 3] konnte eine Verschlechterung in der Transferrate bei containerisierten Applikationen von ca. 10 % für Pakete über 100 kB bis zu 20 % für Pakete unter 100 kB nachweisen. Dies ist auf den zusätzlichen Overhead im Networking zurückzuführen.

Des Weiteren ist klar, dass isolierte Module auf die Kommunikation untereinander angewiesen sind. Das führt zu sehr hohen Anforderungen an die Bandbreite, Zuverlässigkeit und Schnelligkeit der Kommunikation untereinander, vor allem bei Echtzeitanwendungen wie dem DT. Grieves listete die Datenflut, die zur genauen Simulation des DT nötig ist, in seiner Aufzählung bestehender Probleme des DT, siehe Kap. 2.1. Bei hochperformanten oder kritischen Anwendungen muss dieser Punkt also ausführlich geprüft werden. Es konnte aber gezeigt werden, dass zumindest die durchschnittlichen Latenzen der containerisierten Software ausreichend sind für viele Industrieanwendungen (Zykluszeiten ungefähr 1ms [MSF16, S. 1839]) [Hin+19, S. 72].

#### Echtzeitproblematik

Aus der Forderung, ein Digitaler Zwilling solle die Realität möglichst genau abbilden, lässt sich ableiten, dass die Datenverbindung zwischen physischem und digitalen Zwilling in Echtzeit stattfinden muss.

Jede Aktion des Zwillingspaares muss also in einer fest garantierten Zeit ausgeführt werden können. Handelsübliche Personal Computer garantieren dies bei keiner ihrer Berechnungen. Es muss im Zweifelsfall der besondere Aufwand betrieben werden, Container in Echtzeit zu betreiben (siehe Kap. 2.5). Dieses Forschungsgebiet ist noch nicht

ganz erschlossen: Struhár et al. [Str+20, S. 7, 8] kritisieren u. a. die fehlenden Werkzeuge für den verlässlichen Einsatz von Echtzeit-Containern, fehlende Erforschung von Echtzeit-Container-Kommunikation, sowie fehlende Sicherheitsanalyse zum Betrieb von Echtzeit-Containern.

Dies stellt noch ein Hindernis für den CMDPT dar. Der Einsatz des CMDPT wird daher vom Autor vorerst für echtzeitkritische Applikationen nicht empfohlen.

## 3.3. Konzeptionierung für nicht-verteilte Systeme

In diesem Abschnitt wird die Architektur des Konzepts entwickelt, das die in Kapitel 3.2 beschriebenen Charakteristika der Modularisierung des DPT durch Containerisierung besitzt, den sogenannten Containerized Modular Digital Process Twin (CMDPT). Die Architektur, dargestellt in Abbildung 3.1, konkretisiert vor allem die notwendige Infrastruktur, die für den Betrieb von Container-Modulen nötig ist. Zunächst wird sich auf den Einsatz der Container auf einem einzelnen Host fokussiert. In Kapitel 3.4 wird diese Architektur auf den Betrieb im Cluster ausgeweitet. Rahmenbedingungen sind dabei weiterhin die in Kapitel 3.1 aufgeführten Anforderungen an das Konzept.

### 3.3.1. Container-Infrastruktur

Die Entwicklung des Konzepts beginnt mit der Bereitstellung der notwendigen Container-Infrastruktur, ohne die der Betrieb des CMDPT nicht möglich wäre. Der DPT läuft nach Anforderung 11 in seiner Gesamtheit containerisiert in der Container-Runtime. Dies ist erforderlich, um die Portabilität des DPT zu gewährleisten und alle Vorteile der Containerisierung zu nutzen. Die Container-Runtime wird in Abb. 3.1 als Teil der Container-Engine gehandelt (siehe 2.3.3).

Die Container-Runtime selbst muss auf dem Host manuell installiert werden. Es empfiehlt sich die Installation einer Container-Engine, die die Container-Runtime und zusätzliche Werkzeuge für den Umgang mit Containern enthält, insbesondere auch einen Container-Manager. Der User kann diese Werkzeuge benutzen, um die Module des CMDPT herunterzuladen und zu starten. Soll die Verwaltung des CMDPT über das Administrative Interface (siehe 3.3.3) erfolgen, wird dieses Modul zuerst manuell gestartet. Danach kann der CMDPT über das Admin-Interface zusammengestellt werden.

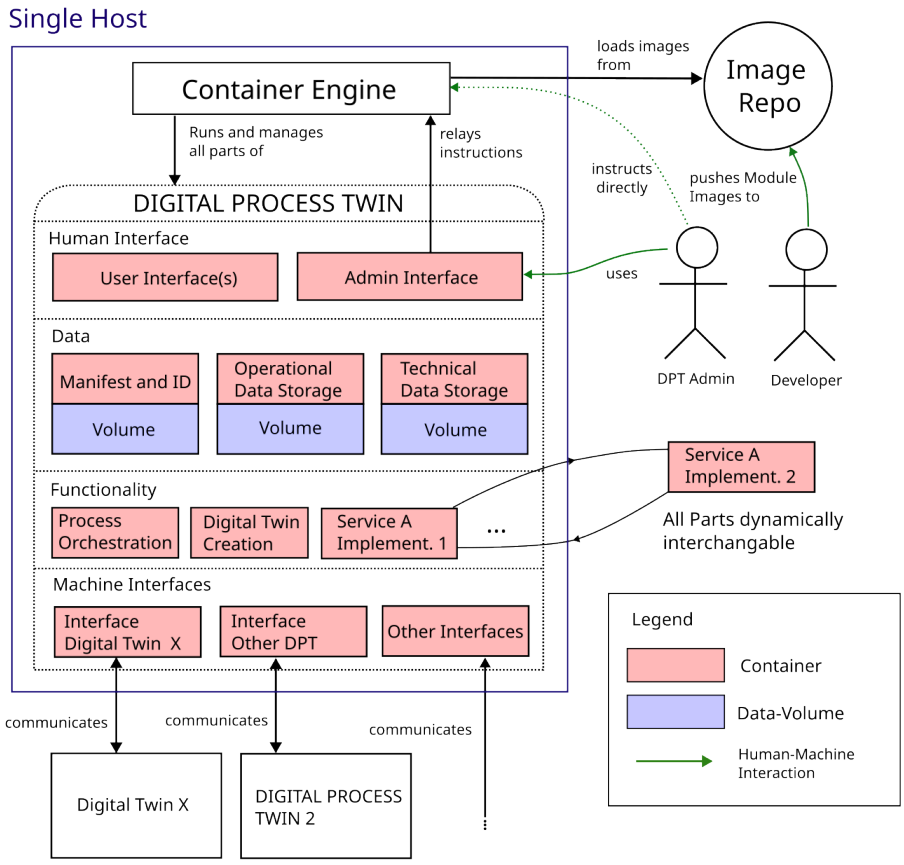


Abbildung 3.1.: Single-Host-Architektur des Digitalen Prozesszwillings mit containerisierten Modulen CMDPT

Die Werkzeuge der Container-Engine sind auch nach Start aller Module sehr hilfreich für den Betrieb des CMDPT. Die Container und damit auch die Module können während der Laufzeit des DPT mit dem Container-Manager dynamisch aktualisiert oder ausgetauscht werden, manuell oder über das Admin-Interface. Neue Module können hinzugefügt und

Verbindungen zwischen den Modulen modifiziert werden.

Für maximale Flexibilität enthält jeder Container genau ein Modul. Wären mehrere Module in einem Container enthalten, wären diese nicht mehr kontextunabhängig und eigenständig ausführbar. Damit wären Anforderungen 7 und 8 verletzt. Außerdem ließen sich die Module nicht unabhängig voneinander austauschen (siehe 3.2.1). Die genaue Granularität der Module muss nach den in 2.7 genannten Abwägungen gewählt werden.

### 3.3.2. Image-Repository

Das Image-Repository ist der Speicherort für die Images aller Module des CMDPT. Damit wird Anforderung 14 erfüllt. Die Container-Engine bezieht ihre Images stets von einer oder mehreren dieser Repositories. Jeder Entwickler von Modulen des CMDPT kann ein eigenes Repository besitzen, auf dem dessen Software-Images abgelegt werden. Viele nicht-DPT-spezifische Module wie SQL-Datenbanken oder Webserver liegen häufig bereits als Images auf öffentlichen Image-Repositories (z. B. Docker-Hub) vor und können ohne Probleme direkt von dort bezogen werden. Der CMDPT ist also keinesfalls auf ein Image-Repository als Quelle der Images beschränkt. Jedoch sollte bestenfalls ein betriebseigenes Image-Repository existieren, das alle Images (als Kopie) hält, um von externen Ausfällen nicht betroffen zu sein.

### 3.3.3. Modul-Ebenen des CMDPT

Die grundlegende Modul-Struktur des CMDPT geht aus dem DPT-Konzept von Kempf et al. [Kem+21] hervor (siehe Anforderung 5). Die Ebenen des CMDPT werden davon ausgehend zum einfacheren Verständnis neu benannt. So entstehen die Kategorien „Human Interface“ (HMI), „Data“, „Functionality“ und „Machine Interface“ (MMI).

In Abbildung 3.1 ist auf der Ebene „Functionality“ die Erweiterbarkeit durch weitere Funktionalitäten als Module angedeutet. Im Realfall werden einige Module hier eingefügt werden müssen, damit der Einsatz des DPT zu nennenswerten Vorteilen für die Produktion führt. Die hier abgebildete Anzahl an Containern soll daher nicht final sein.

### Human-Machine Interface

Auf der Human-Machine Interface Ebene existieren zwei Module für den Zugriff auf den CMDPT durch Menschen. Es existiert das User-Interface, das Zugriff auf das Frontend vom DPT gibt. Das ist eine Schnittstelle, die einem technischen Mitarbeiter die Möglichkeit gibt, den Produktionsprozess selbst zu überwachen und zu steuern. Ein generisches User-Interface gibt es nicht, es muss von Modulen geliefert werden.

Außerdem gibt es das Admin-Interface, das nutzerfreundlichen Zugriff auf die Verwaltung der einzelnen Module und die Container-Engine gibt. Dieses ist generisch und nicht abhängig von einem DPT-Anwendungsfall, wie das User-Interface. Das Admin-Interface muss lediglich Instruktionen an die Container Engine weitergeben und grafische Oberflächen anzeigen.

Im Admin Interface können die Logs gelesen, Module gestartet, gestoppt oder ausgetauscht werden. Das Management des CMDPT kann nach Instanziierung des Admin-Interface-Containers darüber erfolgen. Alternativ kann der Admin bei Kenntnis der Container-Engine auch direkt mit dem passenden Command Line Interface arbeiten. Das Admin-Interface ist also kein zwingend notwendiges Modul.

### Data-Ebene

Die Data-Ebene enthält exklusiv den Zustand des DPT (siehe 3.3.4). „Manifest and ID“ enthält eine Selbstbeschreibung des CMDPT. Dazu gehört eine Identifikationsnummer und eine Beschreibung des Aufbaus des CMDPT. „Operational Data Storage“ enthält alle Daten, die für den Betrieb des CMDPT relevant sind. Dazu gehören unter anderem Simulationsdaten, Prozessdaten und Produktdaten. „Technical Data Storage“ enthält Modelle und Dokumente zu dem Produktionsprozess und den Modulen.

### Functionality-Ebene

Auf der Functionality-Ebene existieren alle Services, die der DPT anbietet und die nicht zu einer der anderen Ebenen gehören. Dazu gehören z. B. Simulationen, die Orchestrierung des Produktionsprozesses, das Erstellen von Product-DTs etc. Zu Beachten ist, dass die Services auf dieser Ebene keinen Zustand halten dürfen, sondern dafür auf das Operational Data Storage zugreifen.

#### Machine-Machine Interface

Die Machine-Machine-Interface-Ebene ist zuständig für jegliche Kommunikation nach außen zu anderen Maschinen bzw. DTs. Angedacht sind an dieser Stelle vor allem die DTs der Produktionsmaschinen sowie andere DPTs.

### 3.3.4. Volumes und Zustand

Dass alle Module des CMDPT containerisiert sind, hat zur Folge, dass die Module an sich zustandslos sind. Container haben keinen beständigen Speicher (siehe Kap. 2.3).

Der Zustand des CMDPT besteht aus dessen eigener Struktur, die durch die Modularisierung dynamisch ist, und den in den Volumes gespeicherten Daten. Ersteres wird von der Container-Engine selbst, sowie als Selbstbeschreibung im „Manifest and ID“ gespeichert.

Volumes sollen ausschließlich in der Daten-Ebene des CMDPT existieren, um eine klare Trennung von Daten und Funktionalität/Logik sicherzustellen. Davon profitiert vor allem die Skalierbarkeit des CMDPT im Container-Cluster (siehe Kap. 3.4), da eine Microservice-Architektur begünstigt wird.

### 3.3.5. Modul-Basis als Base-Image

Um Kompatibilität zwischen den verschiedenen Modulen zu vereinfachen, existiert eine Modul-Basis, die vor allem Funktionen zur Kommunikation zwischen den Modulen bereitstellt. Die Kommunikation wird dadurch nicht mehr Aufgabe der Modul-Entwickler sein. Das fördert die Einheitlichkeit der Module.

Jedes Modul-Image baut auf diesem Base-Image auf und erbt damit dessen Funktionalitäten (siehe Kap. 2.3).

### 3.3.6. Service-orientierte Architektur

Die Art der Interaktion der Module miteinander wird über die Service-orientierte Architektur bestimmt (siehe Kap. 2.8). Jedes Modul agiert als Server (Service-Provider), als Client (Service-Konsument) oder beides. Ein Server empfängt Anfragen (requests) und verarbeitet und beantwortet sie unabhängig vom Anfrager. Diese Service-Architektur

ermöglicht es, „vormals monolithische, komplexe Softwaresysteme zu modularisieren“ [BV20, S. 12].

### 3.3.7. Kommunikation

Da die Container-Prozesse voneinander isoliert sind, müssen sie über Netzwerkschnittstellen miteinander kommunizieren.

Die Container-Engine erstellt und verwaltet ein CMDPT-internes Netzwerk, dem alle Container des CMDPT angehören. Das öffnet die Transportwege für die Kommunikation, ohne Kommunikation nach außen zuzulassen, die nicht speziell eingerichtet ist.

Damit die Austauschbarkeit der Module funktionieren kann, muss jedes Modul über klare Kommunikationsprotokolle verfügen. Dies ist ebenso eine Anforderung der SoA (siehe Kap. 2.8). Es wird eine MOM in der Modul-Basis eingesetzt, die den größten Teil des Nachrichtensystems bereitstellt. Es muss aber weiterhin Absender und Empfänger bekannt sein, welche Art von Nachrichten das jeweils andere Modul empfängt und sendet. Das wird in einem Nachrichtenprotokoll spezifiziert.

Es wird ein Beispiel für solch ein CMDPT-Nachrichtenprotokoll angeführt: Es steht fest, dass Modul A eine Anfrage an Modul B stellen kann, die die Nachricht „GET_COORDINATES“ enthält. Modul B antwortet darauf mit einer Bestätigung und den Koordinaten. Jedes Modul von Typ A oder B von einem anderen Hersteller kann problemlos eingebaut werden, wenn es dieses Protokoll implementiert.

Die Adressen der Server/Service-Provider können bei den Clients/Service-Konsumenten über Umgebungsvariablen im Container festgelegt werden, eine Funktionalität, die von den Container-Engines geliefert wird. Damit lassen sich die Module einfach und schnell neu verknüpfen (Anforderung 2). Die Adressen der Container liegen entweder in Form von IP-Adressen oder als Domain-Namen bei Vorhandenseins eines DNS-Servers in der Container-Engine vor. Letzteres ist aufgrund der größeren Flexibilität deutlich vorzuziehen und ist als verbreitetes Werkzeug zumindest bei Docker und Kubernetes vorhanden [Doc22h] [Kub22].

### 3.3.8. Auswahl einer geeigneten Messaging-Technologie

In [Yon+19] wird eine Auswahl moderner MOMs verglichen. Darunter sind MQTT, Apache Kafka, RabbitMQ, ActiveMQ, RocketMQ und ZeroMQ.

Da das Konzept des DPT trotz der Erweiterbarkeit mit beliebigen Modulen feste Kommunikationswege zwischen den Modulen vorsieht, eignet sich eine Publisher-Subscriber Architektur (siehe Kap. 2.9) prinzipiell weniger. Pub-Sub eignet sich eher für lose Verknüpfungen und Systeme, bei denen mehrere Teilnehmer Nachrichten aus dem gleichen Pool abgreifen. Publish-Subscribe ist hauptsächlich für unidirektionale Kommunikation mit vielen Konsumenten konzipiert, die DPT-Module sollten aber bidirektional kommunizieren können [Mic22]. Es steht im DPT zudem ein Produzent eher wenigen Konsumenten gegenüber, die unterschiedliche Informationen benötigen.

Stattdessen soll ein Point-to-Point Zustellsystem verwendet werden. Point-to-Point wird jedoch weder von Kafka noch MQTT unterstützt [Yon+19, S. 94].

Yonggue et al. empfehlen ZeroMQ für Anwendungen, bei denen Performance Priorität hat [Yon+19]. Im Experiment durch Sommer et al. [Som+18, S. 1221] führt ZeroMQ sowohl in Latenz als auch Durchsatz gegenüber Apache Kafka, AMQP (implementiert von RabbitMQ [Yon+19, S. 95]) und MQTT. Da Performance ein kritischer Faktor in DT-Anwendungen ist (siehe Kap. 2.1), wird der Empfehlung nachgekommen. Für den Einsatz sprechen auch die weiteren Eigenschaften von ZeroMQ wie Abwesenheit eines Brokers und Eignung für Echtzeit-Anwendungen, die in 2.10 erläutert sind.

### 3.3.9. Auswahl geeigneter Containertechnologien

Auch wenn das vorgeschlagene Konzept nicht auf eine spezifische Container-Engine aufbaut, wird an dieser Stelle eine Auswahl an geeigneten Technologien diskutiert. Hierbei werden grundsätzlich nur aktiv gepflegte Projekte in Betracht gezogen, um eine Zukunftsperspektive zu schaffen. Gesucht wird ein Container-Manager (zu den Unterschieden zwischen Engine, Manager etc. siehe 2.3.3), der Werkzeuge für den gesamten Container-Lebenszyklus mitbringt. Die Containertechnologie sollte Open-Source sein und den OCI-Standard unterstützen. In diesem Fall kann der Manager mit allen gängigen Images, und auch denen, die im Rahmen dieses Konzeptes entstünden, umgehen. Damit kommen vor allem Docker, Podman (https://podman.io/) und Apptainer (http://apptainer.org/) in Betracht.

Docker ist der de-facto Industriestandard für Container-Manager (siehe 2.3.3). Das IF-DTiM, ein großes Projekt, das Container im DT-Kontext einsetzt (siehe 2.11.2), verwendet Docker. Die Docker-Engine enthält zudem den Container-Orchestrator Docker-Swarm, der aber in einem nicht-verteilten Einsatz keine Verwendung findet. Für eine ausführliche Beschreibung von Docker wird auf 2.6 verwiesen.

Podman ist eine Container-Engine, die das Command Line Interface von Docker übernimmt aber keinen Daemon besitzt. Ein Daemon wie der Docker Daemon ist ein Hintergrundprozess, der notwendig für die Funktionalität der Software ist. Dadurch braucht Podman im Betrieb keine Root-Privilegien [Con21], was sich positiv auf die Sicherheit auswirkt.

Apptainers Use Case ist wissenschaftliches High-Performance Computing [KSB17], während Docker und Podman General Purpose Container-Manager sind.

Dordevic et al. [Dor+22] untersuchten die Performance-Unterschiede zwischen Docker und Podman und fanden kaum Unterschiede. Stattdessen solle die Auswahl der Container-Engine nach Präferenz der Features erfolgen [Dor+22, S. 5]. Emilsson konnte ebenfalls keine signifikanten Unterschiede feststellen [Emi20, S. 24].

#### Empfehlung

Die Vorrangstellung von Docker ist das wohl beste Argument in dieser Auswahl. Dadurch sollte es Unternehmen gut gelingen, Fachkräfte für den Betrieb des containerisierten DPT zu finden. Ist ein Umstieg auf ein verteiltes System eine Möglichkeit, profitiert Docker zudem von dessen integrierten Container-Orchestrator Docker-Swarm (siehe 3.4.4).

## 3.4. Konzeptionierung für verteilte Systeme

In diesem Kapitel wird die Architektur für verteilte Systeme untersucht. Dabei soll der CMDPT in einem Cluster betrieben werden und dort horizontal skalierbar sein (Anforderung 13). Die CMDPT-Architektur für nicht-verteilte Systeme wird dementsprechend erweitert, siehe Abbildung 3.2. Daher wird auf die Aspekte des CMDPT, die schon in der Single-Host-Konzeptionierung erläutert wurden, nicht erneut eingegangen. Stattdessen werden die Unterschiede zum nicht-verteilten System beschrieben.

Das Container-Cluster agiert als eine zusammenhängende Recheneinheit und kumuliert die Rechenleistung der Hosts. Der DPT liegt hier nicht mehr lokal gebunden, sondern delokalisiert und dezentral im Cluster vor. Das Management dieses verteilten Systems soll auch nicht zentral stattfinden, sondern unter Absprache aller beteiligten Recheneinheiten. Dadurch soll ein Single Point of Failure möglichst vermieden werden.

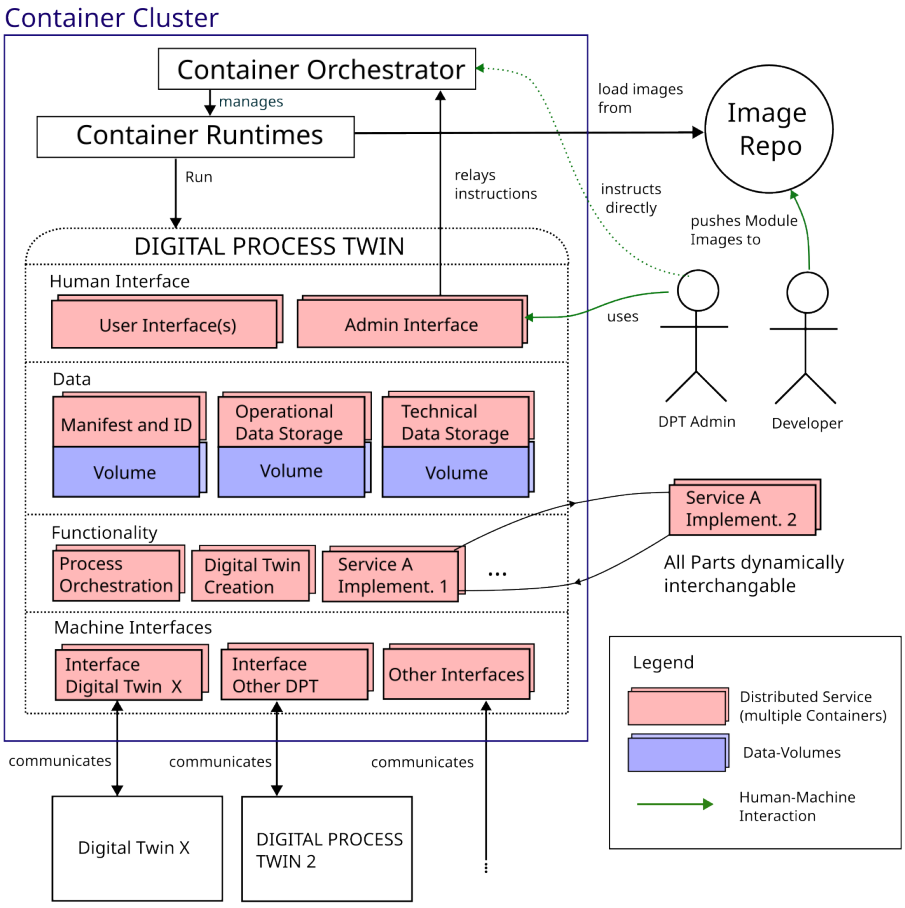


Abbildung 3.2.: Der modularisierte, containerisierte Digitale Prozesszwilling im Cluster aus Sicht des DPT

### 3.4.1. Container-Orchestrator

Im Cluster reicht ein einfacher Container-Manager nicht mehr aus, um das hochdynamische Netzwerk aus Containern und Hosts zu verwalten. Stattdessen kommt ein Orchestra-

tor zum Einsatz. Der Orchestrator unterhält Instanzen von sich auf jedem Host, wie zu erkennen in Abbildung 3.3. Es sind nur exemplarisch Module abgebildet. Diese sind miteinander vernetzt und steuern das Cluster autonom. Der Nutzer legt einen gewünschten Cluster-Zustand fest, den der Orchestrator zu erreichen versucht.

Die Low-Level Container-Arbeit wird weiterhin von Container-Runtimes verrichtet, die auf allen Hosts aktiv sein müssen. Die Runtimes erhalten Befehle von der lokalen Orchestrator-Instanz.

### 3.4.2. Module im Cluster

Der innere Aufbau des DPT ändert sich nicht, sodass Abbildung 3.2 stark an Abbildung 3.1 angelehnt ist. Die gleichen Modul-Images wie im nicht-verteilten Konzept werden vom Image-Repository bezogen.

CMDPT-Module nehmen im Cluster die Rolle von Services ein (siehe Kap. 2.4). Der Container-Orchestrator verwaltet die Services, die innerhalb des Cluster-Netzwerkes miteinander kommunizieren. Zu beachten gilt, dass ein Service/Modul als Gruppierung mehrerer identischer Container dieses Moduls auf verschiedenen Hosts zu betrachten ist. Auf diese Weise wird die horizontale Skalierbarkeit realisiert. Für unproblematisches Verhalten bei der Skalierung müssen die Module aber als Microservice entworfen und zustandslos sein (siehe 2.8.1).

Die Module können auch im Cluster von der Portabilität der Container profitieren und unkompliziert installiert, gestartet, aktualisiert und ausgetauscht werden.

### 3.4.3. Kommunikation

Die Grundsätze der Kommunikation im verteilten System ändern sich nicht im Vergleich zur Single-Host-Kommunikation aus 3.3.7. Da der CMDPT in diesem Modell auf verschiedenen Hosts verteilt ist, müssen die einzelnen Container miteinander über lokale Netzwerke kommunizieren. Dafür tauschen sie untereinander Nachrichten asynchron über TCP aus.

Sobald ein Modul skaliert in Form mehrerer identischer Container vorliegt, ist jedoch nicht eindeutig, zu welchem Container eine Nachricht geleitet werden soll, die an das Modul adressiert ist. Dafür soll ein Load Balancer die eingehenden Nachrichten gleichmäßig auf die einzelnen Container des Moduls verteilten. So werden die gegebenen Ressourcen optimal genutzt. Antworten auf eine Anfrage müssen exakt an den anfragenden Container

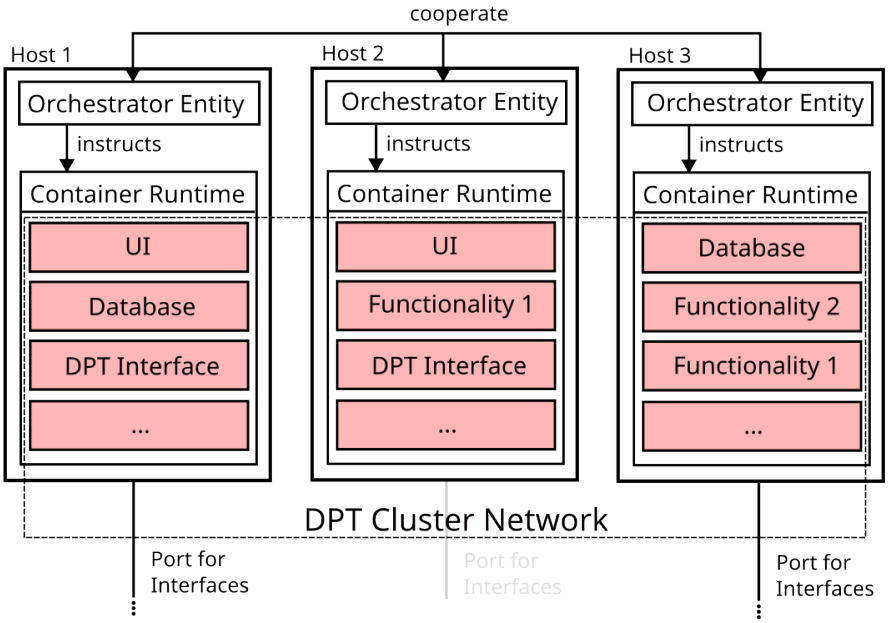


Abbildung 3.3.: Der modularisierte, containerisierte Digitale Prozesszwilling im Cluster aus Sicht der Hosts

zurückgeleitet werden. Dafür wird einer Anfrage stets ein Absender beigefügt, der den anfragenden Container identifiziert.

Die Auswahl der geeigneten Messaging-Technologie wird aus 3.3.8 übernommen, da ZeroMQs Fähigkeiten agnostisch gegenüber den Transportwegen sind.

### 3.4.4. Auswahl geeigneter Container-Orchestratoren

Zum Betrieb des DPT in einem Cluster ist ein Container-Orchestrator nötig (siehe Kap. 2.4). Moderne, verbreitete Lösungen sind Kubernetes, Docker Swarm, Apache Meses (mit Marathon) und HashiCorp Nomad.

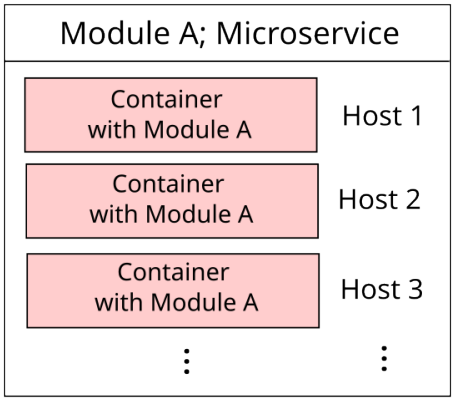


Abbildung 3.4.: Ein Modul im Cluster-DPT

Mercl und Pavlik empfehlen die Nutzung von Docker Swarm für den Einsatz auf kleinerem Maßstab und Kubernetes für großflächigeren Einsatz in komplizierteren Projekten [LJ19, S. 684]. Dies lässt sich auch aus den Analysen von Pan et al. herleiten: Der Einsatz von Docker Swarm ist für den Docker-Nutzer einfach („out of the box“), da Swarm auf natürliche Art und Weise mit dem Rest der Engine integriert [Pan+19, S. 192, 198].

Kubernetes ist dagegen eine größere, komplexere Software mit einer steilen Lernkurve für den Nutzer [Pan+19, S. 192]. Die erweiterte Funktionalität sei für großflächige Einsätze sehr vorteilhaft, die zusätzliche Infrastruktur für kleine Cluster aber teuer [Pan+19, S. 198]. Jawarneh et al. konkludieren: „[...] Kubernetes is one of the most complete orchestrators nowadays on the market. That explains why practitioners gravitate toward preferring it to other ones. At the same time, its complex architecture introduces, in some cases, a significant overhead that may hinder its performances.“ [Jaw+19, S. 6] Die Verbreitung von Kubernetes ist im industriellen Einsatz hilfreich für die Verfügbarkeit von Fachkräften, die solche Systeme überwachen, bedienen und warten können.

Weitere, aber weniger übliche Lösungen sind ein Apache Mesos Cluster mit dem dafür konzipierten Container-Orchestrator Marathon und HashiCorp Nomad. Mesos ist eine Ressource-Management-Ebene, die in Clustern Einsatz findet, um Cluster-Ressourcen fair zwischen verschiedenen Cluster-Computing Frameworks aufzuteilen [Hin+11, S. 13]. Damit können noch weit mehr Aufgaben erfüllt werden als Container-Orchestrierung. Es

hat auch den Vorteil des guten Zugangs zu Apaches Ökosystems.

HashiCorp Nomad ist eine Orchestrierungs-Software, die auch das Management von nicht-containerisierten Applikationen und Virtuellen Maschinen übernehmen kann [Has22]. Nomad wird im Gegensatz zu Kubernetes Komplexität als einzelne Binary entwickelt und kann auf HashiCorps Software-Ökosystems zurückgreifen.

#### Empfehlungen

Die Verbreitung und Reife von Kubernetes spricht für sich als die im Moment für den ernsthaften Einsatz empfohlene Technologie. Dafür spricht auch dessen Einsatz im Projekt IF-DTiM (siehe 2.11.2). Docker Swarm kann als einfache, schnelle und leichte Alternative für experimentelle Einsätze empfohlen werden, besonders, wenn Docker schon im Einsatz ist.

# 4. Prototypische Implementierung

Es folgt die Beschreibung der prototypischen Implementierung des CMDPT für verteilte und nicht-verteilte Systeme, basierend auf deren Konzeptionen aus Kapiteln 3.3 und 3.4.

Zur Untersuchung der Charakteristika aus Kap. 3.2 werden nur exemplarisch Module implementiert. Dafür werden drei Komponenten eines DPT, die im Rahmen des AI-in-Orbit Factory Projekts an der TU Darmstadt entwickelt wurden, auf die Verwendung im CMDPT hin umgeschrieben und containerisiert. Dabei handelt es sich um:

1. Das Operational Data Storage (siehe 3.3.3), bestehend aus
    a) einer MongoDB Datenbank,
    b) einem Interface-Modul, das auf eingehende Anfragen hin die Datenbank modifiziert,
    c) einem Verwaltungs-UI für die Datenbank (mongo-express),
2. dem Interface zur Steuerung des EVA Automata Roboterarms und
3. einer UI für den Prozessablauf des Roboters, bestehend aus
    a) einer Django-Applikation, die über den WSGI-Server Gunicorn Anfragen entgegennimmt und
    b) einem NGINX Webserver.

Bei der Auswahl der Containertechnologie wird der Empfehlung aus 3.3.9 nachgekommen und die Docker Engine gewählt. Um beide Möglichkeiten der Verwaltung der Multi-Container Applikation abzudecken, wird für das nicht-verteilte System Docker-Compose und für den Cluster das Administrative UI (Portainer) verwendet.

Das Kapitel beginnt mit einer Beschreibung der Rechenumgebung, es folgt eine Erläuterung zu der inneren Programmstruktur der Module, den verwendeten Container-Werkzeugen

und dem Entwicklungsprozess für CMDPT-Images. Daran schließt der Betrieb des CMDPT-Prototypen auf einem einzelnen Host und im Cluster an.

## 4.1. Beschreibung der Rechenumgebung

Die Implementierung des verteilten sowie nicht-verteilte CMDPT erfolgt auf einem ASUS UX3410U Notebook mit einer Intel i7-7500U CPU @ 2,7 GHz, 8 GB RAM und einer Arch Linux Distribution, auf der Docker installiert ist.

## 4.2. Programmstruktur der Module

In diesem Abschnitt werden die Programme, die als Module des CMDPT containerisiert werden, untersucht.

### 4.2.1. Programmiersprache

Für die Implementierung wird Python als die primäre Programmiersprache des Projekts AI-in-Orbit Factory übernommen. „Python ist eine interpretierte, objektorientierte, high-Level Programmiersprache mit dynamischer Semantik.“ [Gui22] [aus dem Englischen] Im Rahmen dieses ersten Konzeptes ist dies eine sinnvolle Wahl, da in erster Linie die Charakteristika des CMDPT demonstriert werden sollen, und Rechenleistung für den Demonstrator des Projekts zu dem Zeitpunkt eine untergeordnete Rolle spielt. Die gute Lesbarkeit von Python erleichtert außerdem das Verständnis der Implementierung.

### 4.2.2. Base-Image/Modul-Basis und Kommunikation

Um Einheitlichkeit zu schaffen, wird allen Modulen eine Modul-Basis in Form eines Base-Images zugrunde gelegt (siehe 3.3.5). Das Base-Image basiert selbst auf dem offiziellen Python-Alpine-Image von Docker-Hub (https://hub.docker.com/), dem größten öffentlichen Image-Registry. Alpine Linux ist eine sehr kompakte Linux-Distribution und daher beliebt für den Einsatz in Containern. Ein Alpine-Image ist nicht größer als 8 MB [Alp22] und enthält nur die notwendigsten Programme für den Container-Betrieb.

Die Modul-Basis besteht aus einer Python-Klasse mit dem Namen BaseModule, welche etwaige Module entweder als Objekt instanziieren oder von der die Module als Klasse erben. Die Modul-Basis implementiert in diesem Prototyp ausschließlich die Kommunikation der Module miteinander. Dafür gibt es eine Methode zur Registrierung von Server-Verbindungen von der Client-Seite aus (register_connection), die aufgerufen werden muss, bevor der Client erste Kommunikation anstoßen kann. Dazu kommen separate Methoden für den Empfang und Versand von Nachrichten von Client- und Server-Seite.

Die Kommunikation wird über ZeroMQ DEALER/ROUTER Sockets implementiert. Jedes Modul erhält einen ROUTER-Socket für die allgemeine Server-Rolle und einen DEALER-Socket für jede Client-Verbindung. Die DEALER-Sockets der Clients verbinden sich mit den ROUTER-Sockets der Server. Dadurch kommt es, dass jeder DEALER-Socket nur eine, jeder ROUTER-Socket aber mehrere Verbindungen verwaltet. Dafür ist der ROUTER-Socket ausgelegt: Die einkommenden Verbindungen erhalten Identitäten, über die der Socket die einzelnen Clients separat ansprechen kann.

### 4.2.3. Microservices / Service-orientierte Architektur

Einzelne Funktionalitäten des CMDPT werden als Microservices implementiert (siehe 3.3.6). Exemplarisch soll dafür das Interface des Operational Data Storage angeführt werden. Der Python-Code dieses Interfaces ist in Appendix B gegeben. Der Code wurde im Anhang gekürzt und enthält nur den Teil für die Bearbeitung von Requests bezüglich der Waypoints des EVA Automata Roboters.

Besonders hervorzuheben ist hierbei die Service-Schleife, die während der Laufzeit des Services auf eingehende Nachrichten (Requests) wartet und den jeweiligen Funktionen zuordnet. Dies geschieht über Pythons Structural Pattern Matching. Die Funktionen greifen über PyMongo auf die in einem separat laufenden Container betriebenen MongoDB Datenbank zu. PyMongo ist eine Python Bibliothek für die Modifikation von MongoDB Datenbanken.

Die Nachrichten in dieser Implementierung sind standardisiert und bestehen aus einem Tupel JSON-kompatibler Datentypen. Das erste Element des Tupels gibt den Typ der Nachricht (Request-Typ) an. Die nächsten Elemente enthalten für den Request zu übermittelnde Daten. Das stellt ein primitives Nachrichtenprotokoll dar, wie in 3.3.7 gefordert. Auf diese Weise können alle Module die Struktur eingehender Anfragen verstehen.

## 4.3. Container-Werkzeuge

Für den Umgang mit Images und Containern gibt es sehr hilfreiche Werkzeuge, die bei der Implementation des Prototyps Verwendung finden. Das Erstellen von Images mit einer sogenannten Dockerfile sowie das Verwalten von Multi-Container Applikationen mit Docker-Compose sind Fähigkeiten, die von Docker bereitgestellt werden. Die grafische Nutzeroberfläche Portainer dagegen ist ein unabhängiges Open-Source-Projekt, das hier Anwendung als Administrative User Interface des CMDPT findet.

### 4.3.1. Erstellen von Images mit Dockerfiles

Die Instruktionen für den Aufbau des Images erhält die Container-Engine über die Dockerfile (siehe Kap. 2.6). Die Dockerfile des Interfaces der MongoDB-Datenbank ist in Abbildung 4.1 abgebildet.

Ausgehend vom Base-Image, das von der Registry bezogen wird, wird die Python-Bibliothek PyMongo mit dem Package Installer for Python installiert. Anschließend wird der Programmcode in den Container kopiert und der Modulname als Umgebungsvariable festgelegt. Die letzte Zeile gibt den bei Containerstart auszuführenden Befehl an, in diesem Fall ist es das Starten des Python-Programms.

Diese Datei ist ausreichend, um mit `docker build` das Container-Image zu bauen. Bei großen Projekten bietet sich es jedoch auch an, nicht alle Images einzeln zu bauen, sondern auf der Werkzeug Docker-Compose zurückzugreifen, das den Bau von mehreren Images gleichzeitig automatisieren kann.

EVA > DPT > DATA > op_data > Dockerfile

```
1  ARG REGISTRY
2
3  FROM $REGISTRY/base_module
4  WORKDIR /app
5  RUN ["pip", "install", "pymongo"]
6  COPY op_data.py op_data.py
7  ENV MODULE_NAME op-data-interface
8  CMD ["python", "op_data.py"]
```

Abbildung 4.1.: Die Dockerfile des Operational Database Interface

### 4.3.2. Docker-Compose

Docker-Compose ist ein Werkzeug, mit dem der gewünschte Zustand der Containerkonstellation in einer YAML-Datei festgehalten und von dieser ausgehend umgesetzt werden kann. Außerdem kann damit der Bau von Images für Multi-Container Applikationen automatisiert werden. Die Docker-Compose Datei (compose.yaml) des nicht-verteilten DPT-Prototypen ist in Anhang A gegeben.

Jedes Modul wird unter dem Abschnitt „services“ eingetragen. Ein Modul-Eintrag enthält den Ort der Dockerfile zum Bauen des Images, sowie den Image-Namen inklusive gegebenenfalls der Registry-Adresse. In diesem Fall wird die Registry-Adresse durch die Umgebungsvariable REGISTRY festgelegt.

Dazu kommen Informationen für den Betrieb des Containers. Hierbei handelt es sich um verbundene Volumes und Netzwerke. In diesem Konzept soll nur die Data-Ebene Zustand erhalten (siehe 3.3.3), weshalb nur der Datenbank ein Volume zugewiesen wird. Alle DPT-Module werden in einem Netzwerk „dpt“ miteinander verbunden, abgesehen von dem Webserver, der nur eine Verbindung zum WSGI-Server benötigt. Beiden wird daher ein separates Netzwerk „webserver“ zugewiesen. Für Module, auf die von Außen zugegriffen werden soll, werden Ports freigegeben. Das betrifft vor allem die UI. Um den einzelnen Modulen die Kommunikationspartner zuzuweisen, werden Umgebungsvariablen zugewiesen, jeweils mit dem Service-Namen des Partners.

### 4.3.3. Portainer

Alternativ zu der manuellen Verwaltung des CMDPT über den Docker Client oder mit Docker-Compose wird der Einsatz einer grafischen Nutzeroberfläche für diese Zwecke vorgeschlagen. Dafür eignet sich Portainer (https://www.portainer.io/), auch aufgrund dessen Fähigkeit zum Betrieb im Cluster. Portainer wird vor der Instanziierung des CMDPT über den Docker Client installiert. Anschließend können die einzelnen Module über das Interface intuitiv installiert und konfiguriert werden. Portainer verfügt außerdem über ein User-System, das verschiedenen User-Gruppen verschiedene Befugnisse zuteilen kann.

## 4.4. Implementierung des Entwicklungsprozesses für CMDPT-Images

Bevor der CMDPT zum Einsatz kommen kann, müssen dessen Module entwickelt, containerisiert und abrufbar gespeichert werden. Auf die Entwicklung der Module wird in Kap. 4.2 eingegangen. Im Folgenden wird auf die Installation und Verwendung des Image-Repositorys (Docker Registry) sowie das Erstellen und Abspeichern der Modul-Images eingegangen.

### 4.4.1. Registry

Für das Image-Repository (siehe 3.3.2) kommt Docker Registry zum Einsatz. Docker Registry ist die von Docker entwickelte und empfohlene Applikation zum Speichern und Bereitstellen von Container-Images. Die Registry wird als Container (Single-Host) oder Service (Cluster) installiert. Dafür muss ein Port, z. B. Port 5000, am Container freigegeben werden. Für die Installation mit dem Docker-Client genügt daher der Befehl [Doc22b]:

```
docker run -d -p 5000:5000 --name registry registry:2
```

Images können auf der Registry gespeichert werden, indem sie zuerst nach folgendem Muster benannt werden:

```
[IP]:[Port]/[Name]
```

und dann mit `docker push [Image-Name]` hochgeladen werden. Anschließend können sie von beliebigen Maschinen mit `docker pull [Image-Name]` heruntergeladen werden.

Für diese Implementierung wird eine Registry auf einer virtuellen Maschine mit der IP `192.168.56.104` installiert. Das zeigt die Auslagerungsfähigkeit des Image Repositories. Diese Registry findet Verwendung sowohl in der Single-Host- als auch der Cluster-Implementierung. Die Verwendung einer Registry ist bei einer Single-Host-Implementierung jedoch optional, solange die Images lokal gebaut wurden.

### 4.4.2. Erstellen und Abspeichern der Images

Mithilfe der compose.yaml aus Appendix A werden die Images aus dem Quellcode gebaut. Dafür braucht es zwei Befehle: „export“ legt eine Umgebungsvariable mit der Adresse der Registry fest, „docker compose build“ instruiert Docker-Compose, das Bauen der Images zu veranlassen. Da alle Module auf dem Base-Image aufbauen, wird es explizit zuerst gebaut. Folgende Befehle sind also auszuführen:

```
export REGISTRY=192.168.56.104:5000
docker compose build base-module
docker compose build
```

Dafür wird für jedes Modul eine Dockerfile im Quellcode-Ordner erstellt, die der Container-Engine die Anleitung für den Bau liefert (siehe 4.3.1).

Wird eine Registry auf dem Host oder anderswo betrieben, können die gebauten Images dort aufgespielt werden:

```
docker compose push
```

Möglicherweise muss die IP der Registry noch auf allen Hosts zu den „insecure_registries“ in Dockers daemon.json hinzugefügt werden. Das ist der Fall, wenn die Verbindung zur Registry nicht über ein TLS Zertifikat geschützt ist. Vom ungeschützten Betrieb der Registry ist in der Praxis stark abzuraten. Siehe hierzu [Doc22g].

## 4.5. Betrieb auf einem einzelnen Host

In diesem Abschnitt wird die Konzeption aus Kap. 3.3 im Betrieb prototypisch an den exemplarischen Modulen umgesetzt. Dabei handelt es sich um den Betrieb in virtualisiert modularer Bauweise, bei der die Container der Module auf einem einzelnen Host in der Nähe des Produktionsprozesses betrieben werden.

### 4.5.1. Installation des CMDPT mit Docker-Compose

Im Folgenden wird das Aufsetzen des CMDPT mit Docker-Compose gezeigt, wobei die Installation über das Admin-Interface (Portainer) auch möglich ist. Dieses wird im nächsten Abschnitt (4.6) für die Implementierung im Cluster gezeigt.

### Starten des CMDPT mit Docker-Compose

Über das Terminal wird im Ordner der compose.yaml die Applikation gestartet:

```
docker compose up
```

Es kann nun das Django-UI über 127.0.0.1:80 im Browser aufgerufen werden, siehe Abbildung 4.2.

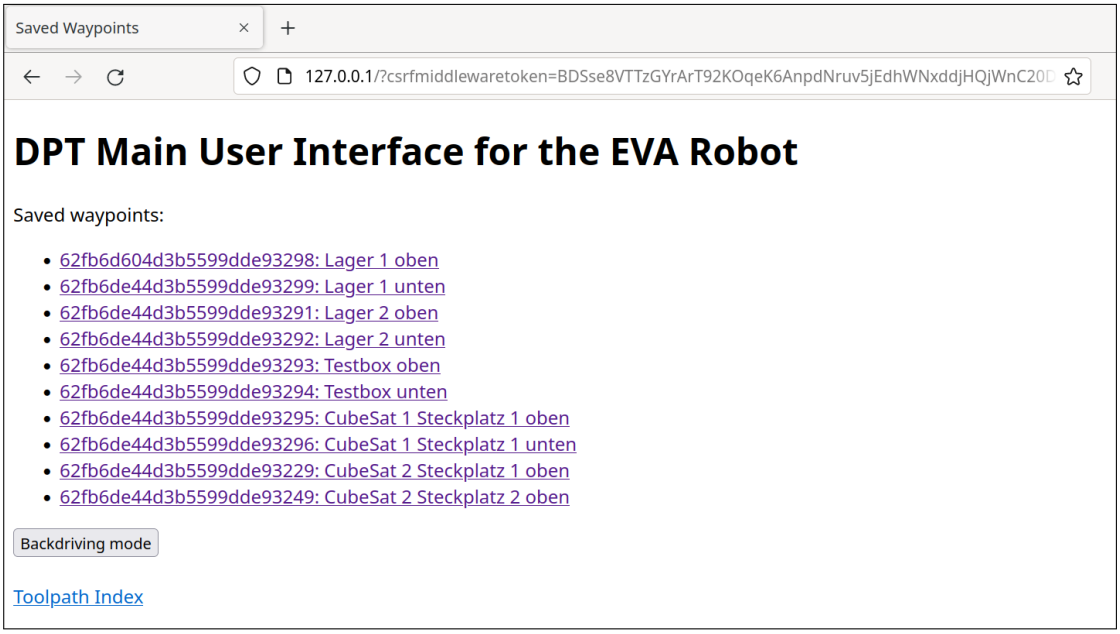


Abbildung 4.2.: Ein User-Interface des CMDPT, das einen Index der erstellten Waypoints des EVA-Roboters anzeigt.

Gezeigt wird ein User Interface, das exemplarische für den EVA Roboter relevante Wegpunkte bei der Montage von CubeSat Satelliten anzeigt. Die Montage dieser Satelliten ist der Produktionsprozess, der als Anwendungsfall des DPT im Projekt AI-in-Orbit-Factory dient. Die Waypoints sind Hyperlinks, die auf die passende Detailansicht des Wegpunkts leiten, siehe Abbildung. 4.3.

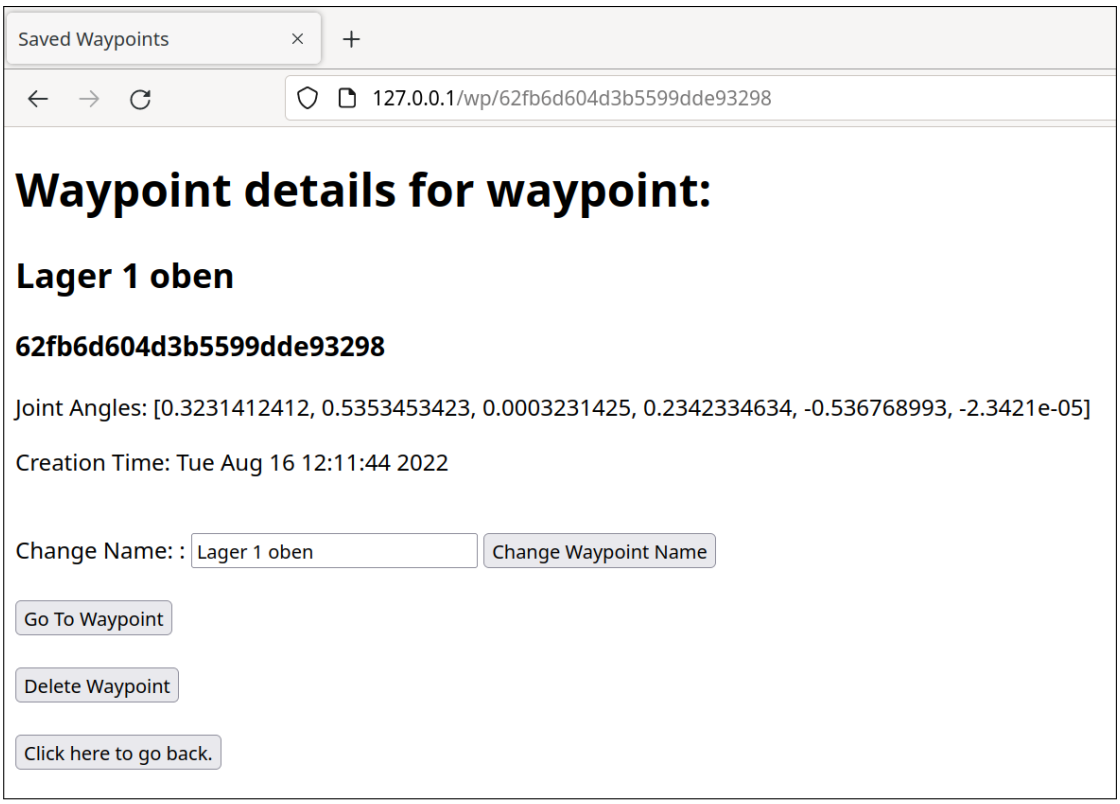


Abbildung 4.3.: Ein User-Interface des CMDPT, das einen Waypoint des EVA-Roboters im Detail darstellt und Manipulationen zulässt.

### 4.5.2. Demonstration von Kommunikation und SoA

Anhand der Änderung des Namens eines Wegpunktes im UI aus Abb. 4.3 soll das funktionierende Zusammenspiel der realisierten Module demonstriert werden. Dafür wird im Feld: „Change Name“ der Name zu „Demo-Waypoint“ geändert. Das wird mit einem Klick auf den Button „Change Waypoint Name“ bestätigt.

Die Ausgabe der Logs von Docker wird in Abbildung 4.4 gezeigt. Die Reihenfolge der Ausgabe wurde dem Verständnis halber angepasst.

Die HTTP-Anfrage des Browsers wird von NGINX verarbeitet und an die Django-App (process-ui) weitergeleitet. Diese sendet eine Nachricht an das Interface der Operational

```
ui-webserver    | 172.18.0.1 - - [12/Sep/2022:13:57:44 +0000] "POST /
   wp/62fb6d604d3b5599dde93298 HTTP/1.1" 200 2031 "http://127.0.0.1/
   wp/62fb6d604d3b5599dde93298" "Mozilla/5.0 (Windows NT 10.0; rv
   :104.0) Gecko/20100101 Firefox/104.0"

process-ui      | 2022-09-12 15:57:44,937 Send message to op-data-
   interface, request id: b'X5&\xe1\xf5\xfa\x9e\xd2': b["
   CHANGE_WP_NAME", "62fb6d604d3b5599dde93298", "Demo-Waypoint"]'

op-data-interface | INFO:root:op-data-interface_453740412899637388:
   Received message from b'process_ui_90496405698722320': ["
   CHANGE_WP_NAME", "62fb6d604d3b5599dde93298", "Demo-Waypoint"]

op-data-interface | INFO:root:Sending message: ('CHANGE_WP_NAME',) to
    b'process_ui_90496405698722320'

process-ui      | 2022-09-12 15:57:44,942
   process_ui_90496405698722320: Received message From op-data-
   interface: ["CHANGE_WP_NAME"]
```

Abbildung 4.4.: Docker-Log bei Änderung des Namens eines Wegpunktes in der UI.

Database. Hier ist das Protokoll der Nachrichten der SoA (siehe 3.3.7 und 4.2.3) gut zu erkennen: Die Anfrage besteht aus dem Anfragen-Typ „CHANGE_WP_NAME“ und dem Anfragen-Inhalt. Der Anfrage-Inhalt ist in diesem Fall die ID des Wegpunktes und der neue Name.

Das Operational Data Interface empfängt diese Anfrage und verarbeitet sie, indem eine Veränderung in der Operational Database getätigt wird. Da die Veränderung erfolgreich war, gibt das Interface nur den Anfragen-Typ „CHANGE_WP_NAME“ als Antwort wieder. Diese Nachricht empfängt die Django-App, die den Erfolg in das HTML-Dokument einbaut, das über den Webserver dem Nutzer zurückgegeben wird.

Insgesamt sind also 4 Container an der Aktion beteiligt: der Webserver, die Django-App, das OP-Database Interface und die MongoDB Datenbank.

## 4.6. Betrieb im Cluster

In diesem Abschnitt wird die Implementierung auf den Betrieb im Container-Cluster erweitert. Die Installation erfolgt über das Admin-Interface (Portainer). Diese Methode ist flexibler als die mit Docker-Compose aus Abschnitt 4.5, jedoch mit mehr Aufwand durch manuelles Konfigurieren verbunden.

Die Demonstration der Kommunikation der Module und der Django-App aus der Single-Host-Implementierung (4.5.2) wird nicht wiederholt. Stattdessen wird sich auf Neuigkeiten und Unterschiede zur Single-Host-Implementierung fokussiert.

### 4.6.1. Auswahl eines Orchestrators

Es handelt sich um eine experimentelle Implementierung auf sehr kleinem Maßstab. Auch spricht in diesem Fall der Einsatz des Orchestrators auf drei VMs auf einem Mittelklasse-Laptop gegen den Orchestrator Kubernetes. Kubernetes birgt einen signifikanten Overhead für den Betrieb von Nodes aufgrund der deutlich komplexeren Architektur (siehe 3.4.4). Daher wird der Empfehlung aus 3.4.4 nachgegangen und Docker-Swarm verwendet.

### 4.6.2. Rechenumgebung der Nodes

Für den Betrieb im Cluster werden mit Oracle VM VirtualBox drei Virtuelle Maschinen mit Debian 11 erstellt. Diese werden mit VirtualBox an dasselbe virtuelle Netzwerk angeschlossen. Eine Desktop-Umgebung wird nicht auf den VMs installiert, stattdessen ein SSH-Server und Docker. Der SSH-Server ermöglicht den Zugriff auf die Virtuellen Maschinen über den Host. Die VMs erhalten jeweils 1 GB (virtuellen) Arbeitsspeicher.

### 4.6.3. Docker-Swarm

Jede der drei VMs betreibt eine Docker-Swarm Node im Manager-Modus. Das hat den Grund, dass ein einzelner Manager nicht ausfallsicher ist und der Betrieb von zwei Managern nicht zu empfehlen ist. Im Fall von zwei Managern würde es zu einem Deadlock kommen, falls kein Konsens zwischen beiden besteht. Kann kein Konsens erreicht werden, sind die Programme im Cluster zwar noch betriebsfähig, aber deren Verwaltung nicht

mehr. Gibt es drei Manager und der Zustand eines Managers divergiert vom Rest, kann noch ein Konsens über den korrekten Zustand erreicht werden. [Doc22e]

Die Docker-Engine wird auf der ersten VM mit `docker swarm init` in den Swarm-Modus versetzt. Mit dem Befehl `docker swarm join-token (worker|manager)` wird eine Zeichenkette erzeugt, die zusammen mit der IP dieser VM verwendet wird, um dem Swarm beizutreten. In diesem Fall wird der Token für den Beitritt als Manager erzeugt. Die weiteren VMs treten den Swarm mit `docker swarm join --token [token] [IP]:2377` bei. Nun ist das Cluster betriebsfähig.

### 4.6.4. Administratives Interface

In dieser Implementierung erfolgt die Installation und Verwaltung der Module über das Admin-Interface. Dafür wird das Open-Source-Programm Portainer verwendet. Dafür wird die geeignete compose.yaml von der Portainer-Webseite geladen und mit

```
docker stack deploy -c portainer-agent-stack.yml portainer
```

eingesetzt [Por21]. Der Portainer-Stack besteht aus dem Portainer-Server und den Agenten, von denen auf jeder Manager-Node einer läuft.

Abbildung 4.5 zeigt das Administrative Interface, das die Services des CMDPT auflistet. Dies sind die Module. In der Spalte „Scheduling Mode“ ist das UI-Element „Scale“ zu erkennen, mit dem der Service durch einfache Eingabe horizontal skaliert werden kann. Über den Button „Add service“ lässt sich ein neuer Service/ein neues Modul hinzufügen.

### 4.6.5. Hinzufügen/Anpassen von Modulen

Vor dem Einbau von CMDPT-Modulen muss das CMDPT-Netzwerk mit Overlay-Driver erstellt werden. Dies geschieht im Admin-Interface unter dem Reiter „Network“ →„Add Network“. Um Portainer Images von einer eigenen Registry beziehen zu können, muss diese im Reiter „Registries“ hinzugefügt werden.

Das UI zum Hinzufügen von Services verfügt über alle wichtigen Optionen, die der Docker Client besitzt. Ein ähnliches UI steht auch für die Änderung des Services nach dessen Erstellung zur Verfügung. Beim Hinzufügen der Module für diese prototypische Implementierung wurde sich an der Konfiguration des CMDPT, wie sie in der compose.yaml in Appendix A hinterlegt ist, orientiert. Es sind also die gleichen Module in der gleichen

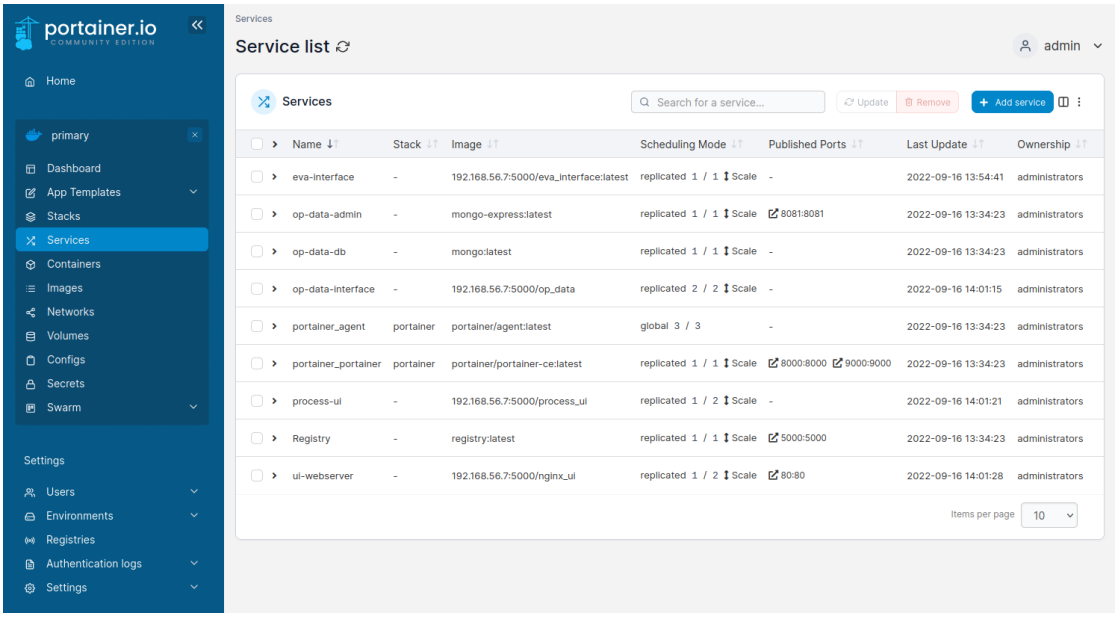


Abbildung 4.5.: Auflistung der Module des CMDPT als Services im Administrative Interface (Portainer)

Konfiguration wie in der Single-Host-Implementierung vorhanden. Der Einsatz im Cluster ermöglicht aber neue Konfigurationsmöglichkeiten, wie die Festlegung der Anzahl der Container-Replika oder die Positionierung der Container im Cluster.

### 4.6.6. Überblick über den verteilten CMDPT

Abbildung 4.6 gibt eine Übersicht über die Verteilung der Services auf den Hosts. Beispielsweise liegt das Operational Data Interface auf den Hosts deb2 und deb3 vor. Dies erzeugt die Ausfallsicherheit und zeigt die Skalierbarkeit der Module, wie beschrieben in 3.2.2.

### 4.6.7. Skalierung der Module

Um die Module/Services zu skalieren kann im Services-Interface (siehe Kap. 4.5) von Portainer die Schaltfläche „Scale“ verwendet werden. Dort lässt sich dann die Anzahl der Container dieses Services im Cluster ändern, siehe Kap. 4.7


Abbildung 4.6.: Host-basierte Übersicht über die Verteilung der Services im CMDPT-Cluster

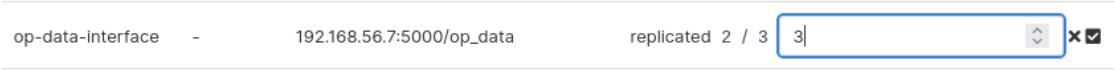


Abbildung 4.7.: Schaltfläche zur Skalierung der Module im Administrative Interface (Portainer)

# 5. Verifizierung und Validierung

Anhand der prototypischen Implementierung im Cluster aus 4.6 sollen die Anforderungen aus 3.1 verifiziert und das Systems CMDPT als Ganzes validiert werden. In der Validierung wird der Nutzen des Konzepts in mehreren Anwendungsfällen überprüft.

## 5.1. Verifizierung

Eine Übersicht über die Verifizierung der Anforderungen ist in Tabelle 5.1 gegeben. Alle Anforderungen beziehen sich, wenn nicht anders vermerkt, auf die Module des CMDPT. Verifizierte Anforderungen sind mit ✓ versehen, Anforderungen, die nicht anhand der Implementierung verifiziert werden konnten, mit -. Anforderungen mit Verifizierung unter Vorbehalt sind mit (✓) gekennzeichnet.

Alle Verifizierungen werden in den folgenden Abschnitten erläutert.

#### 1. Modulare Architektur ✓

Es wurde erfolgreich eine modulare Architektur des DPT konzipiert und umgesetzt.

#### 2. Flexible Verknüpfung ✓

Wie in 4.3.2 und 4.3.3 gezeigt, lassen sich die Module mit Docker flexibel miteinander verknüpfen, besonders unter Zuhilfenahme von Docker-Compose oder Portainer. Module können während dem Betrieb des CMDPT ausgetauscht, neu konfiguriert oder neu eingesetzt werden. Vor allem die Benutzeroberfläche von Portainer (Admin-UI) erleichtert das Zusammenstellen der gewünschten CMDPT-Konfiguration.

| Anforderung | verifiziert |
|---|---|
| 1. Modulare Architektur | ✓ |
| 2. Flexible Verknüpfung | ✓ |
| 3. Unabhängigkeit von ausf. Ressource | ✓ |
| 4. Auslagerungsfähigkeit | ✓ |
| 5. Vollständigkeit des DPT | - |
| 6. Modularisierbarkeit beliebiger DPT-Teile | (✓) |
| 7. Kontextunabhängigkeit | ✓ |
| 8. Eigenständige Existenz | ✓ |
| 9. Einheitliche Schnittstellen | (✓) |
| 10. Kommunikationsfähigkeit | ✓ |
| 11. Vollständige Portabilität des DPT | (✓) |
| 12. Lose Kopplung durch Microservice-Architektur | ✓ |
| 13. Skalierbarkeit | (✓) |
| 14. DPT-Modul-Image-Speicher | ✓ |

Tabelle 5.1.: Verifizierung der Anforderungen des CMDPT

**3. Unabhängigkeit von der ausführenden Ressource ✓**

Die Unabhängigkeit der Module von der ausführenden Ressource kann durch den Betrieb im Cluster in Kap. 4.6 nachgewiesen werden. Grundlage dafür ist die Portabilität der Container (siehe Kap. 3.2). Die Module existieren virtualisiert und sind nur zeitweise an spezifische Hosts gebunden. Die Container können auf Befehl vom Nutzer zwischen den Hosts bewegt werden. Ausgenommen von dieser Unabhängigkeit sind Module, die für den Betrieb an spezifischen Hosts angeschlossene Hardware benötigen und daher lokal gebunden sind.

**4. Auslagerungsfähigkeit ✓**

Der Betrieb im Cluster aus Kap. 4.6 zeigt, dass Module auf externe Ressourcen ausgelagert werden können.

**5. Vollständigkeit des DPT -**

Der Betrieb eines vollständigen DPT als CMDPT konnte nicht gezeigt werden, da nur exemplarische Module containerisiert wurden. Ein vollständiger Demonstrator ließ sich auch aufgrund der Voraussetzungen am Projekt AI-in-Orbit-Factory nicht realisieren.

**6. Modularisierbarkeit beliebiger Teile des DPT (✓)**

Es wurde exemplarisch gezeigt, dass Funktionen des DPT in Module separiert und containerisiert werden können. Dennoch kann nicht ausgeschlossen werden, dass sich Teile des DPT unter besonderen Umständen nicht modularisieren lassen.

**7. Kontextunabhängigkeit ✓**

Aufgrund der Virtualisierung durch Container können die Module als kontextunabhängig angesehen werden. Die Prozesse der Container sind von den meisten Host-Prozessen und von anderen Containern isoliert (siehe Kap. 2.3). Das Verhalten der Module ist daher von deren Rechenumgebung außerhalb des Containers weitestgehend unabhängig. Dies kann durch den Betrieb im Cluster in Kap. 4.6 gezeigt werden, da dort die Module beliebig zwischen Hosts bewegt werden können, ohne dass sich das Systemverhalten ändert.

**8. Eigenständige Existenz ✓**

Alle Module können eigenständig als Container betrieben werden, ohne dass diese Fehler im Systemverhalten aufweisen. Das kann in 4.6.6 nachgewiesen werden. Dort werden die Module erst nacheinander gestartet. Diese Eigenständigkeit ist ein wichtiger Faktor für die Flexibilität des Systems.

**9. Einheitliche Schnittstellen (✓)**

Im Rahmen der Implementierung wurden Schnittstellen eingeführt, die für die umgesetzten Module sinnvoll sind. Die Nachrichten, die die Module im Prototyp untereinander austauschen, folgen einem festen Muster, siehe 4.2.3. Ein abgeschlossenes Konzept für die Schnittstellen aller einzelnen Standard-Module im CMDPT wurde aber nicht entwickelt.

### 10. Kommunikationsfähigkeit ✓

Die Kommunikationsfähigkeit der Module untereinander und nach außen kann durch die Logs der Demonstration in 4.5.2 nachgewiesen werden. Untereinander wird über ein internes Netzwerk unter Zuhilfenahme einer MOM kommuniziert, nach außen über die Port-Zuordnung der Container-Engine.

### 11. Vollständige Portabilität (✓)

Alle prototypisch modularisierten Teile des DPT konnten containerisiert werden. Die Portabilität der containerisierten Module zeigt sich durch deren Migrationsfähigkeit im Cluster in 4.6.6. In Anlehnung an die Verifizierung von Punkt 5 kann aber nicht nachgewiesen werden, dass ein umfangreicherer DPT vollständig containerisiert werden kann.

### 12. Lose Kopplung durch Microservice-Architektur ✓

Die Microservice-Architektur, die durch lose Kopplung der Komponenten des CMDPT die Skalierbarkeit sicherstellt, wurde implementiert, siehe 4.2.3.

### 13. Skalierbarkeit (✓)

Die Skalierbarkeit wird in der Cluster-Implementierung demonstriert, siehe dazu 4.6.6 und 4.6.7. Mehrere Services liegen dort horizontal skaliert über mehrere Nodes vor. Das Administrative Interface befähigt den Nutzer dazu, in kürzester Zeit die Skalierung der Komponenten des CMDPT vorzunehmen.

Einzig die Datenbanken wurden nicht skalierbar implementiert. Das hat den Grund, dass der Aufwand dafür den Nutzen auf dem geringen Maßstab dieser Implementierung übersteigt. Die horizontale Skalierung von MongoDB Datenbanken wäre über das *Sharding* aber möglich [Mon22].

#### 14. DPT-Image-Speicher ✓

Die Images der CMDPT-Module werden auf einer dedizierten Registry gespeichert und dort verwaltet, siehe 4.4.1. Diese kann extern vorliegen und muss lediglich von den Hosts über das Internet erreichbar sein.

## 5.2. Validierung

Die Validierung des Konzeptes in diesem Abschnitt dient zur Überprüfung der Funktionalität des CMDPT anhand dessen Anwendungsfällen. Die Anwendungsfälle werden aus den wichtigsten Gründen für und gegen einen praxisnahen Einsatz des CMDPT hergeleitet. Dies sind die variable und hohe Rechenleistung, die ein DPT erfordern kann, und damit Verbunden die Möglichkeit der Auslagerung von Modulen in die Cloud; die Verlässlichkeit bei kritischen Anwendungen, die Ermöglichung eines DPT-Modul Ökosystems und der Einsatz bei Echtzeit- und Latenz-kritischen Systemen. Vor diesem Hintergrund wird die Eignung des CMDPT als Ersatz für einen monolithischen DPT in der Praxis erläutert.

In dieser Arbeit wurde ein System vorgeschlagen, das den DPT in einzelne Module zerlegt, die austauschbar und wiederverwertbar sind. Die lose Kopplung der Module, vor allem durch die SoA, ermöglicht die Integration von neuen Modulen in das bestehende System. Die Modularisierung führt auch zu einer Reduktion der Kompliziertheit des Systems DPT.

Aufbauend auf der Modularisierung wurde die Containerisierung der Module vollzogen. Die Containerisierung bietet den Modulen des CMDPT vor allem Portabilität, Skalierbarkeit und Fehlertoleranz. Der Betrieb von Containern erfordert eine unterstützende Infrastruktur für den DPT, die in der Architektur des CMDPT enthalten ist.

Insgesamt kann damit die Flexibilisierung des DPT als erfolgreich angesehen werden. Unter Flexibilisierung werden in diesem Fall vor allem die Austauschbarkeit, Wiederverwertbarkeit, Portabilität und Skalierbarkeit der Module und die Erweiterungsfähigkeit des DPT zusammengefasst.

#### Anwendungsfall DPT mit variabler und hoher erforderlicher Rechenleistung

Die horizontale Skalierbarkeit des CMDPT, gezeigt in Kap. 4.6, ermöglicht beliebig anspruchsvolle Rechenaufgaben für den DPT. Das ist vor allem bei Angelegenheiten der

Künstlichen Intelligenz und Simulation von großer Bedeutung für die Industrie. Bei schwankender Belastung ermöglicht die Skalierbarkeit, eine Überbelegung von Rechenressourcen zu verhindern.

### Anwendungsfall Auslagerung von DPT-Modulen in die Cloud

Die Abstraktion der Software von der Hardware durch Virtuelle Maschinen oder Container-Virtualisierung ist mittlerweile der Standard im Cloud-Computing. Durch den Einsatz der Containertechnologie ist eine Auslagerung von besonders rechenintensiven Teilen des DPT in die Cloud möglich. Dies kann wirtschaftliche Vorteile haben. Auch möglich ist der Einsatz bestimmter Module an der Edge, das heißt in Nähe zu den relevanten physischen Assets. Besonders für Latenz-kritische Module ist dies relevant.

### Anwendungsfall DPT für verlässliche Anwendungen

Die horizontale Skalierbarkeit des CMDPT im Cluster führt zu einer hohen Ausfalltoleranz. Die verteilte Natur des CMDPT lässt den Ausfall von einzelnen Containern oder sogar Hosts zu, ohne dass die Funktionstüchtigkeit gefährdet wird. Aber auch Single-Host CMDPTs profitieren von der schnellen und zuverlässigen Installationsfähigkeit von Containern. Der Host kann bei Ausfall in kurzer Zeit mit geringem Aufwand gewechselt werden. Das birgt großes Potenzial für Ausfall-kritische Anwendungen.

### Ermöglichung eines DPT-Modul-Ökosystems

Die gezeigte Portabilität der containerisierten Module des CMDPT führt zur optimierten Modifizierbarkeit des DPT. Module sind unkompliziert und zuverlässig installierbar und deinstallierbar. Es konnte gezeigt werden, dass die Installation eines neuen Moduls von einer Image-Registry in den meisten Fällen nur wenige Schritte manueller Konfiguration benötigt. Aufgrund der Isolation vom Hostsystem und anderen Modulen kann es außerdem grundsätzlich nicht zu Kompatibilitätsproblemen und anderen unerwünschten Nebenwirkungen beim Betrieb mehrerer Module kommen. Das hat vor allem Vorteile bei Verwendung unterschiedlicher Versionen oder Modulen von verschiedenen Herstellern. Das alles erlaubt eine sehr dynamische und flexible Konfiguration des CMDPT, was lange Lebensdauern begünstigt.

Kontextunabhängige Module können in unterschiedlichen Anwendungsfällen eingesetzt werden, ohne dass sie für jeden Einsatz neu entwickelt werden müssen (Wiederverwertbarkeit). Die Separation der Module ermöglicht eine Arbeitsteilung und damit Spezialisierung von Herstellern. Das fördert die Standardisierung von Modulen.

Wird außerdem die Möglichkeit der Speicherung der Module als kompakte Images auf Image-Registries beachtet, erschließt sich eine Prädestination dieser modularen Architektur für ein DPT-Modul-Ökosystem.

### Anwendungsfall DPT für Echtzeit- und Latenz-kritische Systeme

Der Einsatz des CMDPT für echtzeit- und Latenz-kritische Systeme birgt Risiken. Die Erforschung von Echtzeit-Containern ist noch nicht sehr fortgeschritten (siehe 3.2.2). Die verteilte Architektur des CMDPT erhöht außerdem die Latenzen der internen DPT-Kommunikation gegenüber monolithischen und nicht-containerisierten Architekturen. Daher kann Anwendungsfall Echtzeit- und Latenz-kritische Systeme vom vorgeschlagenen Konzept nicht abgedeckt werden.

# 6. Ausblick

Während die Software-Modularität als Konzept schon älter als 20 Jahre ist, ist die Containertechnologie erst in den letzten 10 Jahren für wichtige Entwicklungen im Cloud-Computing Bereich verantwortlich. Der Einsatz dieser Technologie in Maschinenbau-nahen Anwendungen in Industrie und Forschung findet erst in den letzten drei Jahren sehr langsam statt. Während das in dieser Arbeit ausgearbeitete Konzept des CMDPT noch nicht allen Anwendungsfeldern gewachsen ist, konnten doch deutliche Vorteile der Containerisierung und Modularisierung gezeigt werden. Hervorzuheben sind die Portabilität und Skalierbarkeit des CMDPT sowie die Austauschbarkeit, Wiederverwertbarkeit und Erweiterbarkeit dessen Module. Die Kontextunabhängigkeit der Module gepaart mit der Portabilität durch die Containertechnologie ermöglichen zudem ein Modul-Ökosystem, das Standardisierung fördert.

Es fehlen aber noch wichtige Forschungsobjekte auf dem Weg zum Einsatz des CMDPT:

- Eine umfangreiche quantitative Analyse und Auswertung mit Vergleich von modular-containerisierter und nichtmodularer DPT-Architektur wurde nicht vorgenommen, da der Prototyp am Projekt AI-in-Orbit Factory keine besonders hohen Anforderungen an Latenz, Datenübertragungsmengen oder Rechenleistung stellt. Eine quantitative Studie müsste genau diese Bereiche auswerten.
- Ein umfangreicher CMDPT-Demonstrator im Industrie-nahen Einsatz ist nötig um die Effekte der Skalierung zu beobachten und die Einschnitte durch Nachteile des CMDPT wie Latenz und große Datenströme zu beurteilen.
- Zu den großen Herausforderungen des CMDPT gehören außerdem die Kommunikation und die Echtzeitanforderung. Die Kommunikation zwischen den Modulen ist entscheidend für den Funktionsgrad des Gesamtsystems. Deshalb lässt sich ein starker Forschungsbedarf bei Latenz-armer, zuverlässiger Kommunikation feststellen.

Unter Produktionsbedingungen ist zudem der zeitliche Determinismus des CMDPT-Verhaltens wichtig. Es braucht dafür Fortschritte in der Forschung zu Echtzeit-Betriebssystemen, -Containern und -Kommunikation.

Die am DPT ausgearbeitete Architektur könnte sich in abgewandelter Form auch auf andere DT-Anwendungen übertragen lassen. In einem ersten Schritt könnte das ausgearbeitete Konzept des CMDPT auf einen allgemeinen DT generalisiert werden. In diesem Fall müsste eine umfangreiche Analyse zum weiträumigen Einsatz von Containern in Multi-DT-Applikationen getätigt werden.

Die Entwicklungen in der Industrie 4.0 sind sehr dynamisch und technologieoffen. Eine umfangreichere Betrachtung der Containertechnologie in der Forschung zur Informationstechnologie in der Produktion könnte für verschiedene Aufgabenstellungen neue Lösungsansätze liefern.

# A. Docker-Compose des nicht-verteilten CMDPT

```yaml
version: '3.8'

services:
  base-module:
    build:
      context: ./CONTROL/base_module
    image: ${REGISTRY}/base_module
    deploy:
      mode: replicated
      replicas: 0
      restart_policy:
        condition: none

  op-data-db:
    container_name: op-data-db
    image: mongo
    networks:
      - dpt
    volumes:
      - op-data:/data/db
    logging:
      driver: none

  op-data-admin:
    image: mongo-express
    networks:
      - dpt
    environment:
      - ME_CONFIG_MONGODB_SERVER=op-data-db
      - ME_CONFIG_BASICAUTH_USERNAME=admin
```

```
    - ME_CONFIG_BASICAUTH_PASSWORD=desktop
  ports:
    - 8081:8081

op-data-interface:
  container_name: op-data-interface
  build:
    context: ./DATA/op_data
    args:
      REGISTRY: ${REGISTRY}
  image: ${REGISTRY}/op_data
  networks:
    - dpt
  environment:
    - EVA_INTERFACE_ADDR=eva-interface
    - OP_DATA_ADDR=op-data-interface
    - DB_ADDRESS=op-data-db

process-ui:
  container_name: process-ui
  build:
    context: ./UI/django_app
    args:
      REGISTRY: ${REGISTRY}
  image: ${REGISTRY}/process_ui
  networks:
    - dpt
    - webserver
  environment:
    - WEBSERVER_IP=127.0.0.0/24
    - EVA_INTERFACE_ADDR=eva-interface
    - OP_DATA_ADDR=op-data-interface

ui-webserver:
  container_name: ui-webserver
  build:
    context: ./UI/nginx_conf
  image: ${REGISTRY}/nginx_ui
  networks:
    - webserver
  ports:
    - "80:80"
```

```
    environment:
      - APP_SERVER_ADDR=process-ui:8000

  eva-interface:
    container_name: eva-interface
    build:
      context: ./INTERFACE/eva
      args:
        REGISTRY: ${REGISTRY}
    image: ${REGISTRY}/eva_interface
    networks:
      - dpt
    environment:
      - OP_DATA_ADDR=op-data-interface


networks:
  dpt:
  webserver:

volumes:
  op-data:
    version: '3.8'
```

# B. Operational Data Interface

```
"""
author: robert.knobloch@stud.tu-darmstadt.de

Program managing the operational data storage (MongoDB) of the DPT.
PyMongo docs: https://pymongo.readthedocs.io/en/stable/
"""

from pymongo import MongoClient
from bson.objectid import ObjectId
from bson.json_util import dumps
import time
import logging
import os
import json


from base_module import BaseModule

logging.basicConfig(level=logging.DEBUG)

class OpData(BaseModule):
    """
    Class for interfacing with the Operational Data Storage

    Necessary OS Environment Variables:
    -----------------------------------
    DB_ADDRESS: ip or domain dame of the mongodb database

    Service requests:
    -----------------
    SHUTDOWN, NEW_WP, GET_WP, GET_ALL_WP_IDS, CHANGE_WP_NAME,
    NEW_TP, GET_TP, ADD_TO_TP, RM_FROM_TP
    """

    def __init__(self):
        super().__init__()

        # MongoDB Setup
        db_address = os.environ["DB_ADDRESS"]
        self.client = MongoClient(db_address)
        self.db = self.client["dpt_op_data"]

        self.start(self.service_loop())

    async def service_loop(self) -> None:
        """
        Loop for listening to incoming requests.
        Expects a tuple with the first entry being the request type.
        """

        col_waypoints = self.db["waypoints"]
        col_toolpaths = self.db["toolpaths"]

        while True:
```

```
            (sender, req_id, msg) = await self.server_receive()
            resp_id = req_id

            match msg:
                case ["SHUTDOWN"]:
                    break
                case ["NEW_WP", joint_angles]:
                    post = {"coordinates": joint_angles,
                        "wp_name": "New WP",
                        "creation_time": time.time()}
                    wp_id = col_waypoints.insert_one(post).inserted_id

                case ["GET_WP", wp_id]:
                    wp_id = ObjectId(wp_id) # Convert to mongodb id object
                    wp_doc = col_waypoints.find_one(wp_id)
                    if wp_doc is None:
                        await self.server_transmit(sender, resp_id, ("NONEXISTENT_OBJECT", str(wp_id)))
                        continue

                    wp_coor = wp_doc["coordinates"]
                    wp_name = wp_doc["wp_name"]
                    wp_time = wp_doc["creation_time"]
                    await self.server_transmit(sender, resp_id,
                        ("GET_WP", str(wp_id), wp_name, wp_coor, wp_time))

                case ["GET_ALL_WP_IDS"]:
                    all_wps_cursor = col_waypoints.find({})
                    wp_all_ids = [str(wp["_id"]) for wp in all_wps_cursor]
                    all_wps_cursor = col_waypoints.find({})
                    wp_all_names = [wp["wp_name"] for wp in all_wps_cursor]
                    await self.server_transmit(sender, resp_id, ("GET_ALL_WP_IDS", wp_all_ids, wp_all_names))

                case ["DEL_WP", wp_id]:
                    wp_id = ObjectId(wp_id)
                    result = col_waypoints.delete_one({"_id": wp_id})
                    if result.acknowledged and result.deleted_count == 1:
                        await self.server_transmit(sender, resp_id, ("DEL_WP",))
                    else:
                        await self.server_transmit(sender, resp_id, ("UNEXPECTED_FAILURE",))

                case ["CHANGE_WP_NAME", wp_id, wp_name]:
                    wp_id = ObjectId(wp_id)
                    new_value = {"$set": {"wp_name": wp_name}}
                    result = col_waypoints.update_one({"_id": wp_id}, new_value)
                    if result.acknowledged and result.modified_count == 1:
                        await self.server_transmit(sender, resp_id, ("CHANGE_WP_NAME",))
                    else:
                        await self.server_transmit(sender, resp_id, ("UNEXPECTED_FAILURE",))

                # Code an dieser Stelle für den Anhang gekürzt

                case _:
                    await self.server_transmit(sender, resp_id, ("UNKNOWN_REQUEST",))


if __name__ == "__main__":
    opd = OpData()
```

# Literatur


[Abe+15] Eberhard Abele u. a.
*Industrie 4.0 - Potentiale, Nutzen Und Good-Practice-Beispiele Für Die Hessische Industrie, Zwischenbericht Zum Projekt Effiziente Fabrik 4.0*.
Bamberg, 2015.

[Ala+20] Ameer B. A. Alaasam u. a. „Analytic Study of Containerizing Stateful Stream Processing as Microservice to Support Digital Twins in Fog Computing“.
In: *Programming and Computer Software* 46.8 (Dez. 2020), S. 511–525.
ISSN: 0361-7688, 1608-3261. DOI: 10.1134/S0361768820080083.
URL: http://link.springer.com/10.1134/S0361768820080083
(besucht am 05. 05. 2022).

[Alp22] Alpine Linux Development Team. *About | Alpine Linux*. 2022. URL:
https://www.alpinelinux.org/about/ (besucht am 18. 08. 2022).

[Bal09] Helmut Balzert.
*Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*.
Heidelberg: Spektrum Akademischer Verlag, 2009.
ISBN: 978-3-8274-1705-3 978-3-8274-2247-7.
DOI: 10.1007/978-3-8274-2247-7.
URL: http://link.springer.com/10.1007/978-3-8274-2247-7
(besucht am 12. 05. 2022).

[BB22] Marco Bertoni und Alessandro Bertoni. „Designing Solutions with the Product-Service Systems Digital Twin: What Is Now and What Is Next?“
In: *Computers in Industry* 138 (Juni 2022), S. 103629. ISSN: 01663615.
DOI: 10.1016/j.compind.2022.103629.
URL: https://linkinghub.elsevier.com/retrieve/pii/S0166361522000240 (besucht am 05. 05. 2022).

[Boe15] Carl Boettiger. „An Introduction to Docker for Reproducible Research“. In: *ACM SIGOPS Operating Systems Review* 49.1 (20. Jan. 2015), S. 71–79. ISSN: 0163-5980. DOI: 10.1145/2723872.2723882. URL: https://dl.acm.org/doi/10.1145/2723872.2723882 (besucht am 12. 05. 2022).

[Bor+17] Kirill Borodulin u. a. „Towards Digital Twins Cloud Platform: Microservices and Computational Workflows to Rule a Smart Factory“. In: *Proceedings of The10th International Conference on Utility and Cloud Computing*. UCC '17: 10th International Conference on Utility and Cloud Computing. Austin Texas USA: ACM, 5. Dez. 2017, S. 209–210. ISBN: 978-1-4503-5149-2. DOI: 10.1145/3147213.3149234. URL: https://dl.acm.org/doi/10.1145/3147213.3149234 (besucht am 12. 05. 2022).

[BV20] Hartwig Baumgärtel und Richard Verbeet. „Service- und Agenten-basierte Ansätze für die Implementierung von I4.0-Systemen“. In: *Handbuch Industrie 4.0*. Hrsg. von Michael ten Hompel, Birgit Vogel-Heuser und Thomas Bauernhansl. Springer Reference Technik. Berlin, Heidelberg: Springer Berlin Heidelberg, 2020, S. 1–36. ISBN: 978-3-662-45537-1. DOI: 10.1007/978-3-662-45537-1_129-1. URL: http://link.springer.com/10.1007/978-3-662-45537-1_129-1 (besucht am 05. 05. 2022).

[C M+06] C. Matthew MacKenzie u. a. *Reference Model for Service Oriented Architecture 1.0*. 8. Feb. 2006. URL: http://www.oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf (besucht am 18. 08. 2022).

[CNC22] CNCF. *Graduated and Incubating Projects*. Cloud Native Computing Foundation. 2022. URL: https://www.cncf.io/projects/ (besucht am 20. 08. 2022).

[Con21] Containers Organisation. *What Is Podman?* 5. Apr. 2021. URL: https://podman.io/whatis.html (besucht am 22. 08. 2022).

[Cur05] Edward Curry. „Message-Oriented Middleware“. In: *Middleware for Communications*. Hrsg. von Qusay H. Mahmoud. Chichester, UK: John Wiley & Sons, Ltd, 1. Juli 2005, S. 1–28. ISBN: 978-0-470-86208-7 978-0-470-86206-3. DOI: 10.1002/0470862084.ch1. URL: https:

//onlinelibrary.wiley.com/doi/10.1002/0470862084.ch1 (besucht am 25. 08. 2022).

[Doc17] Docker, Inc. *What Is Containerd ? - Docker*. 7. Aug. 2017. URL: https://www.docker.com/blog/what-is-containerd-runtime/ (besucht am 20. 08. 2022).

[Doc22a] Docker, Inc. *Docker Overview*. Docker Documentation. 23. Aug. 2022. URL: https://docs.docker.com/get-started/overview/ (besucht am 23. 08. 2022).

[Doc22b] Docker, Inc. *Docker Registry*. Docker Documentation. 6. Sep. 2022. URL: https://docs.docker.com/registry/ (besucht am 07. 09. 2022).

[Doc22c] Docker, Inc. *Dockerfile Reference*. Docker Documentation. 29. Aug. 2022. URL: https://docs.docker.com/engine/reference/builder/ (besucht am 29. 08. 2022).

[Doc22d] Docker, Inc. *How Services Work*. Docker Documentation. 24. Aug. 2022. URL: https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/ (besucht am 24. 08. 2022).

[Doc22e] Docker, Inc. *Raft Consensus in Swarm Mode*. Docker Documentation. 7. Sep. 2022. URL: https://docs.docker.com/engine/swarm/raft/ (besucht am 07. 09. 2022).

[Doc22f] Docker, Inc. *Swarm Mode Overview*. Docker Documentation. 19. Aug. 2022. URL: https://docs.docker.com/engine/swarm/ (besucht am 20. 08. 2022).

[Doc22g] Docker, Inc. *Test an Insecure Registry | Docker Documentation*. 2022. URL: https://docs.docker.com/registry/insecure/ (besucht am 16. 09. 2022).

[Doc22h] Docker, Inc. *Use Bridge Networks*. Docker Documentation. 19. Aug. 2022. URL: https://docs.docker.com/network/bridge/ (besucht am 21. 08. 2022).

[Doc22i] Docker, Inc. *Use Volumes*. Docker Documentation. 5. Juli 2022. URL: https://docs.docker.com/storage/volumes/ (besucht am 06. 07. 2022).

[Dor+22] Borislav Dordevic u. a. „Performance Comparison of Docker and Podman Container-Based Virtualization“.
In: *2022 21st International Symposium INFOTEH-JAHORINA (INFOTEH)*.
2022 21st International Symposium INFOTEH-JAHORINA (INFOTEH).
East Sarajevo, Bosnia and Herzegovina: IEEE, 16. März 2022, S. 1–6.
ISBN: 978-1-66543-778-3.
DOI: 10.1109/INFOTEH53737.2022.9751277.
URL: https://ieeexplore.ieee.org/document/9751277/
(besucht am 03. 07. 2022).

[DTB22] Suparna Dhar, Pratik Tarafdar und Indranil Bose. *Understanding the Evolution of Digital Twin and Its Impact a Topic Modeling Approach*.
SSRN Scholarly Paper 4003286.
Rochester, NY: Social Science Research Network, 7. Jan. 2022.
DOI: 10.2139/ssrn.4003286.
URL: https://papers.ssrn.com/abstract=4003286 (besucht am 05. 05. 2022).

[Emi19] Emiliano Casalicchio. „Container Orchestration: A Survey“.
In: *Systems Modeling: Methodologies and Tools*.
Hrsg. von Antonio Puliafito und Kishor S. Trivedi.
EAI/Springer Innovations in Communication and Computing.
Cham: Springer International Publishing, 2019, S. 221–235.
ISBN: 978-3-319-92377-2 978-3-319-92378-9.
DOI: 10.1007/978-3-319-92378-9.
URL: http://link.springer.com/10.1007/978-3-319-92378-9
(besucht am 20. 08. 2022).

[Emi20] Rasmus Emilsson.
*Container Performance Benchmark between Docker, LXD, Podman & Buildah*.
1. Juli 2020. URL: http://urn.kb.se/resolve?urn=urn%3Anbn%3Ase%3Ahis%3Adiva-18748 (besucht am 10. 08. 2022).

[ES21] Timm Eichstädt und Stefan Spieker.
*52 Stunden Informatik: Was jeder über Informatik wissen sollte*.
Wiesbaden: Springer Fachmedien Wiesbaden, 2021.
ISBN: 978-3-658-33428-4 978-3-658-33429-1.
DOI: 10.1007/978-3-658-33429-1. URL:

https://link.springer.com/10.1007/978-3-658-33429-1 (besucht am 17. 09. 2022).

[Fel+15] Wes Felter u. a. „An Updated Performance Comparison of Virtual Machines and Linux Containers“. In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). Philadelphia, PA, USA: IEEE, März 2015, S. 171–172. ISBN: 978-1-4799-1957-4. DOI: 10.1109/ISPASS.2015.7095802. URL: http://ieeexplore.ieee.org/document/7095802/ (besucht am 09. 06. 2022).

[Gri19] Michael W. Grieves. „Virtually Intelligent Product Systems: Digital and Physical Twins“. In: *Complex Systems Engineering: Theory and Practice*. Hrsg. von Shannon Flumerfelt u. a. Reston, VA: American Institute of Aeronautics and Astronautics, Inc., Jan. 2019, S. 175–200. ISBN: 978-1-62410-564-7 978-1-62410-565-4. DOI: 10.2514/5.9781624105654.0175.0200. URL: https://arc.aiaa.org/doi/10.2514/5.9781624105654.0175.0200 (besucht am 05. 05. 2022).

[Gui22] Guido van Rossum. *What Is Python? Executive Summary*. Python.org. 2022. URL: https://www.python.org/doc/essays/blurb/ (besucht am 01. 09. 2022).

[GV17] Michael Grieves und John Vickers. „Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems“. In: *Transdisciplinary Perspectives on Complex Systems*. Hrsg. von Franz-Josef Kahlen, Shannon Flumerfelt und Anabela Alves. Cham: Springer International Publishing, 2017, S. 85–113. ISBN: 978-3-319-38754-3 978-3-319-38756-7. DOI: 10.1007/978-3-319-38756-7_4. URL: http://link.springer.com/10.1007/978-3-319-38756-7_4 (besucht am 05. 05. 2022).

[Han17] Martin Hankel. „Unterwegs lernen zu laufen: Smarte Produkte und Lösungen explorativ und agil entwickeln“. In: Thomas Schulz. *Industrie 4.0 : Potenziale erkennen und umsetzen*. 1. Aufl.

Würzburg: Vogel Business Media, 2017, S. 127–150.
ISBN: 978-3-8343-6228-5.

[Has22] HashiCorp. *Nomad*. HashiCorp, 23. Aug. 2022. URL: https://github.com/hashicorp/nomad (besucht am 23. 08. 2022).

[Hin+11] Benjamin Hindman u. a. „Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center“. In: 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11). Boston, MA, 2011, S. 14.

[Hin+18] Christoph Hinze u. a. „Towards Real-Time Capable Simulations with a Containerized Simulation Environment“. In: *2018 25th International Conference on Mechatronics and Machine Vision in Practice (M2VIP)*. 2018 25th International Conference on Mechatronics and Machine Vision in Practice (M2VIP). Stuttgart: IEEE, Nov. 2018, S. 1–6. ISBN: 978-1-5386-7544-1. DOI: 10.1109/M2VIP.2018.8600827. URL: https://ieeexplore.ieee.org/document/8600827/ (besucht am 22. 05. 2022).

[Hin+19] Christoph Hinze u. a. „Control Architecture for Industrial Robotics based on Container Virtualization“. In: *Tagungsband des 4. Kongresses Montage Handhabung Industrieroboter*. Hrsg. von Thorsten Schüppstuhl, Kirsten Tracht und Jürgen Roßmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2019, S. 64–73. ISBN: 978-3-662-59316-5 978-3-662-59317-2. DOI: 10.1007/978-3-662-59317-2. URL: http://link.springer.com/10.1007/978-3-662-59317-2 (besucht am 05. 05. 2022).

[Hun+22] Min-Hsiung Hung u. a. „A Novel Implementation Framework of Digital Twins for Intelligent Manufacturing Based on Container Technology and Cloud Manufacturing Services“. In: *IEEE Transactions on Automation Science and Engineering* (2022), S. 1–17. ISSN: 1545-5955, 1558-3783. DOI: 10.1109/TASE.2022.3143832. URL: https://ieeexplore.ieee.org/document/9697094/ (besucht am 05. 05. 2022).

[Ior+18] Michaela Iorga u. a. *Fog Computing Conceptual Model*. NIST SP 500-325. Gaithersburg, MD: National Institute of Standards and Technology, März 2018, NIST SP 500–325. DOI: 10.6028/NIST.SP.500-325. URL:

http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.500-325.pdf (besucht am 10. 05. 2022).

[Jaw+19] Isam Mashhour Al Jawarneh u. a. „Container Orchestration Engines: A Thorough Functional and Performance Comparison“.
In: *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*. ICC 2019 - 2019 IEEE International Conference on Communications (ICC). Shanghai, China: IEEE, Mai 2019, S. 1–6. ISBN: 978-1-5386-8088-9.
DOI: 10.1109/ICC.2019.8762053.
URL: https://ieeexplore.ieee.org/document/8762053/ (besucht am 06. 05. 2022).

[Kem+21] Florian Kempf u. a. „AI-In-Orbit-Factory – AI Approaches for Adaptive Robotic in-Orbit Manufacturing of Modular Satellites“. In:
72nd International Astronautical Congress (IAC). Dubai: IAF, 29.10.21.

[Kha17] Asif Khan. „Key Characteristics of a Container Orchestration Platform to Enable a Modern Application“.
In: *IEEE Cloud Computing* 4.5 (Sep. 2017), S. 42–48. ISSN: 2325-6095.
DOI: 10.1109/MCC.2017.4250933.
URL: http://ieeexplore.ieee.org/document/8125559/ (besucht am 20. 08. 2022).

[Kra17] Nane Kratzke. „About Microservices, Containers and Their Underestimated Impact on Network Performance“. 14. Sep. 2017.
arXiv: 1710.04049 [cs]. URL:
http://arxiv.org/abs/1710.04049 (besucht am 06. 05. 2022).

[Kri+18] Werner Kritzinger u. a. „Digital Twin in Manufacturing: A Categorical Literature Review and Classification“.
In: *IFAC-PapersOnLine* 51.11 (2018), S. 1016–1022. ISSN: 24058963.
DOI: 10.1016/j.ifacol.2018.08.474.
URL: https://linkinghub.elsevier.com/retrieve/pii/S2405896318316021 (besucht am 05. 05. 2022).

[KS17] Zhanibek Kozhirbayev und Richard O. Sinnott.
„A Performance Comparison of Container-Based Technologies for the Cloud“.
In: *Future Generation Computer Systems* 68 (März 2017), S. 175–182.
ISSN: 0167739X. DOI: 10.1016/j.future.2016.08.025.
URL: https://linkinghub.elsevier.com/retrieve/pii/S0167739X16303041 (besucht am 10. 05. 2022).

[KSB17] Gregory M. Kurtzer, Vanessa Sochat und Michael W. Bauer.
„Singularity: Scientific Containers for Mobility of Compute“.
In: *PLOS ONE* 12.5 (11. Mai 2017). Hrsg. von Attila Gursoy, e0177459.
ISSN: 1932-6203. DOI: 10.1371/journal.pone.0177459.
URL: https://dx.plos.org/10.1371/journal.pone.0177459
(besucht am 22. 08. 2022).

[Kub22] Kubernetes. *DNS for Services and Pods*. Kubernetes. 21. Juli 2022.
URL: https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/ (besucht am 21. 08. 2022).

[LJ19] Lubos Mercl und Jakub Pavlik.
„The Comparison of Container Orchestrators“.
In: *Third International Congress on Information and Communication Technology: ICICT 2018, London*. Hrsg. von Xin-She Yang u. a. Bd. 797.
Advances in Intelligent Systems and Computing.
Singapore: Springer Singapore, 2019, S. 677–685.
ISBN: 9789811311642 9789811311659.
DOI: 10.1007/978-981-13-1165-9.
URL: http://link.springer.com/10.1007/978-981-13-1165-9
(besucht am 20. 08. 2022).

[Mic22] Microsoft.
*Publisher-Subscriber Pattern - Azure Architecture Center | Microsoft Docs*.
2022. URL: https://docs.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber
(besucht am 28. 08. 2022).

[Mon22] MongoDB, Inc. *MongoDB Sharding*. MongoDB. 2022.
URL: https://www.mongodb.com/basics/sharding (besucht am 13. 09. 2022).

[MSF16] Alexandru Moga, Thanikesavan Sivanthi und Carsten Franke. „OS-level Virtualization for Industrial Automation Systems: Are We There Yet?“
In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*.
SAC 2016: Symposium on Applied Computing.
Pisa Italy: ACM, 4. Apr. 2016, S. 1838–1843. ISBN: 978-1-4503-3739-7.
DOI: 10.1145/2851613.2851737.
URL: https://dl.acm.org/doi/10.1145/2851613.2851737
(besucht am 09. 06. 2022).

[NE22] Oliver Niggemann und Miriam Elmers. *Künstliche Intelligenz in Produktion Und Maschinenbau : Hintergründe, Anwendungsszenarien, Expertentipps*. Berlin: VDE Verlag, 2022. ISBN: 978-3-8007-5495-3. URL: https://d-nb.info/1232937282/04.

[Nis+22] Anil Nish u. a. *Microservices Architecture*. Microservices Architecture | Microsoft Docs. 13.4.22. URL: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/microservices-architecture (besucht am 05. 07. 2022).

[Ope20] Open Container Initiative. *About the Open Container Initiative - Open Container Initiative*. 2020. URL: https://opencontainers.org/about/overview/ (besucht am 20. 08. 2022).

[Pah15] Claus Pahl. „Containerization and the PaaS Cloud“. In: *IEEE Cloud Computing* 2.3 (Mai 2015), S. 24–31. ISSN: 2325-6095. DOI: 10.1109/MCC.2015.51. URL: http://ieeexplore.ieee.org/document/7158965/ (besucht am 06. 05. 2022).

[Pan+19] Yao Pan u. a. „A Performance Comparison of Cloud-Based Container Orchestration Tools“. In: *2019 IEEE International Conference on Big Knowledge (ICBK)*. 2019 IEEE International Conference on Big Knowledge (ICBK). Beijing, China: IEEE, Nov. 2019, S. 191–198. ISBN: 978-1-72814-607-2. DOI: 10.1109/ICBK.2019.00033. URL: https://ieeexplore.ieee.org/document/8944745/ (besucht am 22. 08. 2022).

[Por21] Portainer. *Install Portainer with Docker Swarm on Linux*. 2021. URL: https://docs.portainer.io/v/ce-2.6/start/install/server/swarm/linux (besucht am 07. 09. 2022).

[Red20] Red Hat, Inc. *What Is a Hypervisor?* 1. Okt. 2020. URL: https://www.redhat.com/en/topics/virtualization/what-is-a-hypervisor (besucht am 17. 09. 2022).

[SM11] Gökce Sargut und Rita Gunther McGrath.
„Learning To Live With Complexity“.
In: *Harvard business review* 89.9 (2011), S. 68–76.

[Som+18] P. Sommer u. a.
„Message-Oriented Middleware for Industrial Production Systems“.
In: *2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)*. 2018 IEEE 14th International Conference on Automation Science and Engineering (CASE).
Munich, Germany: IEEE, Aug. 2018, S. 1217–1223.
ISBN: 978-1-5386-3593-3. DOI: 10.1109/COASE.2018.8560493.
URL: https://ieeexplore.ieee.org/document/8560493/
(besucht am 25. 08. 2022).

[Sta+20] Rainer Stark u. a. „WiGeP-Positionspapier: „Digitaler Zwilling““. In:
*Zeitschrift für wirtschaftlichen Fabrikbetrieb* 115.s1 (7. Apr. 2020), S. 47–50.
ISSN: 0947-0085, 2511-0896. DOI: 10.3139/104.112311.
URL: https://www.degruyter.com/document/doi/10.3139/104.112311/html (besucht am 04. 10. 2022).

[Str+20] Václav Struhár u. a. „Real-Time Containers: A Survey“. In:
2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020).
Sydney: Dagstuhl, 2020.

[Sus20] Susan Moore. *Gartner Forecasts Strong Revenue Growth for Global Container Management Software and Services Through 2024*. Gartner. 25. Juni 2020.
URL: https://www.gartner.com/en/newsroom/press-releases/2020-06-25-gartner-forecasts-strong-revenue-growth-for-global-co (besucht am 01. 10. 2022).

[TMV18] Timur Tasci, Jan Melcher und Alexander Verl.
„A Container-based Architecture for Real-Time Control Applications“.
In: *2018 IEEE International Conference on Engineering, Technology and Innovation (ICE/ITMC)*. 2018 IEEE International Conference on Engineering, Technology and Innovation (ICE/ITMC). Stuttgart: IEEE, Juni 2018.
ISBN: 978-1-5386-1469-3. DOI: 10.1109/ICE.2018.8436369.
URL: https://ieeexplore.ieee.org/document/8436369/
(besucht am 05. 05. 2022).

[VFN20] Birgit Vogel-Heuser, Juliane Fischer und Eva-Maria Neumann.
„Softwaremodularität als Voraussetzung für autonome Systeme“.
In: *Handbuch Industrie 4.0*. Hrsg. von Michael ten Hompel,

Birgit Vogel-Heuser und Thomas Bauernhansl. Springer Reference Technik. Berlin, Heidelberg: Springer Berlin Heidelberg, 2020, S. 1–26. ISBN: 978-3-662-45537-1. DOI: 10.1007/978-3-662-45537-1_134-1. URL: http://link.springer.com/10.1007/978-3-662-45537-1_134-1 (besucht am 05. 05. 2022).

[Wat20] Stephen Watts. *The State of Containers Today: A Report Summary*. BMC Blogs. 13. Aug. 2020. URL: https://www.bmc.com/blogs/state-of-containers/ (besucht am 01. 10. 2022).

[Yon+19] Jiang Yongguo u. a. „Message-Oriented Middleware: A Review“. In: *2019 5th International Conference on Big Data Computing and Communications (BIGCOM)*. 2019 5th International Conference on Big Data Computing and Communications (BIGCOM). QingDao, China: IEEE, Aug. 2019, S. 88–97. ISBN: 978-1-72814-024-7. DOI: 10.1109/BIGCOM.2019.00023. URL: https://ieeexplore.ieee.org/document/8905013/ (besucht am 25. 08. 2022).

[Zer22a] ZeroMQ. *Get Started*. ZeroMQ. 2022. URL: https://zeromq.org/get-started/ (besucht am 18. 08. 2022).

[Zer22b] ZeroMQ. *Tests on Linux Real-Time Kernel - Zeromq*. 2022. URL: http://wiki.zeromq.org/results:rt-tests-v031 (besucht am 21. 08. 2022).