Institut für Textiltechnik und Lehrstuhl für Textilmaschinenbau

RWTH AACHEN UNIVERSITY

Diese Arbeit wurde vorgelegt am

Institut für Textiltechnik der RWTH Aachen University

Univ.-Prof. Prof. h.c. (MGU)
Dr.-Ing. Dipl.-Wirt. Ing. Thomas Gries

Deep Learning zur industriellen Qualitätsprüfung:

Entwicklung eines Plug-and-Play Bildverarbeitungssystems

Vorgelegt als:	Bachelorarbeit
Von:	Patrick Kunz
Matr.-Nr.	356018

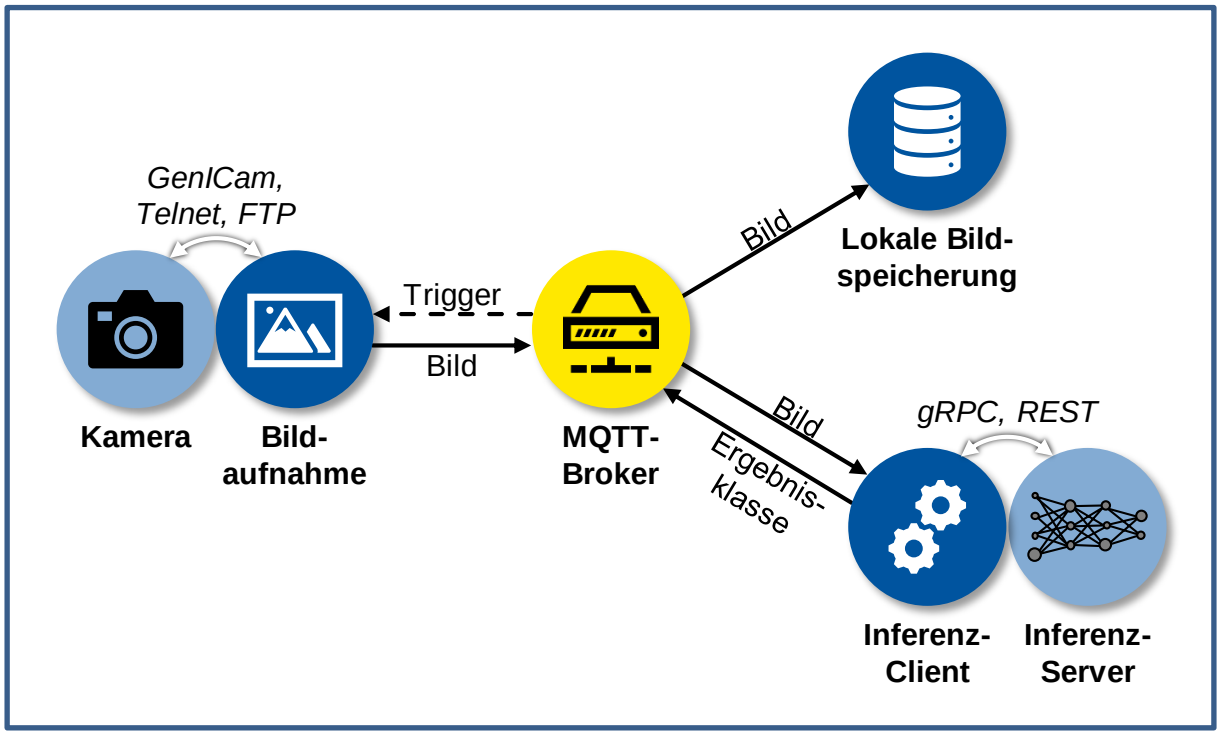
1. Prüfer: Univ.-Prof. Prof. h.c. (MGU) Dr.-Ing. Dipl.-Wirt. Ing. Thomas Gries

2. Prüfer: Dr.-Ing. Dieter Veit

Ansprechpartner/in: Kai Müller, M. Sc.

Aachen, Juni 2020

Institut für Textiltechnik und Lehrstuhl für Textilmaschinenbau
RWTH AACHEN UNIVERSITY



2020

Bachelorarbeit

Deep Learning zur industriellen Qualitätsprüfung: Entwicklung eines Plug-and-Play Bildverarbeitungssystems

Patrick Kunz

Kurzfassung

Ziel der Arbeit:

Das Ziel der Abschlussarbeit ist die Entwicklung einer robusten und anwenderfreundlichen Software für ein System zur industriellen Bildverarbeitung, das Methoden des Deep Learnings anwendet. Der Nutzer dieser Software wird aufgrund der Plug-and-Play Fähigkeit und der standardisierten Schnittstellen in die Lage versetzt, ein Bildverarbeitungssystem schnell und einfach in Betrieb zu nehmen. Die Systemsoftware basiert ausschließlich auf lizenzgebührenfreien Softwareprodukten.

Lösungsweg:

Zur Erarbeitung des Gesamtsystems werden relevante Standards, Schnittstellen und Softwarelösungen recherchiert und vorgestellt. Durch die Einteilung des Systems in Teilprozesse werden funktionale Anforderungen an die Software abgeleitet und in der Entwicklung mit den allgemeinen Anforderungen in einer Systemarchitektur umgesetzt. Die Implementierung und anschließende Validierung erfolgt in der Modellproduktion für textile Armbänder am Digital Capability Center Aachen.

Zentrale Ergebnisse:

Das zentrale Ergebnis ist eine Gesamtprozessübersicht und eine Microservice-Architektur, mit deren Hilfe sich ein industrielles Bildverarbeitungssystem nur durch Konfiguration der Kamera und Eingabe der Umgebungsvariablen softwareseitig in Betrieb nehmen lässt. Derzeit werden dabei Kameras des GenICam Standards mit GigE Vision Schnittstelle und Cognex Kameras unterstützt. Durch die offene Architektur wird eine Basisplattform für die Entwicklung weiterer Microservices und Folgeprozesse im Kontext der industriellen Bildverarbeitung geschaffen.

Schlagwörter: Industrielle Bildverarbeitung, Qualitätskontrolle, Deep Learning, Microservice-Architektur, MQTT

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungs- und Formelverzeichnis

Abb.	Abbildung
AIA	Automated Imaging Association
API	Application Programming Interface (Programmierschnittstelle)
ASCII	American Standard Code for Information Interchange
AutoML	automatisiertes maschinelles Lernen / automated Machine Learning
CMVU	China Machine Vision Industry Union
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DCC	Digital Capability Center
DL	Deep Learning
EMVA	European Machine Vision Association
engl.	englisch
ERP	Enterprise-Resource-Planning
etc.	et cetera
FPGA	Field Programmable Gate Array
FTP	File Transfer Protocol
Gb	Gigabit
Gb/s	Gigabit pro Sekunde
GenICam	Generic Interface for Cameras
GPU	Graphics Processing Unit
gRPC	google Remote Procedure Calls
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
i. A. a.	in Anlehnung an
IBV	industrielle Bildverarbeitung
IIDC	Instrumentation & Industrial Digital Camera
IIoT	Industrial Internet of Things
IoT	Internet of Things

IT	Informationstechnologie / Information-Technology
JIIA	Japan Industrial Imaging Association
JSON	JavaScript Object Notation
KNN	Künstliches Neuronales Netz
M2M	Machine-to-Machine
MB	Megabyte
MB/s	Megabyte pro Sekunde
ML	Maschinelles Lernen / Machine Learning
MQTT	Message Queuing Telemetry Transport
OPC UA	Open Platform Communications Unified Architecture
OCI	Open Container Initiative
OT	operative Technologien / Operational-Technology
PFNC	Pixel Format Naming Convention
PoCL	Power over Camera Link
PoE	Power over Ethernet
Protobuf	Protocol Buffer
QoS	Quality of Service
REST	Representational State Transfer
ROI	Region of Interest
RPC	Remote Procedure Call
s.	siehe
SDK	Software Development Kit
SFNC	Standard Feature Naming Convention
Tab.	Tabelle
Telnet	Teletype Network
TL	Transport Layer
TPU	Tensor Processing Unit
VDMA	Verband Deutscher Maschinen- und Anlagenbau e. V.
VM	virtuelle Maschine
VPU	Vision Processing Unit

1 Einleitung und Zielsetzung

In diesem Kapitel wird zunächst die Problemstellung im Gesamtkontext eingeordnet (s. Kapitel 1.1) und das Ziel der Bachelorarbeit definiert (s. Kapitel 1.2). Zuletzt wird die Vorgehensweise zur Lösung der Fragestellung dargestellt (s. Kapitel 1.3).

1.1 Kontext der wissenschaftlichen Fragestellung

Die industrielle Bildverarbeitung (IBV) ermöglicht das Erfassen und Verarbeiten umfangreicher Daten aus Prozessen und von Produkten. Das Ziel ist stets Transparenz über die Qualität zu erlangen, um entweder automatisch oder durch manuelles Eingreifen eines Operators die Qualität des Prozesses beziehungsweise des Produkts zu verbessern. Die automatisierte visuelle Qualitätsprüfung bietet die Möglichkeit, Vollprüfungen in der industriellen Produktion unter geringen laufenden Kosten umzusetzen. Im Vergleich zu einer stichprobenartigen Qualitätsüberwachung wird hierdurch die Reaktionszeit verkürzt und somit Ausschusskosten verringert. Im Gegensatz zu konventionellen Bildverarbeitungsmethoden sind mit Deep Learning nun auch variierende und komplexe Prüfaufgaben, die bisher dem menschlichen Sehen vorbehalten waren, wirtschaftlich automatisierbar.

Durch den Einsatz von Deep Learning ergeben sich aber auch Änderungen am Gesamtprozess. Neue Teilprozesse wie das Training des Künstlichen Neuronalen Netzes (KNN) wurden wissenschaftlich bereits ausreichend erarbeitet. Jedoch mangelt es bisher an systematischen Prozessbeschreibungen, die diese in den Gesamtprozess zur IBV integrieren und beschreiben.

Während in der Industrie das Potential durch Deep Learning Methoden bereits erkannt wurde und die neue Technologie schrittweise in kommerzielle Softwareprodukte integriert wird, existiert bisher keine ganzheitliche Lösung, die lizenzgebührenfrei verfügbar ist. Eine solche Lösung würde sich insbesondere für das Prototyping und den Einsatz in kleinen und mittelständischen Unternehmen eignen, denn die Einführung und Nutzung kommerzieller Software ist aufgrund der Lizenzgebühren und der Systemintegration mit hohen Kosten verbunden.

1.2 Zielsetzung

Das Ziel der Abschlussarbeit ist die Entwicklung einer robusten und anwenderfreundlichen Software für ein System zur IBV, das Methoden des Deep Learnings anwendet. Der Nutzer dieser Software wird aufgrund der Plug-and-Play Fähigkeit

und der standardisierten Schnittstellen in die Lage versetzt, ein IBV-System schnell und einfach in Betrieb zu nehmen. Die Systemsoftware basiert ausschließlich auf lizenzgebührenfreien Softwareprodukten. Damit ergeben sich konkret folgende zentrale Forschungsfragen:

1. Wie lässt sich der Gesamtprozess beim Einsatz von Deep Learning beschreiben und welche Anforderungen ergeben sich hieraus an die Systemsoftware?
2. Welche Standards, Kameraschnittstellen und lizenzgebührenfreien Softwarelösungen können verwendet werden?
3. Wie kann eine Software bereitgestellt werden, die möglichst unabhängig von der eingesetzten Hardware sowie der Folgeprozesse einen robusten und anwenderfreundlichen Einsatz eines auf Deep Learning-basierenden IBV-Systems ermöglicht?

Die detaillierte Betrachtung von Hardwarekomponenten des IBV-Systems wie Kamera, Beleuchtung oder Prozessor ist nicht Teil dieser Bachelorarbeit. Kompatible Kameras werden auf die Hersteller Cognex Corporation, Natick, USA, und Allied Vision Technologies GmbH, Stadtroda, mit GigE Vision Standard eingegrenzt. Das Training des Künstlichen Neuronalen Netz (KNN) wird nur grundlegend anhand beispielhafter Softwareprodukte beschrieben und nicht durch die zu entwickelnde Software selbst unterstützt. Der Inferenz-Server mit dem trainierten KNN wird für die Arbeit bereitgestellt.

1.3 Vorgehensweise

Zu Beginn der Bachelorarbeit werden begriffliche Definitionen und konzeptionelle Grundlagen beschrieben, um ein Grundverständnis über die Themen zu schaffen (Kapitel 2). Im Kapitel zum Stand der Technik wird auf Standards und Schnittstellen, die für die Entwicklung eines solchen Systems relevant sind, eingegangen (Kapitel 3). Anschließend wird der Gesamtprozess erarbeitet, um daraus neben den allgemeinen auch die funktionalen Anforderungen an die Software abzuleiten (Kapitel 4). Basierend auf den Anforderungen werden Softwarelösungen recherchiert und vorgestellt (Kapitel 5.1). Aufgrund der Vielzahl an verfügbaren Einzellösungen wird danach eine Eingrenzung vorgenommen (Kapitel 5.2). Um die ausgewählten Softwarelösungen zu verbinden, wird die Systemarchitektur und die Kommunikation zwischen den Teilprozessen betrachtet (Kapitel 5.3). Daraus wird auf verbleibende Entwicklungsaspekte geschlossen (Kapitel 5.4). Diese werden schließlich entwickelt, um die Systemsoftware zu erhalten (Kapitel 6). Zuletzt erfolgt eine Validierung der Software an zwei Anwendungsfällen in der Modellproduktion für smarte, textile Armbänder im Digital Capability Center Aachen (Kapitel

7). Das System wird zur Kontrolle der Druckqualität und des Nähprozesses implementiert. Im ersten Anwendungsfall wird eine Kamera von Allied Vision Technologies GmbH, Stadtroda, und im zweiten Anwendungsfall eine Kamera der Cognex Corporation, Natick, USA, eingesetzt. Die Erkenntnisse aus der Validierung des Systems werden am Ende zusammengefasst und ein Ausblick auf offene Forschungsfragen gegeben (Kapitel 8).

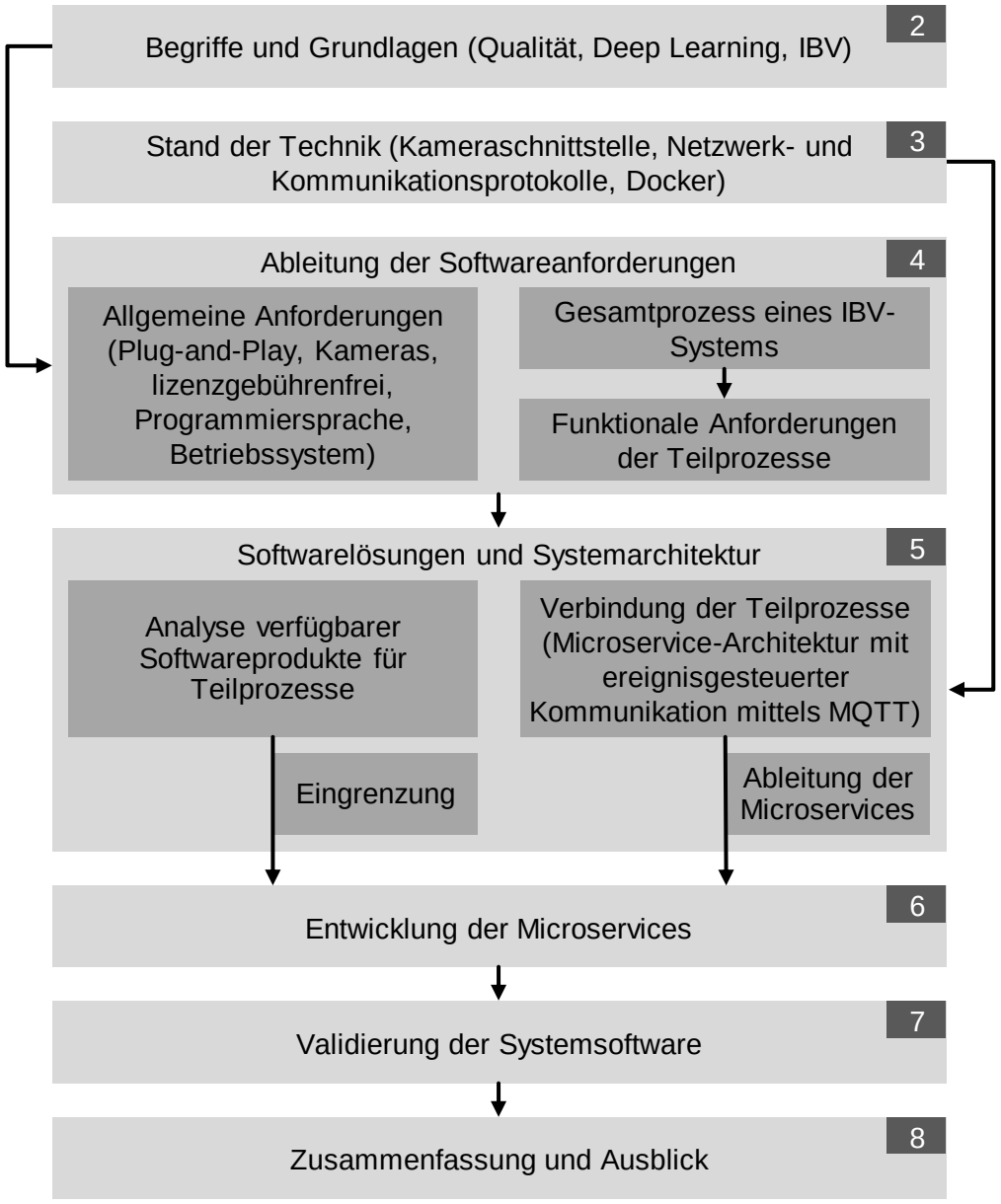


Abb. 1.1: Methodische Vorgehensweise

2 Begriffliche Definitionen und konzeptionelle Grundlagen

In den nachfolgenden Kapiteln werden Begriffe und Grundlagen beschrieben, die für das weitere Verständnis der Arbeit essenziell sind. Zu Beginn wird der Kontext des Vorhabens in der Qualitätssicherung eingeordnet (s. Kapitel 2.1). Anschließend werden die Begriffe Künstliche Intelligenz (s. Kapitel 2.2.1), Maschinelles Lernen (s. Kapitel 2.2.2) und Deep Learning (s. Kapitel 2.2.4) definiert und eingeordnet. Die Reihenfolge der Definitionen ist dabei von allgemein zu spezifisch. Des Weiteren wird ein grundlegendes Verständnis über industrielle Bildverarbeitungssysteme (s. Kapitel 2.3) geschaffen. Da die Softwarebereitstellung mittels Containervirtualisierung einige Vorteile mit sich bringt, wird zuletzt auf diese Methode eingegangen (s. Kapitel 2.4).

2.1 Qualitätssicherung

Die Qualität der Produkte und Prozesse eines Unternehmens sind entscheidend für dessen Erfolg. Fehlerhafte Produkte führen in nachgelagerten Produktionsschritten oder beim Endkunden zu hohen Folgekosten, um diese zu beseitigen oder das Produkt zu ersetzen. Neben den direkten Kosten droht auch ein gemindertes Ansehen durch Imageverlust. Zusätzlich stehen Unternehmen gerade in Hochlohnländern ohnehin unter gewaltigem Kostendruck. Um im globalen Wettbewerb agieren zu können, müssen Ausschussraten verringert und die Prozesseffizienz gesteigert werden. Auch die Erwartungen der Kunden an die Produktqualität steigen immer weiter an. Beispielsweise verringern sich die zulässigen Toleranzen fortschreitend. Dabei erhöht sich jedoch auch die Komplexität der Produkte. Die Variantenanzahl steigt, während sich die Losgrößen verringern. Um diesen Herausforderungen zu begegnen, wird die Qualitätssicherung weiter an Bedeutung gewinnen. [BB20; NB10] Qualität ist gemäß DIN EN ISO 9000 als der *„Grad, in dem ein Satz inhärenter Merkmale eines Objekts Anforderungen erfüllt“*, definiert [DIN EN ISO9000]. Die Qualitätssicherung lässt sich in drei Teilfunktionen unterteilen (s. Abb. 2.1) [Voi18]:

- Qualitätsplanung: Die Qualitätsplanung legt die Anforderungen an ein Produkt oder Prozess fest.
- Qualitätskontrolle (auch Qualitätsprüfung genannt): In der Qualitätskontrolle werden die Ist-Werte mit den Soll-Werten aus der Qualitätsplanung verglichen.
- Qualitätssteuerung (auch Qualitätslenkung oder -regelung genannt): Die Qualitätssteuerung minimiert durch entsprechende Maßnahmen am Prozess oder Produkt die Soll-Ist-Abweichung.

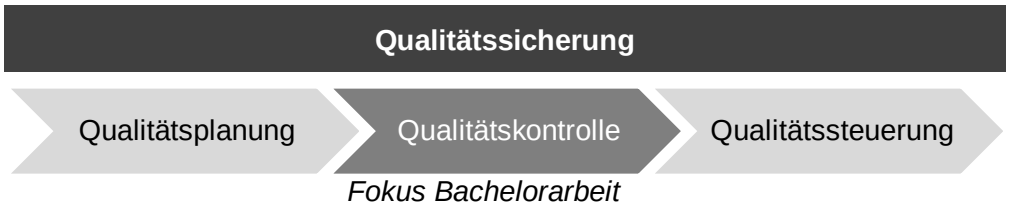


Abb. 2.1: Teilfunktionen der Qualitätssicherung

Im Rahmen dieser Bachelorarbeit liegt der Fokus auf der Qualitätskontrolle, um eine Aussage über die Qualität der gefertigten Produkte tätigen zu können und im nächsten Schritt eine Möglichkeit zu bieten, fehlerhafte Produkte frühzeitig auszusortieren oder Prozessparameter gezielt anzupassen. Automatisierte Bildverarbeitungssysteme bieten eine Möglichkeit, Produkte konstant und zuverlässig zu prüfen und Schlussfolgerungen daraus zu ziehen (s. Kapitel 2.3.2) [DSS11].

2.2 Begriffe und Einordnung zu Künstlicher Intelligenz, Maschinellem Lernen und Deep Learning

Nachfolgend wird zunächst der Begriff der Künstlichen Intelligenz (s. Kapitel 2.2.1) und das Maschinelle Lernen als Teilgebiet der Künstlichen Intelligenz (s. Kapitel 2.2.2) erklärt. Deep Learning (s. Kapitel 2.2.4) basiert auf Künstlichen Neuronalen Netzen (s. Kapitel 2.2.3) und ist ein Teilgebiet des Maschinellen Lernens. Zudem werden die Deep Learning-Prozesse Training und Inferenz beschrieben (s. Kapitel 2.2.4). In Kapitel 2.2.5 werden die Definitionen zusammenfassend in einem Venn Diagramm dargestellt. Das Ziel ist ein Grundverständnis über diese Fachbegriffe für diese Bachelorarbeit zu schaffen. Auf eine detaillierte Betrachtung wird in diesem Zusammenhang verzichtet und auf die Masterarbeit von Kevin Denker am ITA über „Architekturen und Augmentierungsmethoden von neuronalen Netzen zur robusteren Anwendung von lokal trainierten Modellen für die maschinelle Bilderkennung“ (2020) verwiesen.

2.2.1 Künstliche Intelligenz

Der Begriff der Künstlichen Intelligenz (KI) wird mit verschiedenen, oftmals sehr allgemeinen Definitionen beschrieben und orientiert sich im Rahmen dieser Arbeit an der Definition von Kreutzer und Sirrenberg (2019). Danach ist Künstliche Intelligenz der Versuch menschliche Lern- und Denkweisen auf Maschinen zu übertragen, die dadurch in der Lage sind, durch intelligente Methoden Probleme zu lösen, die mit menschlicher Intelligenz assoziiert sind. [KS19]

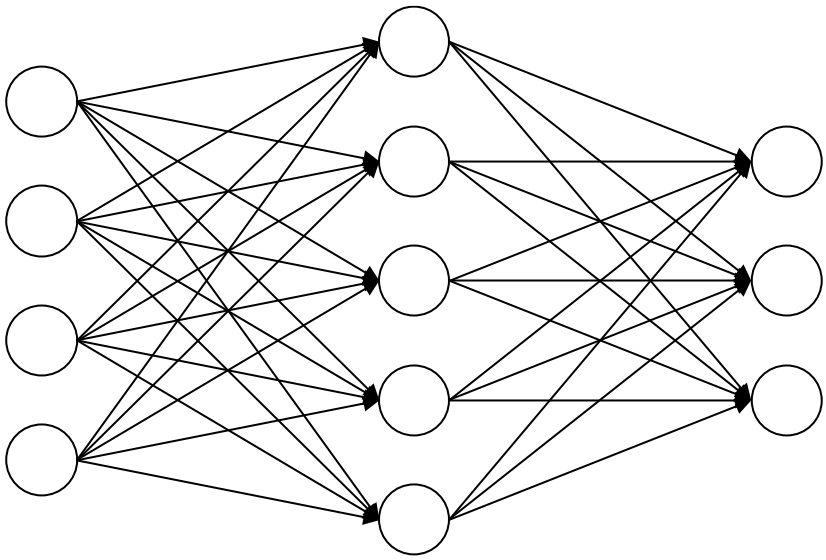
2.2.2 Maschinelles Lernen

Das Maschinelle Lernen (ML) (engl. Machine Learning) ist ein Teilgebiet der KI und beschreibt die Fähigkeit einer Maschine aus Erfahrungen künstlich Wissen durch die Erkennung von Mustern und Gesetzmäßigkeiten zu generieren [Lub16]. Die Maschine erarbeitet dabei aus den initial genutzten, allgemeinen Algorithmen zum Lernen, die durch den Programmierer vorgegeben wurden, selbstständig spezifische Regeln für die Problemlösung. Diese werden weiterentwickelt und ersetzen damit die vorangegangenen, ohne weiteres Eingreifen eines Programmierers zu erfordern. Beispiele hierfür sind Entscheidungsbäume. [KS19]

Es ist nicht trivial die Ansätze und das Wissen eines Menschen in Programmcode zu übertragen. ML liefert daher für Fragestellungen, die für den Menschen intuitiv und einfach zu lösen sind, wie das Lesen von Schrift oder das Erkennen von Objekten, einen erheblichen Mehrwert. [GBC16]

2.2.3 Aufbau von Künstlichen Neuronalen Netzen

Künstliche Neuronale Netze (KNN) sind aus einzelnen Schichten von künstlichen Neuronen aufgebaut. Diese Neuronen, die den Nervenzellen eines menschlichen Gehirns ähneln, können mehrere gewichtete Eingaben verarbeiten und werden je nach Ausgabe einer Aktivierungsfunktion ab einem gewissen Schwellwert aktiviert. Die Gewichte der Eingaben befinden sich auf den sogenannten Kanten zwischen den Neuronen und geben die Wahrscheinlichkeit an, dass diese Kante im neuronalen Netz genommen wird. [Roj93] Das Netz besteht zwischen Eingabe- und Ausgabeschicht aus mehreren Verarbeitungsschichten, den sogenannten verborgenen Schichten. In Abb. 2.2 wird der Aufbau eines KNN schematisch dargestellt. Die verborgene Schicht wurde dabei vereinfacht als einfache Schicht dargestellt. [KS19] Das neuronale Netz ist als eine mathematische Funktion vorstellbar, die einen Satz von Eingabewerten auf Ausgabewerte abbildet. Die Funktion wird dabei durch das Zusammensetzen vieler einfacherer Funktionen in den einzelnen Neuronen gebildet. [GBC16]



Eingangsschicht (engl. Input Layer) | Verborgene Schicht (engl. Hidden Layer) | Ausgangsschicht (engl. Output Layer)

Abb. 2.2: Schematische Darstellung eines typischen KNN aus vernetzten Neuronen

2.2.4 Deep Learning

Beim tiefgehenden Lernen (engl. Deep Learning) wird die Wahrnehmungsfunktion des Gehirns nachgeahmt. Dazu werden KNN verwendet. Die verborgene Schicht besteht dabei aus einer hohen Zahl an Ebenen. Aus der dadurch entstehenden Tiefe des Netzes leitet sich der Begriff „Deep“ ab. Diese Schichten nehmen riesige Mengen an Eingabedaten auf und verarbeiten sie über mehrere Schichten weiter, um ein Ergebnis zu liefern. [KS19] Damit können komplizierte Strukturen aus umfangreichen Daten dargestellt werden [VDD+18].

Deep Learning (DL) ist eine Art des Maschinellen Lernens, die eine größere Bandbreite an Datenressourcen verarbeiten kann, weniger Datenvorverarbeitung durch den Menschen erfordert und genauere Ergebnisse liefern kann als traditionelle Maschinelle Lernansätze, wie einfachere KNN oder Entscheidungsbäume [CKM18].

Im Deep Learning müssen zwei Prozesse unterschieden werden. Bevor Bilder in der Inferenz klassifiziert werden können, wird ein trainiertes KNN benötigt. Der Prozess kann somit in das Training und die Inferenz unterteilt werden (s. Abb. 2.3). Der Prozess wird hier am Beispiel des Überwachten Lernens (engl. Supervised Learning) dargestellt, wozu ein Mensch die Trainingsdaten vorher klassifiziert. Beim Unüberwachten Lernen (engl. Unsupervised Learning) würde dieser Schritt entfallen und sich das System eigene Klasen durch Clustering erzeugen. [Ert16]

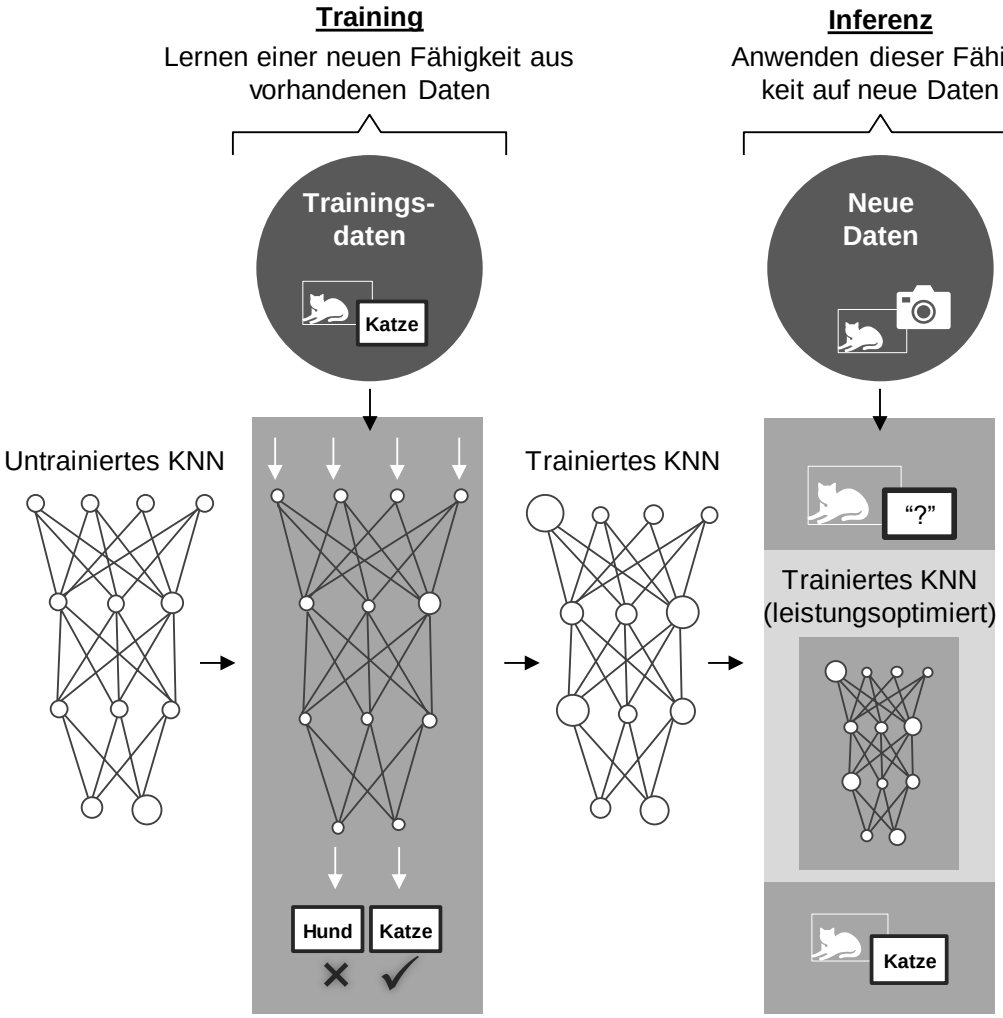


Abb. 2.3: Unterscheidung der Prozesse Training und Inferenz beim Deep Learning (i. A. a. [Cop16])

Training

Zu Beginn des Trainings existiert lediglich ein untrainiertes KNN, dessen Grundaufbau durch den Anwender vorgegeben wird. Für das Training wird ein Datensatz mit klassifizierten Bilddaten benötigt. Der Prozess, in dem die aufgenommenen Bilder durch einen menschlichen „Lehrer" einer Klasse zugeordnet werden, wird als Labeln bezeichnet. Die Begriffe Klasse und Label werden daher Synonym in dieser Arbeit verwendet. Die Bilddaten werden anschließend für eine Vorhersage ins KNN gegeben. Die durch das KNN vorhergesagte Klasse wird mit der vorher zugewiesenen Klasse verglichen. Je nach Ergebnis werden die Gewichte der Kanten, die Aktivierungsfunktionen der Neuronen und weitere Parameter angepasst (s. Kapitel 2.2.3). Am Ende des Trainings steht ein trainiertes KNN zur Verfügung. [Cop16; Ert16]

Inferenz

Unter Inferenz wird die Verwendung eines zuvor trainierten KNN verstanden, um damit neue Daten zu beurteilen. Das Ergebnis der Inferenz sind Wahrscheinlichkeiten, mit denen die Bilddaten den einzelnen Klassen zugeordnet werden können. [Cop16; Ert16]

Hybride Formen mit kontinuierlichem Lernen

Die Möglichkeit von kontinuierlichem Lernen wird durch das im Weiteren betrachtete System nicht ausgeschlossen, jedoch an dieser Stelle nicht näher betrachtet. Dabei wird das KNN stetig weiter trainiert und verbessert [Ett19; Liu17]. Im Folgenden werden die Prozesse als getrennt angesehen, sodass zunächst das Training vor der Inferenz erfolgen muss. Es kann jedoch jederzeit ein neues Netz trainiert und zur Inferenz genutzt werden. Dies geschieht jedoch nicht kontinuierlich oder automatisch.

2.2.5 Einordnung der Begriffe

Die Zusammenhänge zwischen KI, ML und DL lassen sich vereinfacht in einem Venn Diagramm darstellen (Abb. 2.4).

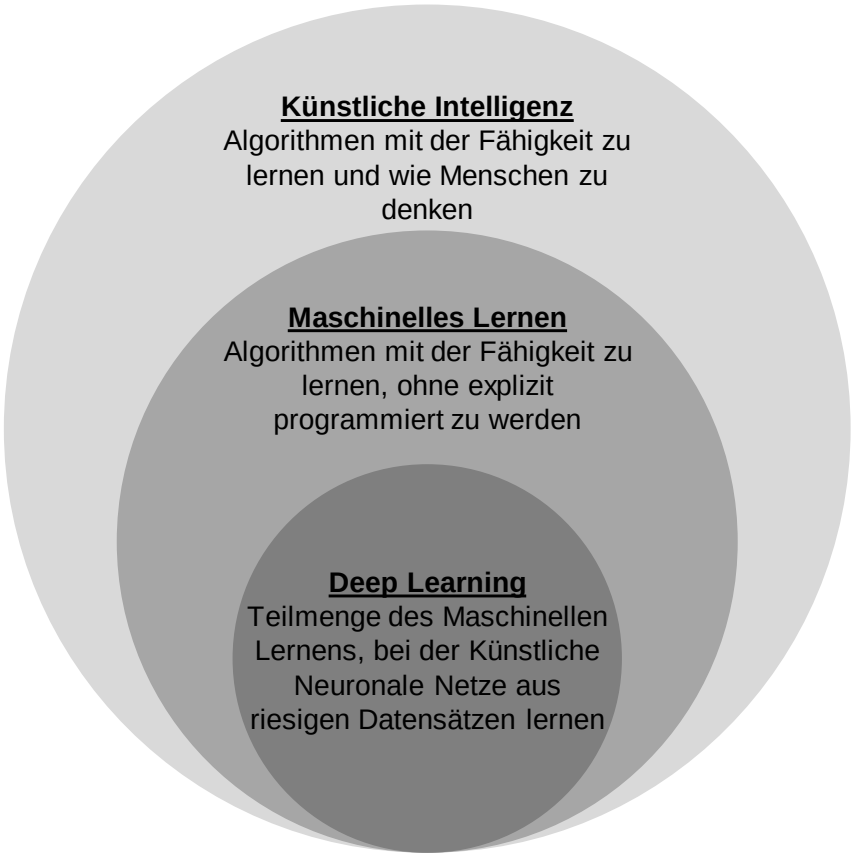
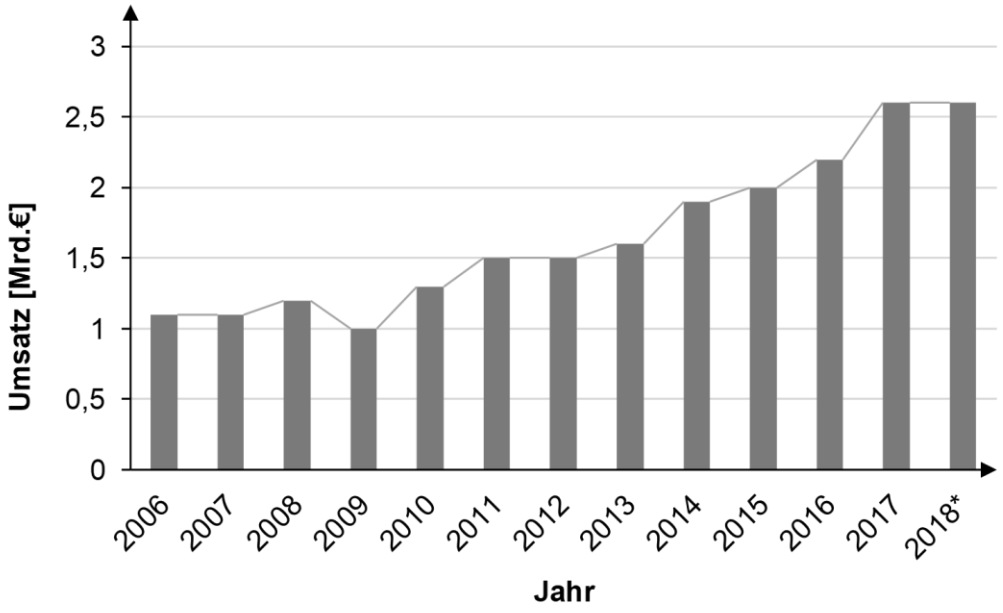


Abb. 2.4: Venn Diagramm zur Einordnung von Deep Learning (i. A. a. [Mat16])

2.3 Industrielle Bildverarbeitung

In der industriellen Bildverarbeitung (IBV) wird maschinelles Sehen verwendet, um mit optischen Technologien Aufgabenstellungen, die eigentlich mit Hilfe des menschlichen Sehsystems beurteilt werden, computergestützt zu lösen. Dabei wird besonders im englischen Sprachraum der Begriff Machine Vision als Synonym verwendet. Nachfolgend wird eine Arbeitsdefinition der IBV basierend auf den genannten Quellen aufgestellt, die auf den Anwendungsbereich der Sichtprüfung beschränkt und für diese Arbeit verwendet wird: IBV befasst sich mit der Auslegung von integrierten mechanisch-optisch-elektronischen Software-Systemen zur Untersuchung natürlicher Objekte und Materialien sowie von Herstellungsprozessen, um Abweichungen und Fehler zu erkennen und dadurch die Qualität, die Prozesseffizienz oder die Sicherheit von Produkten und Prozessen zu verbessern. [SUW18; GB04; BPF16]

IBV-Systeme gewinnen seit Jahren immer mehr an Bedeutung in der industriellen Produktion, was sich auch in den Umsätzen widerspiegelt (Abb. 2.5). Sie spielen eine Schlüsselrolle für Industrie 4.0 und die sogenannten „Smart Factories“ der Zukunft, in denen Menschen, Maschinen und Produkte miteinander kommunizieren und intelligente Produkte unter voller Prozesstransparenz individuell hergestellt werden können. [Vdm16]



* Vorhersage
Aktuellere Daten und verlässliche Aussichten liegen aufgrund der Corona-Pandemie zurzeit noch nicht vor.

Abb. 2.5: Entwicklung der deutschen Bildverarbeitungsindustrie von 2006 bis 2018 (Datenquelle: [Lit18; Wen18])

Im nachfolgenden Unterkapitel werden zunächst industrielle Anwendungsfelder definiert (s. Kapitel 2.3.1). Anschließend werden die Vorteile (s. Kapitel 2.3.2) und Grenzen (s. Kapitel 2.3.3) eines automatisierten Bildverarbeitungssystems aufgezeigt. Zuletzt wird auf die Hardwarekomponenten (s. Kapitel 2.3.4) und Verarbeitungsalgorithmen (s. Kapitel 2.3.5) eines IBV-Systems eingegangen.

2.3.1 Industrielle Anwendung

Mithilfe der IBV können sowohl Bauteile und Produkte identifiziert, sortiert und positioniert sowie deren Qualität geprüft als auch Prozesse überwacht und bei Bedarf nachgeregelt und optimiert werden. IBV-Systeme ermöglichen eine automatisierte zerstörungsfreie und berührungslose Qualitätskontrolle. [Spi12] Das Ergebnis daraus kann für mögliche Folgeschritte, wie das Darstellen auf Dashboards, Sortieren durch Aktoren oder Anpassen von Prozessparametern verwendet werden. Die Folgeprozesse können dabei sehr individuell sein. In Anlehnung an die VDI/VDE-Richtlinie 2628 [VDI/VDE2628] werden die Prüfaufgaben der Bildverarbeitung nach Demant, Streicher-Abel et al. (2011) und Steger, Ulrich et al. (2008) in sechs Klassen eingeordnet [DSS11; SUW18]:

- Objektidentifikation (durch charakteristische Merkmale wie Form, Maße, Farbe, etc. oder angebrachte Kodierungen)
- Vollständigkeitsprüfung (Sonderfall der Objektidentifikation)
- Lageerkennung (Position und Orientierung)
- Form- und Maßprüfung (geometrische Größen)
- Oberflächeninspektion (Rauheit, Farbe, Oberflächendefekte)
- Inspektion des Bauteilinneren (beispielsweise durch Röntgentechnik innere Strukturen, Defekte, Materialinhomogenitäten)

2.3.2 Vorteile im Vergleich zu menschlichen Prüfprozessen

Die IBV ist im Vergleich zu menschlichen Prüfprozessen auch dauerhaft ohne Verluste in der Erkennungsleistung objektiv und reproduzierbar einsetzbar. Die computergestützten Prozesse ermöglichen eine lückenlose und einfache Dokumentation der Prüfung. [BPF16] Zudem werden weniger personelle Ressourcen benötigt. Dadurch ergeben sich hohe Einsparpotenziale, ohne auf höchste Produktqualität durch eine 100%-Kontrolle verzichten zu müssen. Waren vorher beispielsweise aufgrund begrenzter Prüfkapazitäten nur Auswahlprüfungen von Zufallsstichproben möglich, sind durch Maschinelles Sehen auch höhere Prüfraten bis zu 100 %-Prüfungen wirtschaftlich umsetzbar. Eine 100%-Prüfung ist dabei sowohl als Sor-

tierprüfung, bei denen fehlerhafte Teile aussortiert werden, als auch Klassierprüfung, in der Teile in verschiedene Klassen eingeordnet werden, möglich [DIN55350].

2.3.3 Grenzen der industriellen Bildverarbeitung

Durch Deep-Learning-basierte Methoden sind Prüfprozesse auch mit unterschiedlichen Prüfmerkmalen und Produktvarianten effizient durchführbar (s. Kapitel 2.3.5). Um jedoch ein KNN mit einer Vorhersagegenauigkeit von über 99% zu trainieren, werden je Klasse mindestens 20 bis 100 Bilder benötigt. Die genaue Anzahl an benötigten Bildern hängt stark vom konkreten Anwendungsfall und den verwendeten Algorithmen ab. Die Grenze für den wirtschaftlichen Einsatz der IBV wird durch DL in Hinblick auf die Anzahl an Produktvarianten und Prüfmerkmalen zwar weiter in den Bereich verschoben, der bisher ausschließlich durch Menschen wirtschaftlich umsetzbar war, jedoch werden auch weiterhin nur Prozesse, die sich häufig wiederholen und eine hohe Taktrate aufweisen, wirtschaftlich umsetzbar sein. [Spi12] Ansonsten sind der Aufwand für das Training und die Kosten für die Einrichtung des IBV-Systems im Vergleich zu einem menschlichen Prüfprozess zu hoch.

Es gibt jedoch auch generelle Grenzen der IBV, die unabhängig vom Verarbeitungsalgorithmus auftreten. Reflektierende Oberflächen oder durchsichtige Materialien erschweren das Maschinelle Sehen. Häufig wechselnde Ausleuchtungsbedingungen des Prüfobjekts oder raue Industrieumgebungen (zum Beispiel mit Ölnebel) müssen durch passende Auslegung des Systems vermieden werden, um die Umgebungsbedingungen stabil zu halten. Wenn Merkmale nicht eindeutig sind und enge Beurteilungstoleranzen bestehen, kann bei der IBV sogenannter „Pseudo-Ausschuss“ entstehen. Dieser Ausschuss wurde fälschlicherweise als solcher klassifiziert. [Spi12; DSS11]

2.3.4 Komponenten eines industriellen Bildverarbeitungssystems

Der schematische Aufbau eines IBV-Systems ist in Abb. 2.6 anhand einer Fließproduktion dargestellt. Ähnliche Systeme lassen sich auch in der stationären Prüfung einsetzen, bei der das Prüfobjekt durch einen Mitarbeiter oder Roboter in den Prüfbereich eingelegt wird. Die Abbildung und die nachfolgende Einteilung basiert auf einer Literatur- und Internetrecherche [AEJ+18; Spi12]. Die Ergebnisse daraus wurden in Gesprächen mit Experten am DCC Aachen zu dem hier vorgestellten System weiterentwickelt. Ein IBV-System kann demnach in drei Teilbereiche gegliedert werden:

- Bildgewinnung
- Datenauswertung
- Ergebnisverwertung

In der Bildgewinnung wird das visuelle Abbild des Prüfobjekts erzeugt und damit die digitalen Daten bereitgestellt. Die erzeugten Bilddaten werden in der Datenauswertung vorverarbeitet und dann mithilfe von Bildverarbeitungsalgorithmen (s. Kapitel 2.3.5) analysiert. Abschließend können die daraus generierten Ergebnisse für verschiedene Folgeprozesse zum Darstellen, Speichern und Steuern verwendet werden. Der Fokus dieser Bachelorarbeit liegt auf der der Bildgewinnung und Datenauswertung. Die Prozesse der Ergebnisverwertung sind individuell auszulegen und nicht mehr Bestandteil des betrachteten Systems in dieser Arbeit.

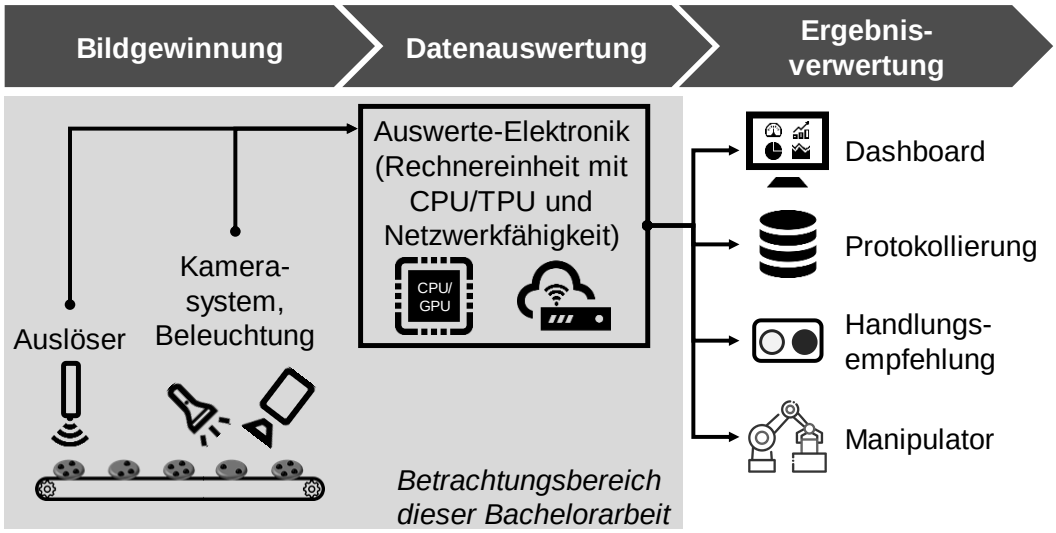


Abb. 2.6: Schematischer Aufbau eines industriellen Bildverarbeitungssystems (i. A. a. [Spi12; AEJ+18])

Nachfolgend werden die einzelnen Hardwarekomponenten, die in Abb. 2.6 im Betrachtungsbereich der Bachelorarbeit dargestellt sind, beschrieben:

Beleuchtung

Die Beleuchtung regt den Prüfling durch Licht an, so dass die emittierten Photonen durch Bildsensoren gemessen werden können. Die Beleuchtung ist so auszulegen, dass die Randbedingungen des Prüfprozesses stabil sind. Dazu kann der Prüfprozess durch Abdeckungen oder durch die Verwendung von Pulslicht (auch als Blitzlicht bezeichnet) unabhängiger von den äußeren Bedingungen gemacht werden. [Jah17]

Kamerasystem

Das Kamerasystem besteht aus einer Abbildungsoptik, einem oder drei Bildsensoren und einer Kamera-Elektronik. Mithilfe eines Objektivs wird die vom Prüfobjekt emittierte Strahlung auf den Bildsensor geleitet. Erst dadurch entsteht ein scharfes und kontrastreiches Bild. Es ist wichtig, dass die Abbildungsoptik reproduzierbar einstellbar und arretierbar ist, um nicht von äußeren Einflüssen wie Vibrationen beeinflusst zu werden. Der Bildsensor der Kamera misst anschließend die Anzahl der Photonen und deren Wellenlänge. Aus den gewonnen analogen Signalen des Bildsensors werden die Farben berechnet. Farbwerte werden dabei in einer Matrix von Bildpunkten, den sogenannten Pixeln, erfasst. Kameras enthalten entweder einen Bildsensor, der nach dem Bayer-Prinzip arbeitet, oder drei Sensoren, bei denen die einfallende Strahlung durch ein Prisma auf alle drei Sensoren geleitet wird. Bei drei Sensoren erfasst jeder Bildsensor einen Farbwert (rot, grün, blau) von jedem Pixel. Beim Bayer-Prinzip wird für jedes Pixel nur ein Farbwert erfasst, da nur ein Sensor in der Kamera verbaut ist. Die beiden übrigen Farbwerte werden aus umliegenden Pixeln interpoliert. Das entstandene Abbild des Prüfteils wird anschließend als digitales Abbild über Hardwareschnittstellen und Kommunikationsprotokolle an die Recheneinheit übertragen. [Mat17; Ste19b]

Im Rahmen dieser Arbeit werden ausschließlich Flächenkameras verwendet, die einzelne Bildaufnahmen machen. Sie eignen sich insbesondere für Einzelaufnahmen in diskontinuierlichen oder stationären Produktionsprozessen. In Prozessen mit kontinuierlichem Materialfluss haben Linienkameras gegenüber Flächenkameras den Vorteil, dass ein kontinuierlicher, überlappungsfreier Bilddatenstrom erzeugt wird. Durch spezielle Verarbeitungsalgorithmen können jedoch auch mit Flächenkameras solche Bilddatenströme erzeugt werden. Eine nähere Betrachtung der hierzu nötigen Algorithmen findet innerhalb dieser Arbeit nicht statt. Es sei auf die Ausführungen von Telljohann (2017) verwiesen. [Tel17]

Früher waren zur Umwandlung der analogen Werte des Bildsensors in digitale Farbwerte externe Framegrabber notwendig. Da dieser Prozess heute, bis auf Ausnahmefälle wie beispielsweise Hochgeschwindigkeitsaufnahmen, durch interne Framegrabber in der Kamera abgedeckt wird, wird das System in dieser Bachelorarbeit ohne separaten Framegrabber angenommen. [Ste19b]

Sogenannte intelligente Kameras haben durch ihre eingebettete Rechenleistung die Möglichkeit, die Bilddaten direkt auf der Kamera auszuwerten, sodass nicht mehr das Bild an eine externe Recheneinheit, sondern direkt das Ergebnis des Prüfprozesses übertragen wird. [Vdm18]

Die passende Auswahl des Kamerasystems und die exakte Abstimmung der Beleuchtung sind essenziell für den Erfolg oder Misserfolg eines Bildverarbeitungs-

systems. Denn nicht das reale Objekt selbst, sondern das visuelle Abbild wird geprüft. Daher hängen die Leistungsfähigkeit und Genauigkeit stark von der Qualität der Aufnahme ab. [Tel17] Für die Auswahl der Hardware wird an dieser Stelle auf die Masterarbeit von Michael Müller am ITA zum Thema „Industrielle Bildverarbeitung zur Qualitätskontrolle in Produktionslinien: Entwicklung einer Entscheidungslogik zur Anwendungsfallspezifischen Auswahl von Hard- und Software“ (2020) verwiesen. Hierin wurde die zugrunde liegende Theorie des Maschinellen Sehens dargestellt und eine Entscheidungslogik zur passenden Auswahl der Bildgewinnungshardware erarbeitet.

Trigger

Der Auslöser (engl. Trigger) dient dazu den Prüfprozess der IBV zum richtigen Zeitpunkt auszulösen. Dies ist insbesondere bei diskontinuierlichen Fließproduktionen wichtig, bei denen die Prüfteile den Inspektionsbereich in unregelmäßigen Abständen durchlaufen. Bei kontinuierlichen Produktionsprozessen, bei denen Produkte einen festen Abstand zueinander haben und die Bewegungsgeschwindigkeit definiert werden kann, können auch zeitliche Trigger in der Software, die nach einem festen Zeitintervall auslösen, genutzt werden. Als Hardware-Trigger können beispielsweise Abstandssensoren, kapazitive und induktive Sensoren oder manuelle Taster dienen. [Vdm18]

Auswerte-Elektronik

Die Auswerte-Elektronik umfasst alle elektronischen Komponenten und Anlagenteile, die aus dem visuellen, digitalen Abbild des Prüfobjekts Messwerte, Steuersignale, Qualitätsparameter oder ähnliches ermitteln und weiterleiten. [Spi12]

Zentraler Bestandteil ist dabei der Industrierechner, sofern keine intelligente Kamera verwendet wird. An den Rechner sind die bildgebenden Sensoren und je nach Systemauslegung auch der Trigger angeschlossen. Das Bild wird auf dem Rechner zunächst vorverarbeitet, um beispielsweise die Bildgröße anzupassen oder den Kontrast zu erhöhen. Die Bildvorverarbeitung wird als Preprocessing bezeichnet. Anschließend erfolgt je nach Art der Prüfaufgabe und gewählter Verarbeitungsmethode ein Algorithmus zur Analyse der Daten. Hierbei wird zwischen konventionellen Bildverarbeitungsalgorithmen und Methoden des Deep Learnings, auf die in Kapitel 2.3.5 näher eingegangen wird, unterschieden.

Je nach Umfang der Verarbeitung ist die Rechenleistung des Industrierechners zu wählen. Durch die Verwendung von Grafikprozessoren (engl. Graphics Processing Unit = GPU) können Berechnungsalgorithmen im Vergleich zu einer Central Processing Unit (CPU) parallelisiert werden. Sofern Deep Learning zur Bildauswertung verwendet wird, kann die Rechenleistung durch eine Tensor Processing Unit (TPU) erhöht werden. TPU wurden für die Verarbeitung von Matrizen in KNN durch

Google LLC, Mountain View, USA, optimiert. Eine Vision Processing Unit (VPU) bietet ähnliche Vorteile wie eine TPU. VPU sind Spezialchips, die von der Intel Corporation, Santa Clara, USA, speziell für die Bildverarbeitung entwickelt wurden. Ein Field Programmable Gate Array (FPGA) ist ein integrierter Schaltkreis, der für KI-Anwendungen verwendet werden kann, um eigene logische Schaltungen individuell umzusetzen. [Xie18] Die Entscheidung für oder gegen eine bestimmte Art von Prozessor hängt auch davon ab, welche Software für die Bildverarbeitung verwendet wird. Denn nicht jede Software unterstützt jede Art von Prozessor.

Basierend auf den Expertengesprächen am DCC Aachen wird durch die Auswerte-Elektronik auch die Netzwerkfähigkeit und Weiterleitung der Ergebnisse sichergestellt. Das Resultat aus der Auswertung kann in firmeneigenen Prozessmanagementsystemen, zur Darstellung auf Dashboards, zur direkten Beeinflussung von Prozessen oder zur Steuerung von Manipulatoren wie Robotern genutzt werden. Das Ziel ist stets die Qualität transparent zu machen, um entweder automatisch oder durch manuelles Eingreifen eines Operators die Qualität des Prozesses beziehungsweise des Produkts zu verbessern.

2.3.5 Bildverarbeitungsalgorithmen

In der IBV können verschiedene Algorithmen für die Analyse der Bilddaten verwendet werden, um eine Aussage über das untersuchte Objekt zu erhalten. Für diese Aussage müssen Merkmale im Bild erkannt und extrahiert werden, um sie anschließend zu beurteilen. Für dieses Vorgehen kann zwischen konventionellen Algorithmen und Methoden des Deep Learnings unterschieden werden.

Konventionelle Bildverarbeitungsalgorithmen

Unter konventionellen Bildverarbeitungsalgorithmen werden jene Algorithmen verstanden, bei denen der Nutzer spezifische Anweisungen zur Merkmalsextraktion und Problemlösung explizit definiert (s. Abb. 2.7a) [WSS16; WMZ+18]. Ein Algorithmus beschreibt in der Informatik eine endliche Folge von wohldefinierten, computerimplementierbaren Handlungsvorschriften, die typischerweise zur Lösung einer Klasse von Problemen oder zur Durchführung einer Berechnung dienen. [Rog02]

Aufgrund der exakten Merkmalsdefinition liefern konventionelle Methoden eindeutige, nachvollziehbare Ergebnisse. Sie erreichen insbesondere bei der Vermessung von Objekten, dem Barcodelesen und -identifizieren, bei der Prüfung der Farbtreue oder der Anwesenheitsprüfung zufriedenstellende Ergebnisse. [Cog18] Jedoch entsteht durch das manuelle Definieren und Programmieren der spezifischen Vorschriften für jeden Einsatzfall ein hoher Aufwand. Aufgrund dieser individuellen Auslegung lassen sich die Vorschriften nicht direkt auf neue oder andere Problemstellungen übertragen. [WSS16]

Methoden des Deep Learnings

Dagegen sind Künstliche Neuronale Netze beim Deep Learning in der Lage aufgrund von klassifizierten Datensätzen, eigene Merkmale in Form von Gewichtungsfunktionen und Neuronen zu identifizieren, nach denen sie anschließend Bilder in der Inferenz klassifizieren. Durch die Zuordnung zu verschiedenen Klassen kann dann die Erfüllung der Qualitätsanforderungen sichergestellt werden (s. Abb. 2.7c). [WSS16] Es sind auch hybride Ansätze, bei denen Bilder zunächst mit konventionellen Methoden vorverarbeitet und anschließend mit Hilfe eines KNN klassifiziert werden, möglich (s. Abb. 2.7b). [OCC+20]

Deep Learning kommt daher besonders dann zum Einsatz, wenn sehr komplexe Prüfungsaufgaben mit vielen oder nicht einfach zu beschreibenden Merkmalen vorliegen und viele Varianten durch tolerierte Schwankungen möglich sind. Die Vorteile können insbesondere bei der Anomaliedetektion von Oberflächen, der Objektklassifikation von komplexen oder variablen Objekten oder der Montageüberprüfung genutzt werden. [Cog18]

Abgrenzung unterschiedlicher Bildverarbeitungsvorgehen

Der zusammenfassende Überblick in Abb. 2.7 verdeutlicht die Unterschiede zwischen konventionellen Bildverarbeitungsalgorithmen und Methoden des Deep Learnings. Eine Entscheidungslogik zwischen konventionellen Algorithmen und Methoden des Deep Learnings zur Bildverarbeitung wurde durch Michael Müller am ITA in seiner bereits oben genannten Masterarbeit erarbeitet. Wie bereits in der Einleitung dieser Bachelorarbeit definiert, wird im Folgenden der Einsatz von Deep Learning zur Bildverarbeitung angenommen (vgl. Kapitel 1.2).

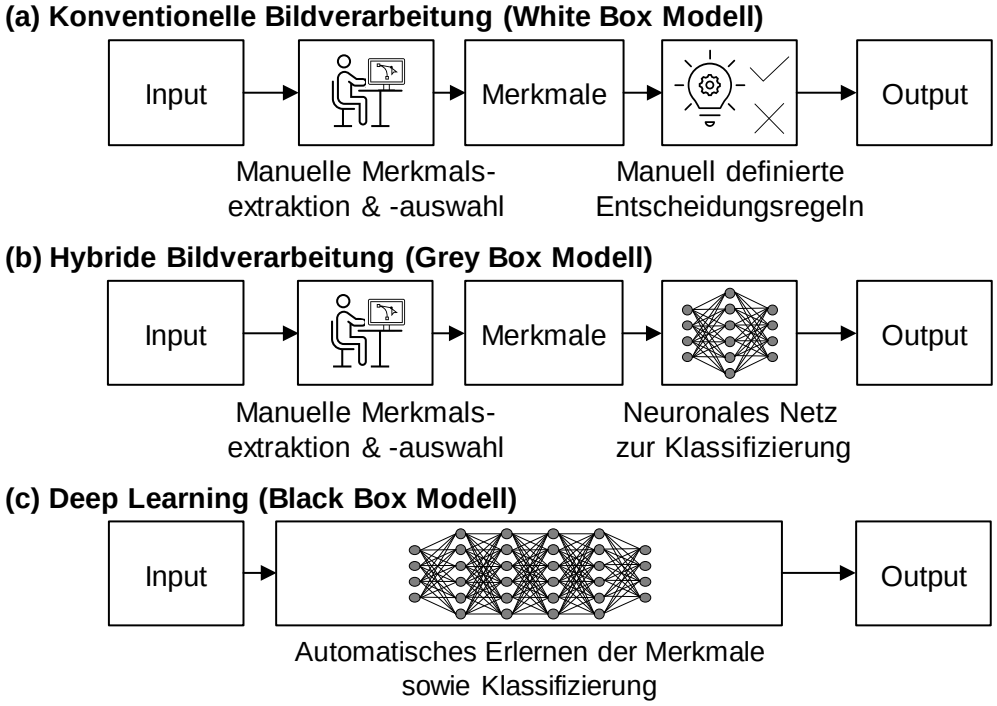


Abb. 2.7: Schematische Darstellung von konventioneller (a), hybrider (b) und Deep Learning-basierter (c) Bildverarbeitung (Bildquelle: [WMZ+18; Cog18])

2.4 Containervirtualisierung

Unterschiedliche Software-Umgebungen, wie beispielsweise unterschiedliche Versionen eines Softwarepakets innerhalb einer Anwendung, führen dazu, dass Anwendungen zunächst auf dem eigenen Computer und dessen Umgebung funktionieren, jedoch beim Ausführen auf einem anderen Computersystem Probleme auftreten. Die Containervirtualisierung entschärft dieses Problem zwischen der Entwicklungsumgebung, in der die Anwendung anfangs umgesetzt und getestet wird, und der Produktionsumgebung, in der das Programm bereitgestellt und ausgeführt wird (s. Abb. 2.8). Die Containervirtualisierung dient der einfachen Bereitstellung von Anwendungen durch Abstraktion von der konkreten Hardware. Container beinhalten dabei alle notwendigen Dateien, Programmbibliotheken und Konfigurationseinstellungen zur Ausführung der Applikation [Kof18].

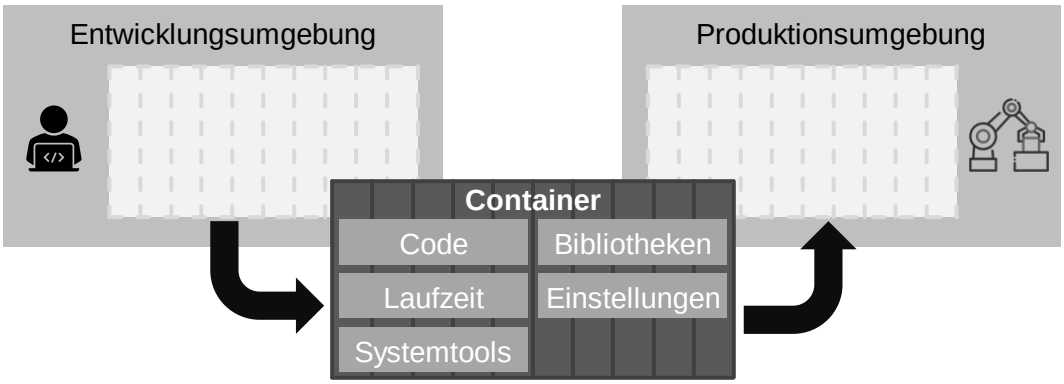


Abb. 2.8: Bereitstellung der Anwendung als Container aus der Entwicklungs- in die Produktionsumgebung

Mehrere Container können isoliert voneinander auf dem gleichen Betriebssystem laufen und teilen sich dabei den Betriebssystem-Kernel. Mithilfe einer Laufzeitumgebung können containerisierte Anwendungen überall und konsistent auf jeder Infrastruktur ausgeführt werden. Ein Beispiel hierzu ist die Docker Engine (s. Docker Kapitel 3.3). Im Container wird kein Betriebssystem installiert oder ausgeführt. Der Container greift stets auf den Host-Betriebssystem-Kernel zurück, was ihn von einer virtuellen Maschine (VM) abgrenzt. Bei der vollständigen Virtualisierung mit einer VM wird auch immer ein eigener Betriebssystem-Kernel installiert und gestartet. Der Boot-Vorgang hiervon entfällt daher bei der Containervirtualisierung. Container lassen sich bei Problemen schnell und einfach neustarten. [Bau17]

Die Containervirtualisierung benötigt aufgrund dieser Eigenschaften deutlich geringere Ressourcen und ist daher besonders für Einsatzzwecke mit begrenzter Rechenkapazität geeignet [Kof18]. Ein Container-Image, das Speicherabbild eines Containers, ist typischerweise mehrere zehn Megabytes (MB) groß, während eine VM Dutzende von Gigabytes (GB) benötigt [Doc20].

VM stellen eine Abstraktion von der physischen Hardware dar, die einen Server in mehrere umwandelt. Dazu wird ein Hypervisor benötigt, um mehrere virtuellen Maschinen auf einem Server auszuführen. Der Hypervisor dient somit der Abstraktion des simultanen Betriebs mehrerer Gastsysteme von der eigentlich vorhandenen Hardware und verteilt die Ressourcen für die einzelnen Systeme. [PG74] In Abb. 2.9 werden die Unterschiede zwischen Containern und VM verdeutlicht. [Doc20] Auch Kombinationen, in denen Container in VM ausgeführt werden, sind möglich.

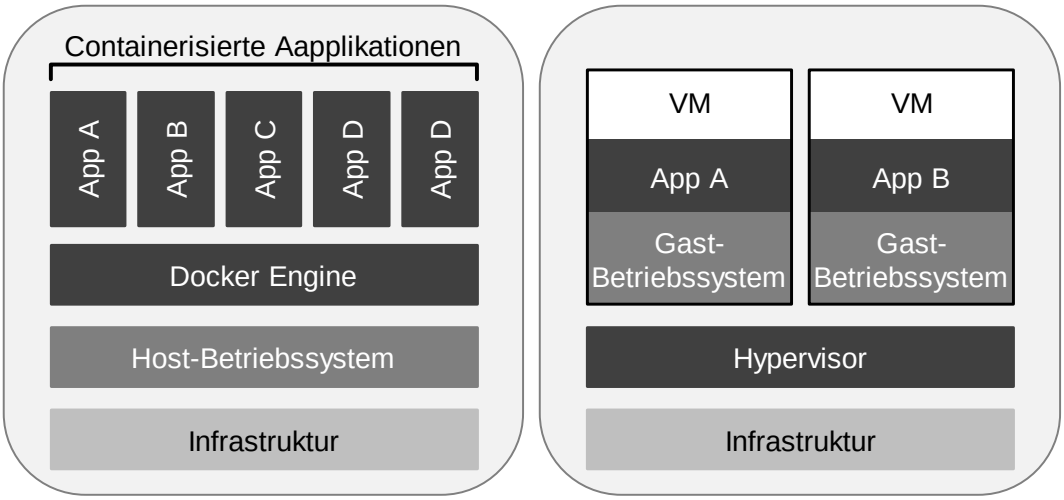


Abb. 2.9: Vergleich von containerisierten Anwendungen (links) mit virtuellen Maschinen (rechts) (Bildquelle: [Doc20])

Die Containervirtualisierung wird zur Modularisierung von Anwendungen in Microservices genutzt. Ein Microservice ist eine einzelne isolierte Anwendung, die eine bestimmte Funktion für ein Gesamtsystem erfüllt. Diese Microservices lassen sich bei Änderungen einfacher austauschen. Jedoch erhöht sich der reine Programmieraufwand und die Ressourcennutzung aufgrund der notwendigen Kommunikation zwischen den Microservices. Eine von der Programmiersprache unabhängige Kommunikation kann dabei beispielsweise über google Remote Procedure Call (gRPC) (s. Kapitel 3.2.3) oder Representational State Transfer (REST) (s. Kapitel 3.2.4) erfolgen. [Kof18] Die Nutzung von Microservices als Software-Architektur wird in Kapitel 5.3 näher betrachtet.

3 Stand der Technik

In diesem Kapitel wird zunächst der aktuelle Stand der Technik für die Kameraschnittstelle des IBV-Systems (s. Kapitel 3.1) und die später verwendeten Netzwerk- und Kommunikationsprotokolle (s. Kapitel 3.2) betrachtet. Zuletzt wird die Software Docker von Docker Inc., Palo Alto, USA, für die Containervirtualisierung vorgestellt (s. Kapitel 3.3)

3.1 Kameraschnittstellen-Standard

Aus dem Alltag sind Anschlussstandards zur Bildanzeige und Visualisierung wie HDMI und DisplayPort bekannt. Jedoch ermöglichen diese Standards nur das Übertragen von Videos ohne Kamerasteuerung. Für die IBV muss eine Kamera auch steuerbar sein, um Einstellungen vorzunehmen und das Bild auszulösen. In den letzten Jahren wurden Standards entwickelt, um die Anwendung von Bildverarbeitungssystemen zu erleichtern und damit die Kosten zu senken. Parallel existieren jedoch auch weiterhin herstellerspezifische Umsetzungen wie bei Kameras des Herstellers Cognex Corporation, Natick, USA, auf die in späteren Kapiteln noch eingegangen wird.

Die international bedeutendsten Verbände zur IBV haben sich 2009 zur sogenannten G3-Initiative zusammengeschlossen. Gegründet wurde die G3-Iniative gemeinsam von der amerikanischen Automated Imaging Association (AIA), der Japan Industrial Imaging Association (JIIA) und der European Machine Vision Association (EMVA). Später schlossen sich dieser Initiative auch die China Machine Vision Industry Union (CMVU) und der Verband Deutscher Maschinen- und Anlagenbau e. V. (VDMA) an. Damit sind in der G3-Initiative die größten Wirtschaftsregionen für die IBV mit ihren jeweiligen Verbänden vertreten. Das Ziel der Initiative ist es, Standards für IBV-Systeme zu definieren und weiterzuentwickeln. Jeder Standard wird dabei durch ein Mitglied der Initiative aktiv unterstützt und in der Weiterentwicklung überwacht. [AEJ+18]

Regelmäßig werden von dieser Initiative die am Markt verbreiteten und bestehenden Standards in einer gemeinsamen Publikation herausgegeben. Diese Empfehlungen spiegeln sich auch im Produktportfolio der Kamerahersteller wider. Das garantiert vor allem Kontinuität und Interoperabilität von Hardware für die Erstellung eigener Anwendungen. Daher wird auch in der Umsetzung der Software im Rahmen dieser Bachelorarbeit auf die von der G3-Initiative vorgeschlagenen Standards zurückgegriffen.

Die Kameraschnittstelle wird in diesen Standards in zwei Softwareebenen unterteilt (s. Abb. 3.1). Die Transportschicht (engl. Transport Layer) bildet die erste Schicht. Diese stellt den Zugang zur Kamera her und ruft die Bilddaten von der Kamera ab. Der für den Transport Layer (TL) benötigte Treiber wird durch die verwendete Hardwareschnittstelle definiert. Daher muss an dieser Stelle die Art des Anschlusstyps betrachtet werden. Bekannte Hardwareschnittstellen aus der IBV sind beispielsweise GigE Vision, USB3 Vision und Camera Link. Ein Vergleich dieser Hardwareschnittstellen folgt in Kapitel 3.1.1. Die Treiber werden von Kameraherstellern zur Verfügung gestellt. [AEJ+18]

Die zweite Schicht bildet die Schnittstelle zwischen der Bildverarbeitungsanwendung des Nutzers und der Transportschicht. Sie wird als Bilderfassungsbibliothek (engl. image acquisition library) bezeichnet. Die Kamerahersteller haben diese in ihre Software Development Kits (SDK) integriert. Die Bilderfassungsbibliothek stellt der Anwendung Methoden und Befehle zur Konfiguration und Steuerung der Kamera zur Verfügung und dient somit als Kamera API (engl. Application Programming Interface = Schnittstelle zur Anwendungsprogrammierung). Die zweite Schicht nutzt hierzu den Transportmechanismus der ersten Schicht über die TL API. Es gibt insbesondere zwei Standards für die Softwareschnittstelle. Das sind GenICam und IIDC2, die in Kapitel 3.1.2 beschrieben werden. [AEJ+18]

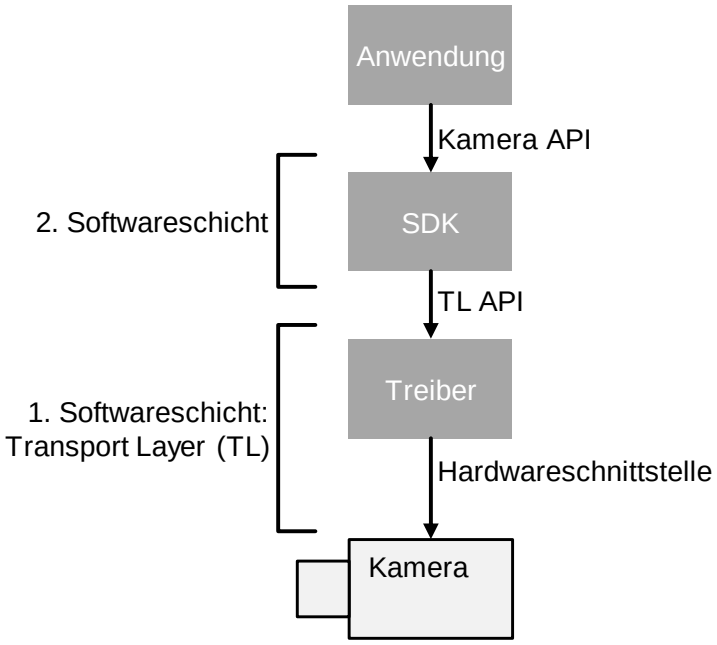


Abb. 3.1: Schlüsselfunktionen bereitgestellt vom Kameraschnittstellen-Standard (i. A. a. [AEJ+18])

3.1.1 Hardwareschnittstelle

Durch die Hardwareschnittstelle wird sichergestellt, dass die Kamera physisch mit der Steuereinheit, beispielsweise einem Industrie-PC, verbunden werden kann. Auf der Steuereinheit muss hierzu stets der passende Treiber installiert sein. Wie in Kapitel 3.1 bereits erwähnt, ist die Hardwareschnittstelle für den Transport Layer und damit für die erste Softwareschicht relevant. Daher wird an dieser Stelle eine Übersicht gegeben. In Tab. 3.1 werden die, durch die G3-Initiative anerkannten und geförderten, Kamera-Hardwareschnittstellen zusammengefasst. Die Standards unterscheiden sich unter anderem in ihrer maximalen Bandbreite zur Bilddatenübertragung, der maximalen Kabellänge, der Stromversorgung über das Datenkabel, der Empfängergeräte und der Echtzeitfähigkeit. Für den Vergleich wurde jeweils die höchste Spezifikation des Standards verwendet, die jeweils unter der Bezeichnung des Standards in Klammern vermerkt ist. Für detailliertere Informationen über die einzelnen Hardwareschnittstellen wird auf den Anhang 10.1. und auf die Datenquellen von Tab. 3.1 verwiesen.

Tab. 3.1: Übersicht der Kamera-Hardwareschnittstellen (Datenquelle: [DSS11; AEJ+18; Aia20a; Aia20b; Aia20c; Aia20d; Jii20])

Hardwareschnittstelle (höchste Spezifikation)	Bandbreite [MB/s]	Kabellänge [m]	Stromversorgung über Datenkabel	Empfängergerät	Echtzeitfähigkeit
GigE Vision (10 GigE)	≤ 1000	> 120	Optional	PC (direkt)	Gut
USB 3 Vision (SuperSpeed 5 Gbits/s)	≤ 500	≤ 120	Notwendig	PC (direkt)	Mäßig
Camera Link (80-bit)	≤ 1000	≤ 10	Optional	Framegrabber	Sehr gut
Camera Link HS (F2)	≤ 5000	≤ 120	Nicht möglich	Framegrabber	Gut
CoaXPress (6X CXP-12)	> 5000	≤ 50	Notwendig	Framegrabber	Gut

3.1.2 Softwareschnittstelle

Die G3-Initiative unterstützt zwei Standards für die zweite Schicht der Kameraschnittstelle: Generic Interface for Cameras (GenICam) sowie Instrumentation & Industrial Digital Camera (IIDC). IIDC wird von keiner Hardwareschnittstelle aus Tab. 3.1 verpflichtend unterstützt. Daher wird IIDC im Folgenden nicht näher beschrieben.

Anders ist das beim GenICam Standard. Hersteller von Kameras mit GigE Vision, USB3 Vision, CoaXPress, Camera Link HS und eingeschränkt Camera Link müssen bei der Verwendung dieser Hardwareschnittstelle verpflichtend die Kompatibilität mit dem GenICam Softwarestandard sicherstellen. Damit ist GenICam der mit Abstand meist verbreitete Standard. Viele herstellerspezifische Anwendungen setzen selbst auch auf diesem Standard als Grundstruktur auf und bieten weitere Funktionalitäten sowie eine Benutzeroberfläche rund um diesen Standard. [AEJ+18] GenICam besteht dabei aus verschiedenen Komponenten. Grundsätzlich dienen jedoch alle dazu die Kamera einheitlich zu konfigurieren und die Bildaufnahme zu steuern. Auf die einzelnen Komponenten des GenICam Standard und die Funktionsweise von GenICam wird nachfolgend eingegangen.

GenAPI (Application Programming Interface)

Die GenAPI definiert den Mechanismus, der verwendet wird, um die generische API über eine selbstbeschreibende XML-Datei im Gerät bereitzustellen. Teil der GenAPI ist ein Schema, das das Format der XML-Datei definiert. Dadurch können die verfügbaren Features der Kamera standardisiert aufgelistet und konfiguriert werden. Als Features werden alle Einstellungen und Funktionen der Kamera bezeichnet. Somit garantiert die GenAPI, dass GenICam kompatible Kameras stets über dieselbe Softwareschnittstelle nutzbar sind. [Lüb20] Die GenAPI stellt die in Kapitel 3.1 beschriebene Bilderfassungsbibliothek der zweiten Softwareschicht dar. Die GenAPI wird durch EMVA den Herstellern und Entwicklern in C++ und Python bereitgestellt. Die Kamerahersteller haben diese in ihre Software Development Kits (SDK) integriert. Für Python-Entwickler steht die Open-Source Software-Bibliothek „genicam“ [The20b] zur Verfügung.

SFNC (Standard Feature Naming Convention)

Dies ist der Teil von GenICam, den die meisten Benutzer sehen. Die SFNC standardisiert den Namen, den Typ, die Bedeutung, die Verwendung und das Verhalten von Gerätemerkmalen, den sogenannten Features. Dadurch verwenden Geräte verschiedener Hersteller immer die gleichen Namen für die gleiche Funktionalität. Diese Funktionen werden in der Regel in einer Baumansicht in Benutzeroberflächen angezeigt oder können direkt von einer Anwendung standardisiert eingestellt und genutzt werden. Kamerahersteller stellen beim Verkauf ihrer Kameras eine Features Referenz zur Verfügung, in der die unterstützten Features und

Werte der jeweiligen Kamera vermerkt sind. Ein verwandter Standard ist die PFNC (Pixel Format Naming Convention), die definiert, wie Pixelformate einheitlich benannt werden, und die verwendeten Formate auflistet. [Lüb20]

GenTL (Transport Layer)

Mit dem GenTL wird die Programmierschnittstelle der Transportschicht standardisiert. Es handelt sich um eine Low-Level-API zur Bereitstellung einer Standardschnittstelle für ein Gerät unabhängig von der Transportschicht (s. Kapitel 3.1). Der GenTL ermöglicht die Aufzählung von Geräten, den Zugriff auf Geräteregister, das Streaming von Daten und die Lieferung asynchroner Ereignisse. Der GenTL hat auch seinen eigenen SFNC. [Lüb20] Zur Implementierung der GenTL Schnittstelle wird ein GenTL Producer benötigt. Dieser wird bei GenICam auch durch den Kameraherstellen oder Softwarehersteller zusammen mit dessen SDK zur Verfügung gestellt. Der GenTL Producer ist eine Softwarebibliothek, der die GenTL-Spezifikation als Abstraktionsebene über der Hardwareschnittstelle implementiert. Das Gegenstück zum GenTL Producer bildet die Bildverarbeitungsanwendung, die als GenTL Consumer bezeichnet wird. [SUW18]

Überblick über die Funktionsweise von GenICam

Die Funktionsweise des GenICam Standards ist in Abb. 3.2 schematisch dargestellt. Durch einen GenTL Producer wird eine GenTL-Schnittstelle generiert. Denn mit Hilfe des GenTL können angeschlossene Kameras gefunden und die XML-Datei, die vom Hersteller im Gerät bereitgestellt wird, übertragen werden. Als darunter liegendes Transportprotokoll können die in Tab. 3.1 dargestellten Hardwarestandards verwendet werden. Aus der XML-Datei wird mithilfe einer Softwarebibliothek die GenAPI als Programmierschnittstelle standardmäßig in der Programmiersprache C++ erzeugt. Als Softwarebibliothek können beispielsweise SDK von Kameraherstellern oder Open-Source Software-Bibliotheken dienen. Über diese GenAPI und GenTL-Schnittstelle kann der GenTL Consumer die Kamera und den GenTL anhand der standardisierten Features konfigurieren. Nach der Konfiguration werden die Bilddaten durch den GenTL Consumer für nachfolgende Anwendungen innerhalb des SDK von Herstellern, sofern dieser auch Bildverarbeitungsalgorithmen in das SDK integriert hat, oder für selbstgeschriebene Applikationen bereitgestellt. Der GenTL Consumer kann somit zum einen Teil eines umfangreicheren SDK eines Kamera- oder Softwareanbieters sein, wie beispielsweise bei dem Softwareprodukt Common Vision Blox der STEMMER IMAGING AG, Puchheim [Ste19a]. Zum anderen ist auch die Open-Source Software-Bibliothek „harvesters“ [The20a] (s. Kapitel 5.1.1) ein GenTL Consumer, der aber keine weitere Bildverarbeitung anbietet und somit lediglich als Bindeglied zur eigenen, selbstgeschriebenen Anwendung dient. Grundsätzlich kann der GenTL Consumer auch selbst entwickelt werden. Da jedoch kommerzielle und Open-Source Lösungen

bereits zur Verfügung stehen und die Entwicklung tiefe Fachkenntnisse über den GenICam Standard erfordert, ist hiervon abzuraten. Wird eine andere Programmiersprache als C/C++ verwendet, muss auf die API der verwendeten Software geachtet werden, da der GenICam Standard in der Referenzimplementierung und die GenTL Producer standardmäßig in C++ bzw. C geschrieben sind.

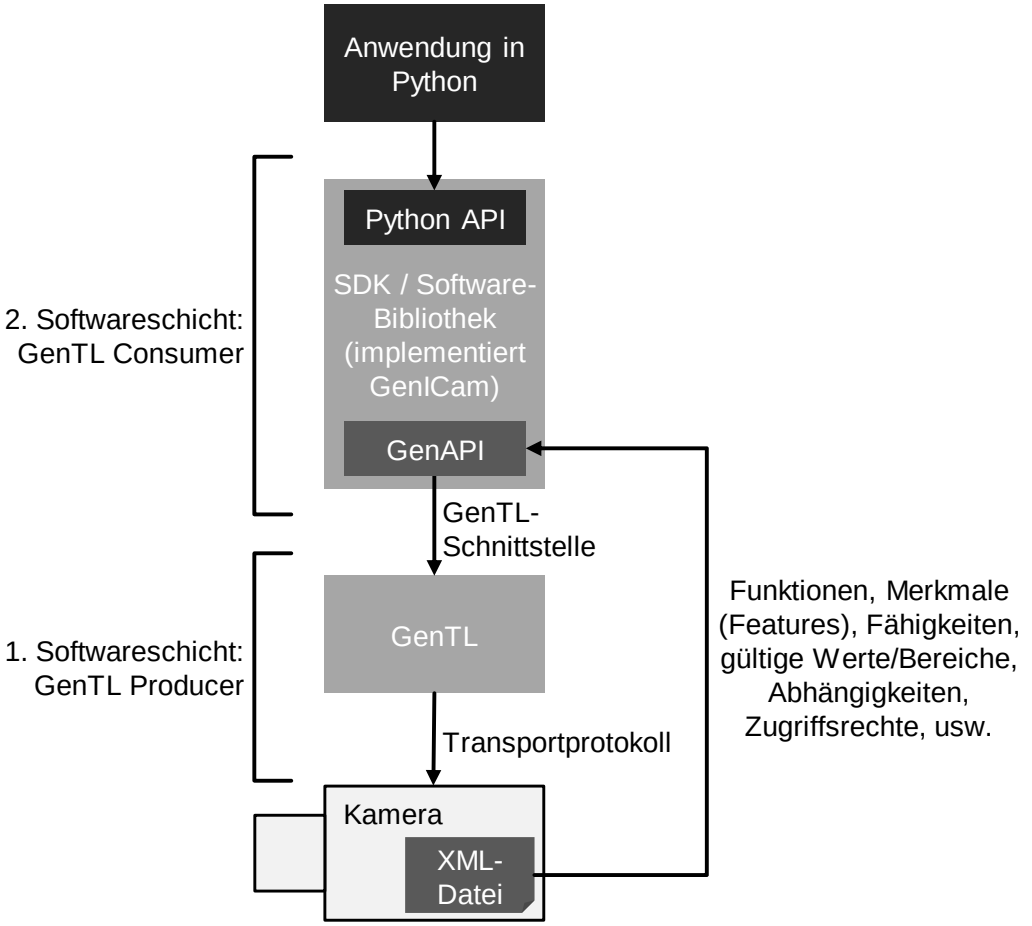


Abb. 3.2: Aufbau der Softwareschnittstelle zur Kamera bei Verwendung des GenICam Standards (i. A. a. [AEJ+18])

3.2 Netzwerk- und Kommunikationsprotokolle

Traditionell war der Bereich der Informationstechnologie (IT) getrennt vom Bereich der operativen Technologien (OT). Zur IT zählen beispielsweise Systeme auf der Unternehmensleitebene wie ERP-Systeme (ERP = Enterprise-Resource-Planning) und die Konnektivität zu Datenbanken und Clouds. Zur OT gehört dagegen die produktionsnahe Steuerung von operativen Abläufen, wie beispielsweise auf der Automatisierungs- und Feldebene mit Maschinen. Vereinfacht wird IT der Unternehmensleitung- und Steuerung in der Verwaltung und OT der operativen Produktion auf dem Shop-Floor zugeordnet. Durch Trends wie das Industrial Internet

of Things (IIoT) und Industrie 4.0 entwickeln sich neue Anforderungen an Netzwerk- und Kommunikationsprotokolle, um die Machine-to-Machine (M2M) Kommunikation zu ermöglichen. Dadurch verschmelzen IT und OT zunehmend, um Daten untereinander in Echtzeit auszutauschen und überall verfügbar zu machen. Häufig werden dazu Technologien aus der IT in den OT-Bereich adaptiert, die anfangs hierzu nicht entwickelt wurden. Dadurch hält beispielsweise das Ethernet aus der IT immer mehr Einzug in die OT. Abb. 3.3 verdeutlicht die beiden Ebenen. [WDK+18; Hub16]

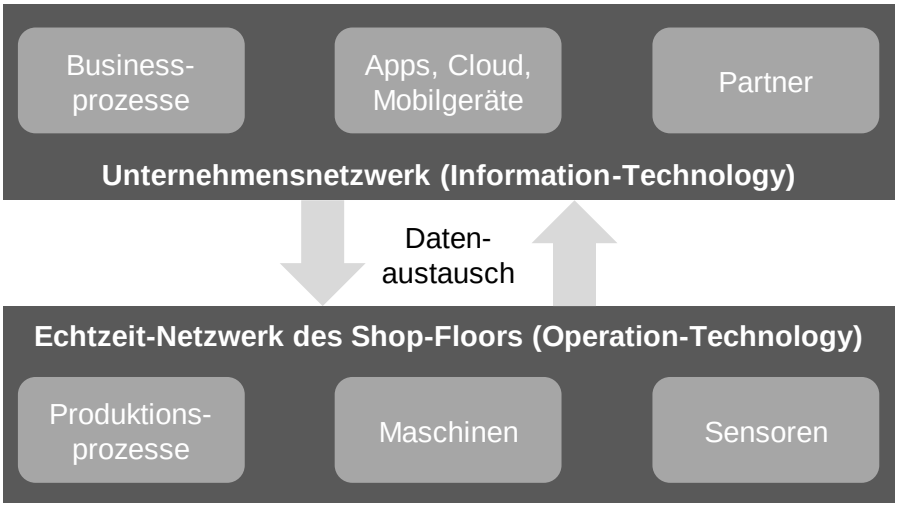


Abb. 3.3: Traditionelle Aufgaben, Prozesse und Geräte der Information- und Operation-Technology (i. A. a. [WDK+18])

Um den Anforderungen aus der Verschmelzung gerecht zu werden, sind flexible Netzwerke notwendig. Hierzu bieten sich modulare und funktionsorientierte Konzepte sowie abstrahierte Lösungen zur Kommunikation, beispielsweise basierend auf Diensten, an. [WDK+18]

Zwei bereits weitverbreitete Netzwerkwerkprotokolle zur Übertragung von Daten zwischen Geräten sind das Message Queuing Telemetry Transport (MQTT) Protokoll und der Open Platform Communications Unified Architecture (OPC UA) Standard [Spe18]. Während MQTT aus dem IT-Bereich kommt und ein sehr schlankes Protokoll darstellt (s. Kapitel 3.2.1), ist OPC UA aus der Automatisierungsindustrie hervorgegangen (s. Kapitel 3.2.2). Nachfolgend werden beide Standards kurz vorgestellt. Der Fokus liegt jedoch auf MQTT, da dieses Protokoll in der Softwareentwicklung aufgegriffen wird.

Kommunikationsprotokolle, die im Verlauf der Bachelorarbeit verwendet und daher ebenfalls an dieser Stelle kurz vorgestellt werden, sind gRPC (s. Kapitel 3.2.3), REST (s. Kapitel 3.2.4) sowie Telnet und FTP (s. Kapitel 3.2.5).

3.2.1 MQTT

Das Kommunikationsprotokoll MQTT wurde von der International Business Machines Corporation (IBM), Armonk, USA, entwickelt und ist ein Netzwerkprotokoll für die M2M Kommunikation. Im IoT (Internet of Things) wird MQTT für die Datenübertragung verwendet. Es ist für Geräte mit eingeschränkter Bandbreite, hoher Latenz oder unzuverlässigen Netzwerken ausgelegt. Denn MQTT ist ein sehr leichtgewichtiges Protokoll mit minimalem Overhead und guter Echtzeitfähigkeit. [LS18; Yua17] MQTT zeichnet sich insbesondere auch durch seine leichte Implementierung aus. Es stehen zahlreiche Implementierungen für die gängigen Programmiersprachen zur Verfügung. [Hub16]

Bei MQTT müssen Herausgeber (engl. Publisher) und Abonnenten (engl. Subscriber) die Identität des jeweils anderen nicht kennen. Wenn ein Benutzer eine Nachricht zu einem bestimmten Thema postet (engl. publish), erhalten alle Benutzer (engl. Clients), die das Thema (engl. Topic) abonniert (engl. subscribe) haben, die Nachricht (s. Abb. 3.4). Dieses Prinzip wird als Publisher/Subscriber-Pattern bezeichnet. Zwischen den Sendern und Empfängern befindet sich der MQTT-Broker, an den Nachrichten gesendet und automatisch durch ein Abonnement an die Empfänger weitergleitet werden. Jede Nachricht ist dabei einer eindeutigen MQTT-Topic zugeordnet.

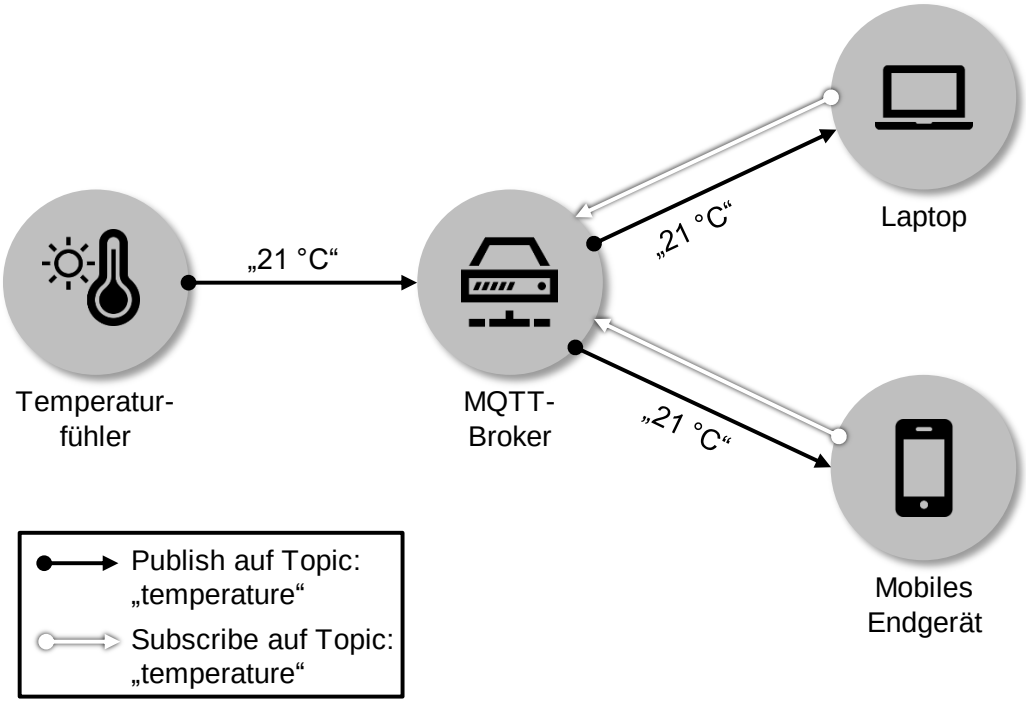


Abb. 3.4: Schematische Darstellung des MQTT-Protokolls über das Publisher/Subscriber-Pattern am Beispiel Temperaturmessung (Bildquelle: [Hub16; Ras17])

MQTT bietet für die Sicherheit der Datenübertragung drei Ebenen der Dienstgüte (engl. Quality of Service = QoS). Stufe 0 bedeutet, dass die Nachricht genau einmal übertragen wird, ohne zu prüfen, ob die Nachricht ankam. Ebene 1 bedeutet, dass jede Nachricht mindestens einmal übertragen wird, indem der Empfänger dem Sender den Erhalt der Nachricht bestätigt. Ebene 2 bedeutet, dass jede Nachricht genau einmal übertragen wird, wobei ein Vier-Wege-Handshake-Mechanismus erforderlich ist, um sicherzustellen, dass die Nachricht genau einmal übertragen wird. Beim Vier-Wege-Handshake-Mechanismus bestätigt sowohl der Empfänger dem Sender den Erhalt der eigentlichen Nachricht als auch der ursprüngliche Sender dem Empfänger den Erhalt der Empfangsnachricht. Je höher der QoS, desto länger dauert der Datenaustausch. [ZZL+19]

MQTT-Nachrichten werden vom MQTT-Broker normalerweise direkt nach dem Empfang verteilt und wieder vom Broker gelöscht. Durch die Kennzeichnung als sogenannte Retained Message ist es möglich, immer die zuletzt gesendete Nachricht unter einem MQTT-Topic zu speichern, bis eine neue kommt. Dadurch können neue Abonnenten direkt nach einem Verbindungsaufbau bereits eine Nachricht erhalten, ohne dass in diesem Moment eine gepostet wurde. Zudem bietet MQTT die Möglichkeit eines letzten Willens (engl. Last Will), der vom MQTT-Broker an alle Abonnenten versendet wird, sobald die Verbindung zu einem Client abbricht. [BBB+19]

3.2.2 OPC UA

OPC UA ist ein offener Kommunikationsstandard, der den industriellen Informationsaustausch auf allen Ebenen der IT und OT spezifiziert. Dabei ist OPC UA unabhängig von Betriebssystemen, Programmiersprachen, Herstellern und Systemlieferanten. OPC UA schafft dadurch die vertikale Integration zwischen IT und OT sowie die horizontale zwischen verschiedenen Herstellern und Systemen. Der Standard legt eine Plug-and-Play Funktionalität fest, wodurch der Funktionsumfang der Kommunikationsteilnehmer durch andere Geräte selbstständig erkannt werden kann. OPC UA bietet ein breiteres Spektrum an Funktionen. So unterstützt es neben dem Publisher/Subscriber-Pattern (s. Kapitel 3.2.1) auch das Client-Server-Modell mit Handshake (s. Abb. 3.5). Beim Client-Server-Modell kann der Client einen Dienst beim Server anfordern. Der Server führt daraufhin eine definierte Funktionalität aus und sendet die angeforderte Antwort. OPC UA eignet sich insbesondere auch dann, wenn sehr viele Daten bevorratet werden sollen. Für zeitkritische Anwendungen bietet sich OPC UA aufgrund der nicht vorhandenen Echtzeitfähigkeit derzeit nicht an. [Hub16] Die reine Datenübertragung mit OPC UA dauert rund zehnmal länger als mit MQTT (s. Kapitel 3.2.1), da es viel komplexer

ist [RSD+19]. OPC UA dient auch als Kommunikationsplattform, um daraus Erweiterungen zu entwickeln. Hierzu zählt beispielsweise die OPC UA Companion Specification Vision, die ein allgemeines Informationsmodell für Bildverarbeitungssysteme zur Verfügung stellt. [AEJ+18]

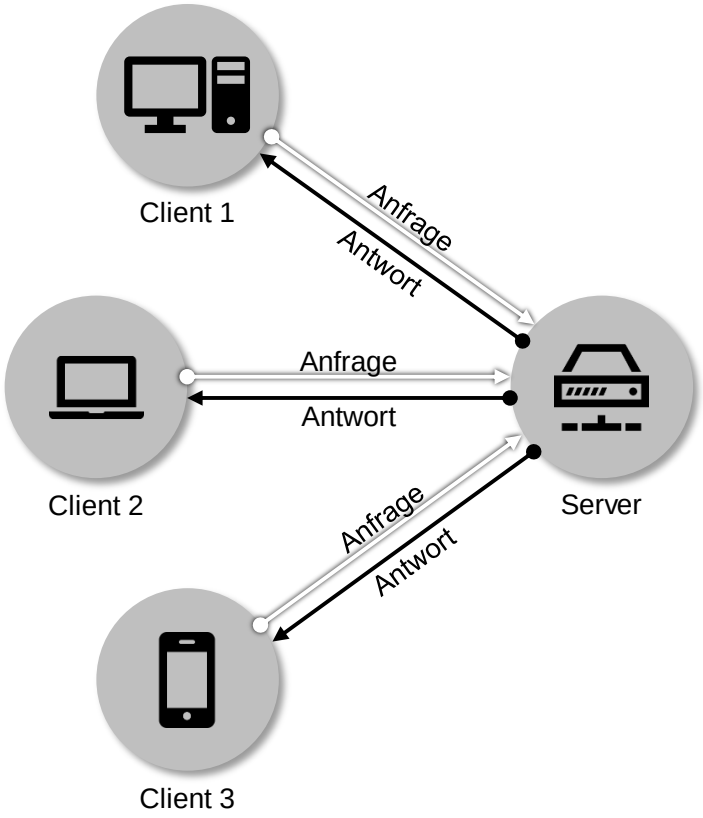


Abb. 3.5: Beispielhafte Darstellung eines Client-Server-Modells

3.2.3 gRPC

Google Remote Procedure Calls (gRPC) ist ein moderner Open-Source Remote Procedure Call (RPC), der von Google entwickelt wurde. RPC werden in Client-Server-Modellen verwendet. Mit diesem Prozeduraufruf ist es möglich die Funktionalität eines Servers abzurufen [MS13]. Aufgrund der geringen Latenz und des hohen Datendurchsatzes eignet sich gRPC insbesondere für Dienste in einer Microservice-Architektur, um diese sehr effizient miteinander zu verbinden. gRPC-Nachrichten werden mit Protocol Buffers (Protobuf), einem effizienten binären Nachrichtenformat, serialisiert. Protobuf serialisiert sehr schnell auf dem Server und dem Client. Die Protobuf-Serialisierung führt zu kleinen Datenmengen, was in Szenarien mit begrenzter Bandbreite wichtig ist. Dabei können die Prozeduren der Clients in unterschiedlichen Programmiersprachen geschrieben sein. Für den Versand wird der HTTP/2 Standard verwendet. HTTP/2 ist eine größere Überarbei-

tung der Vorgängerversionen von HTTP (Hypertext Transfer Protocol) mit erheblichen Leistungsvorteilen. Das Hypertext-Übertragungsprotokoll dient der Datenübertragung in einem Netzwerk. Mithilfe des Protokolls werden Nachrichten zwischen Client und Server ausgetauscht. [New19; Grp20]

3.2.4 REST

REST ist ein Architekturansatz für verteilte Systeme wie Web Services und eignet sich für die M2M Kommunikation. REST funktioniert nur mit Client-Server-Modellen. REST zeichnet sich dabei insbesondere durch die einheitliche Schnittstelle aus, die als REST API bezeichnet wird. REST definiert allerdings keinen neuen Standard, sondern greift auf bestehende standardisierte Verfahren zurück. Dazu zählen unter anderem HTTP und JSON. Die JavaScript Object Notation (JSON) ist ein Datenformat zum Austausch einfach lesbarer Textnachrichten zwischen Anwendungen. [Sro17; Fie00]

Im Rahmen dieser Bachelorarbeit wird nur die REST API, also die REST Programmierschnittstelle, die zur Client-Server-Kommunikation genutzt werden kann, weiterbetrachtet. Über diese Schnittstelle können mittels HTTP-Anfragen beim Server Ressourcen abgerufen werden. Eine Ressource ist dabei eine beliebige Information, die auf dem Server an einem bestimmten Ort abgelegt ist. Dazu dienen insbesondere die HTTP-Methoden GET, POST und PUT. Mit GET wird eine Ressource auf dem Server lesend abgerufen. Neue Ressourcen lassen sich mit POST und PUT anlegen. [Dri17] Weitere Details sind in der Dokumentation des HTTP-Standards nachzulesen [BPT15].

3.2.5 Telnet und FTP

Das Teletype Network (Telnet) Protokoll und das File Transfer Protocol (FTP) wurden bereits gemeinsam in den 1970ern entwickelt. Beide Protokolle basieren auf dem Client-Server-Modell und erfordern zur Authentifizierung der Verbindung die Eingabe von Benutzernamen und Passwort. Telnet ermöglicht den Aufbau einer remote Verbindung zwischen einem Client und einem Server, um textbasierte Nachrichten auszutauschen. FTP dient der Übertragung von Dateien in einer Client-Server-Kommunikation. Der Server stellt Dateien zur Verfügung, die durch einen Client angefordert werden können. [Whe11; MS13]

3.3 Docker

Die bekannteste Software zur Umsetzung einer Containervirtualisierung ist die Software Docker von Docker Inc., Palo Alto, USA [Kof18]. Diese Open-Source Software verwendet die Containervirtualisierung zur Bereitstellung von Anwendungen. Docker Inc. hat 2015 der Open Container Initiative (OCI) die Container-Image-Spezifikationen und den Laufzeitcode gestiftet und damit den Industriestandard für Container geschaffen. Docker wurde zunächst für Linux-Systeme entwickelt. Jedoch werden auch Windows Betriebssysteme seit einer Partnerschaft mit der Microsoft Corporation, Redmond, USA, unterstützt. [Doc20]

Docker-Compose ist dabei das Werkzeug zum Ausführen von Anwendungen mit mehreren Docker Containern. Damit wird die Laufzeitumgebung erzeugt und die Anwendung isoliert ausgeführt. In Kapitel 6.3 wird gezeigt, wie dieses Tool zur Umsetzung der Containervirtualisierung und zum Starten der Software genutzt wird.

4 Ableitung der Softwareanforderungen

Zunächst werden die Allgemeinen Anforderungen, die sich aus der wissenschaftlichen Fragestellung ergeben, sowie die Rahmenbedingungen der Softwareumgebung definiert (s. Kapitel 4.1). Anschließend wird der Gesamtprozess einer Qualitätsprüfung mittels Deep Learning betrachtet, um daraus auf die Teilprozesse zu schließen (s. Kapitel 4.2). Diese Teilprozesse werden beschrieben, um daraus die funktionalen Anforderungen an die Software zu gewinnen (s. Kapitel 4.3).

4.1 Allgemeine Anforderungen und Rahmenbedingungen

Für den Begriff der Plug-and-Play Fähigkeit gibt es in der wissenschaftlichen Literatur keine einheitliche, detaillierte Definition. Grundsätzlich wird darunter das Anschließen beliebiger Hardware an einen Computer verstanden, ohne dass der Nutzer weitere Treiber installieren oder eine Konfiguration vornehmen muss [Sch03; Gei18]. Der Begriff entstand für Computer und Geräte im privaten Bereich. Im industriellen Kontext ergeben sich jedoch im Vergleich dazu individuelle Einsatzgebiete, die ohne Anpassungen und eine angemessene Konfiguration durch den Nutzer nicht vollständig Plug-and-Play umsetzbar sind. Als Beispiel wird an dieser Stelle auf den Prozess der Kamerakonfiguration in Kapitel 4.3.1 verwiesen, die exakt auf die Inspektionsaufgabe abgestimmt sein muss. Daher muss dieser Konfiguration für industrielle Anforderungen durch einen Bediener durchgeführt werden. Unter der Plug-and-Play Fähigkeit des Systems werden in dieser Bachelorarbeit folgende Aspekte verstanden:

- Angeschlossene Geräte im Netzwerk werden automatisch erkannt
- Keine Quellcodeanpassungen durch den Nutzer an der Software notwendig
- Benötigte Treiber, Software-Bibliotheken und allgemeine Einstellungen werden automatisch installiert
- Individuelle Prozessparametereingabe über GUI oder Umgebungsvariablen (Parameter, die von der Hardwareauswahl oder der Inspektionsaufgabe abhängig sind)
- Abfangen falscher Prozessparameter und anschließender Korrektur bzw. Fehlerrückmeldung an Nutzer
- IBV-System robust gegenüber Abstürzen (automatischer Neustart)

Wie bereits zu Beginn der Bachelorarbeit beschrieben (s. Kapitel 1.2), wird das System für Kameras ausgelegt, die am DCC für Testzwecke verfügbar sind. Dazu zählen eine Kamera der Cognex In-Sight Serie sowie zwei auf dem GigE Vision Standard basierende Kameras von Allied Vision (s. Kapitel 5.1.1). Zudem werden

nur Software-Lösungen und Bibliotheken verwendet werden, die ohne Lizenzgebühren genutzt werden können, um der wissenschaftlichen Fragestellung zu entsprechen. Das daraus entwickelte Gesamtsystem soll dennoch kommerziell vertrieben werden können.

4.1.1 Programmiersprache

Python, Java und C/C++ zählen zu den populärsten Programmiersprachen [Cas19; Kam20]. Sie zeichnen sich durch ein großes Anwendernetzwerk und eine Vielzahl von Software-Bibliotheken aus. Besonders für Python wird eine hohe Popularität in den nächsten Jahren vorhergesagt, wie eine Untersuchung der Gesamtsuchanfragen pro Jahr in Stack-Overflow ergab (s. Abb. 4.1) [Ant19]. Für die Laufzeiteffizienz einer Programmiersprache ist insbesondere die Unterscheidung zwischen kompilierten und interpretierten Sprachen wichtig. Kompilierte Programmiersprachen wie C/C++ kompilieren den Programmcode vor der Ausführung vollständig in Maschinensprache. Danach wird der Code nur noch ausgeführt. Python dagegen ist eine interpretierte Programmiersprache. Hierbei wird nicht zu Beginn der gesamte Code kompiliert, sondern er wird Zeile für Zeile in Maschinensprache übersetzt und ausgeführt. Das kostet insbesondere bei Schleifen, die immer wieder neu übersetzt werden müssen, viel Zeit während der eigentlichen Ausführung des Programms. [Pre00]

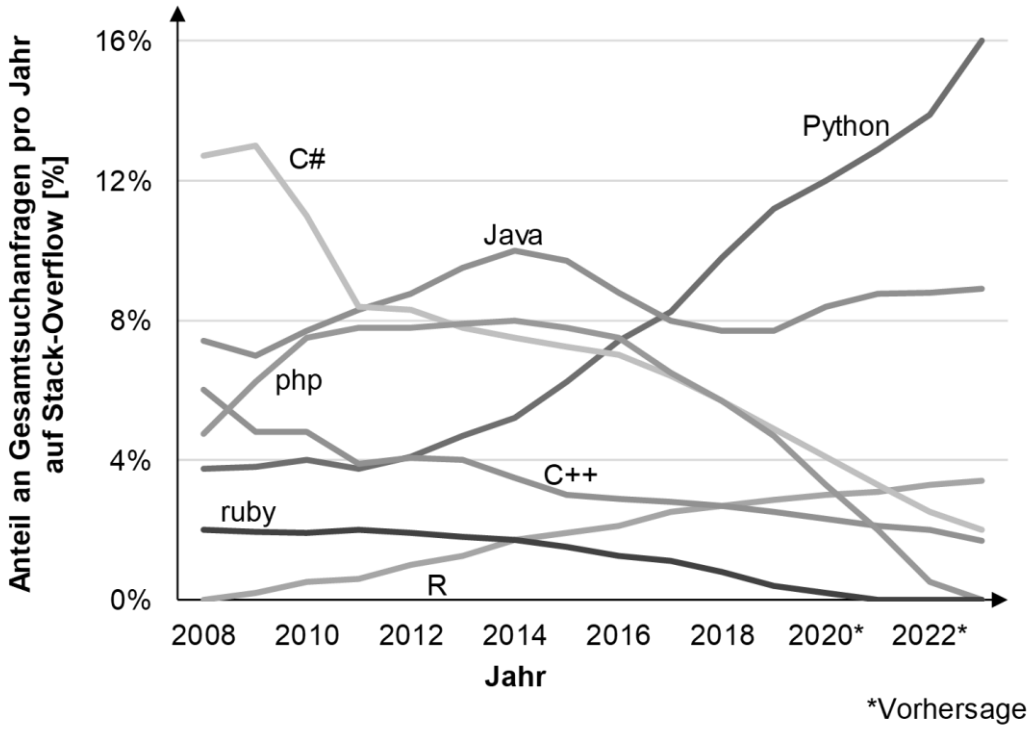


Abb. 4.1: Vorhersage des Anteils der Gesamtsuchanfragen pro Jahr in Stack-Overflow (Datenquelle: [Ant19])

Je nachdem wie nah eine Programmiersprache der Maschinensprache ist, wird diese als Low-Level oder High-Level Programmiersprache bezeichnet. Grundsätzlich ist die Umsetzung in einer High-Level Programmiersprache wie Python einfacher in der Implementierung für den Anwender. Python bietet eine vergleichsweise einfache Syntax. Dadurch wird sichergestellt, dass die Software zukunftssicher und für viele verständlich ist. Das ist besonders bei einer Plug-and-Play Umsetzung sinnvoll, um Anpassungen und Weiterentwicklungen zu ermöglichen. [Sid19; WRA96] Eine Umsetzung in Low-Level Programmiersprachen wie C/C++ bietet sich an, wenn die Verarbeitungsgeschwindigkeit in Python zu gering wäre. In dieser Bachelorarbeit wird die Annahme getroffen, dass die Verarbeitungsgeschwindigkeit in einer interpretierten Sprache ausreicht.

4.1.2 Betriebssystem

Die meist verbreiteten Betriebssysteme für die IBV sind Windows und Linux. Dabei holt Linux immer weiter auf Windows auf. Der Open-Source Charakter von Linux ist ein Vorteil, weil die inneren Prozesse transparenter sind und es somit besser möglich ist, Nuancen oder Fehler zu umgehen, die möglicherweise auf den untersten Ebenen des Betriebssystems vorhanden sind. Darüber hinaus ist Linux besser geeignet, um eine optimierte Umgebung für die IBV zu schaffen, da es relativ einfach ist, nur die Komponenten, die für die Lösung notwendig sind, und keine zusätzlichen Treiber und Dienstprogramme, die nicht benötigt werden, einzubinden. [Hol17] Aus diesen Gründen wird auch in dieser Arbeit, insbesondere bei selbst entwickelten Teilprozessen, auf Linux aufgebaut.

4.2 Gesamtprozess eines Systems zur IBV mittels Methoden des Deep Learnings

Der betrachtete Gesamtprozess in dieser Bachelorarbeit basiert auf dem typischen Grundaufbau eines Systems in der IBV, wie in Kapitel 2.3.4 beschrieben. Dabei werden die Teilbereiche Bildgewinnung und Datenauswertung betrachtet, wobei in der Datenauswertung der Fokus auf der Verarbeitung mittels Methoden des Deep Learnings liegt. Für die Ergebnisverwertung werden die Ergebnisse der Auswertung standardisiert abgelegt, der eigentlich folgende Prozess der Verwertung aber nicht mehr betrachtet.

Mit diesen Prämissen ergibt sich somit folgender Gesamtprozess, der in Abb. 4.2 dargestellt ist. Der Prozess beginnt mit der Kamerakonfiguration, die nur einmal durchgeführt wird. Nach jeder neuen Konfiguration muss das KNN neu trainiert werden. Anschließend folgt ein Kreisprozess mit dem Auslösen des Bildes und der

Bildaufnahme. Alle aufgenommenen Bilder werden für ein etwaiges Training gespeichert, auch nachdem bereits ein KNN trainiert wurde, um mit mehr Daten neue Trainings durchführen zu können. Sofern kein trainiertes KNN vorhanden ist, muss dieses zuerst mit Bilddaten, die zuvor gelabelt werden, trainiert werden (s. Training Kapitel 2.2.4). Ein erfolgreiches Training ist nur mit ausreichend großen Datensätzen möglich. Die benötigte Anzahl an Bildern hängt vom Training und dem Design des KNN ab und wird in dieser Arbeit nicht betrachtet. Sobald ein trainiertes KNN zur Verfügung steht, wird die Inferenz durchgeführt (s. Inferenz Kapitel 2.2.4). Bei Bedarf kann jederzeit ein neues KNN trainiert werden und in der Inferenz genutzt werden (in Prozessbild durch gepunktete Linie verdeutlicht). Nach der Inferenz werden die Ergebnisse der Verarbeitung an Folgeprozesse weitergegeben (in Prozessbild durch gestrichelte Linie verdeutlicht). Der Kreisprozess startet erneut beim Trigger.

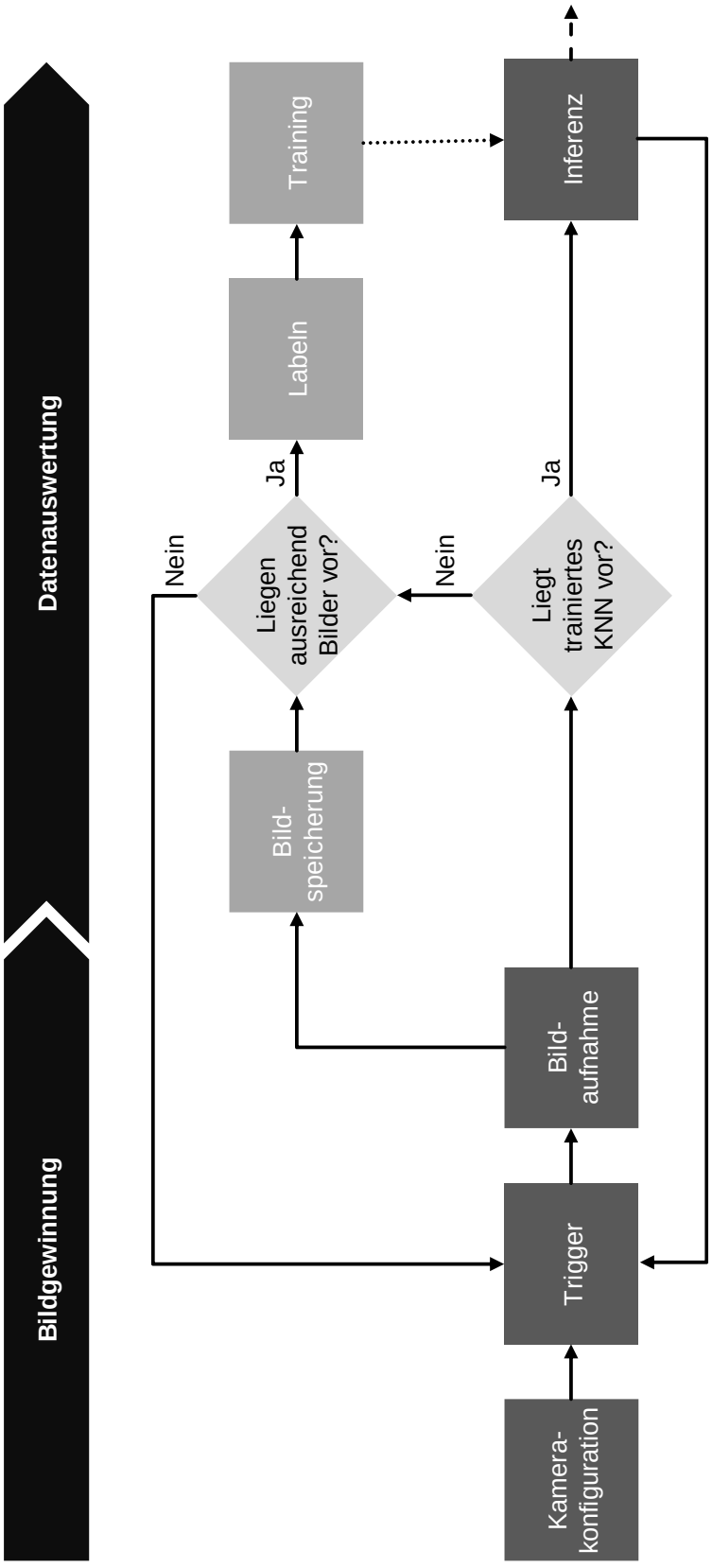


Abb. 4.2: Gesamtprozessdiagramm eines Systems zur IBV mittels Methoden des Deep Learnings

4.3 Funktionale Anforderungen der Teilprozesse

Die Betrachtung des Gesamtprozesses in Kapitel 4.2 hat gezeigt, dass dieser aus sieben differenzierbaren Teilprozessen besteht. Nachfolgend werden für jeden Teilprozess die Funktionen und Anforderungen beschrieben. Die Reihenfolge orientiert sich dabei an dem in Kapitel 4.2 beschriebenen Ablauf des Gesamtprozesses, in dem vor der eigentlichen Inferenz das Training erfolgen muss. In Kapitel 4.4 werden die funktionalen Beschreibungen sowie die jeweiligen Ein- und Ausgaben kurz zusammengefasst, um auch die Schnittstellen zwischen den Teilprozessen zu verdeutlichen.

4.3.1 Kamerakonfiguration

In diesem Teilprozess muss es möglich sein, die Kameraparameter einzustellen. Je nach Art und verwendetem Standard unterscheiden sich die Einstellmöglichkeiten. Smart Kameras und Kameras mit dem GenICam Standard bieten auch die Möglichkeit Parameter automatisch einzustellen. Jedoch müssen die Parameter stets von einem Nutzer geprüft werden und auf die konkrete Aufgabe hin festgelegt werden.

Das Einstellen findet experimentell durch Austesten verschiedener Änderungen an den Parametern statt. Um die Änderungen an den Kameraeinstellungen als Nutzer direkt nachvollziehen zu können, ist eine Benutzeroberfläche (engl. Graphical User Interface = GUI) zum Konfigurieren notwendig, die auch die aktuelle Aufnahme für den Benutzer darstellt.

Zu den Standardeinstellungen zählt der Betrachtungsbereich, der häufig mit dem englischen Begriff Region of Interest (ROI) bezeichnet wird (s. Abb. 4.3). Hierbei wird zum einen die Pixelhöhe und -weite des Bildes sowie der Versatz (engl. Offset) eingestellt. Der Versatz beschreibt, ab welcher Pixelspalte bzw. -zeile relativ zur ersten Spalte bzw. Zeile die Bildaufnahme beginnt. Es muss stets darauf geachtet werden, dass nur das zu prüfende Objekt im Sichtfeld ist. Die Auflösung muss je nach Größe des kleinsten zu prüfendem Merkmal gewählt werden. Je nach Kameramodell ist auch der Farbraum des ausgegebenen Bildes einstellbar. Hierbei wird insbesondere zwischen einer monochromatischen (Graustufen) und polychromatischen (farbigen) Ausgabe unterschieden. [Tel17] Sofern monochromatische Bilder für die Inspektionsaufgabe ausreichen, sollten diese im Sinne der Bildtransfergeschwindigkeit bevorzugt werden. Je nach Aufgabe der Inspektion muss die Belichtungszeit angepasst werden. Beispielsweise kann eine Überbelichtung Kontraste besser herausstellen. Auf die Wichtigkeit der Belichtung im Allgemeinen wurde bereits in Kapitel 2.3.4 (Belichtung) eingegangen. Das Ziel des Prozesses ist eine konfigurierte Kamera bereit zur Bildaufnahme zur Verfügung zu stellen.

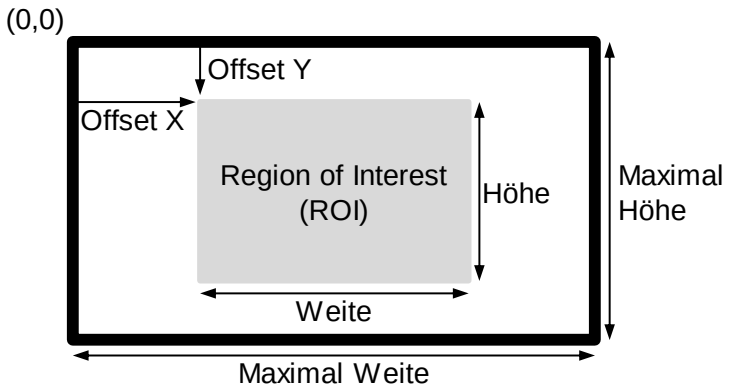


Abb. 4.3: Bildbereich und Region of Interest (ROI) (Bildquelle: [Emv19])

4.3.2 Trigger

In der Bachelorarbeit wird angenommen, dass ein digitaler Auslöser (engl. Trigger) zur Verfügung steht, dessen Signal über eine Standardschnittstelle zum Auslösen der Bildaufnahme genutzt wird. Die Schnittstelle muss hierzu definiert und vorgegeben werden. Sie wird das Anschließen verschiedener Sensoren ermöglichen. Das heißt die Schnittstelle wird nicht von der konkreten Art des Sensors, sondern nur dem digitalen Signal und der darin enthaltenen Nachricht abhängen. Beispielsweise kann ein Laserabstandssensor verwendet werden, der ab einem gewissen Abstandsschwellwert erkennt, dass sich ein Objekt durch die Laserschranke bewegt hat.

Da die Latenz zwischen Aussenden des Sensorsignals und Auslösen der Kamera von den jeweiligen Implementierungsbedingungen, wie Kabellängen und Prozessorgeschwindigkeit, abhängt, muss die Zeit zwischen Sensorsignal und Aufnahme experimentell bei der Implementierung bestimmt und als Parameter hinterlegt werden. Hierüber kann auch bei Fließprozessen die Zeit, die ein Objekt zwischen der Messstelle des Auslösesensors und dem Kamerasensor benötigt, berücksichtigt werden.

Neben dem ereignisgesteuerten Auslösen mit einem Sensor wird auch die Möglichkeit für ein Auslösen mit festem Zeitintervall zur Verfügung stehen. Diese Art des Triggers eignet sich für kontinuierliche Produktionen, in denen ein kontinuierlicher Materialfluss vorliegt, und dennoch Flächenkameras statt Linienkameras genutzt werden (s. Kamerasystem Kapitel 2.3.4).

4.3.3 Bildaufnahme

Die Aufgabe des Teilprozesses Bildaufnahme (engl. Image Acquisition) liegt darin mit der konfigurierten Kamera ein Bild nach Eingang eines Auslösesignals aufzunehmen. Das digitale Bild muss für nachfolgende Prozessschritte in einem Standardformat zur Verfügung gestellt werden.

Als Annahme wird in dieser Bachelorarbeit festgelegt, dass eine Pulslichtunterstützung nur für Kameras, die eine integrierte Beleuchtung besitzen, umgesetzt wird. Da in allen anderen Fällen die Ansteuerung sehr individuell ausfällt, muss diese bei der Systemintegration separat betrachtet werden oder die Beleuchtung dauerhaft eingeschaltet bleiben.

4.3.4 Bildspeicherung

Dieser Teilprozess stellt einen notwendigen Zwischenprozess dar, um die aufgenommenen digitalen Bilder für das Labeln und anschließende Training zu sammeln. Die Bilder werden hierzu lokal in einem Ordner gespeichert. Das Speichern wird nicht nur direkt vor dem Training des ersten KNN stattfinden, sondern auch kontinuierlich parallel zur Inferenz möglich sein. Dadurch kann das Training mit größeren Datensätzen wiederholt werden.

4.3.5 Labeln

Da beim Überwachten Maschinellen Lernen klassifizierte Bildaufnahmen genutzt werden, damit das KNN prüfen kann, ob die eigene Vorhersage richtig war, müssen die Bilder nach der Bildaufnahme von einem menschlichen „Lehrer“ klassifiziert werden. Die lokal gespeicherten Bilder in einem Ordner müssen dazu in verschiedene Ordner sortiert werden, die dem jeweiligen Label entsprechen und nach diesem benannt sind. Diese Ordnerstruktur bildet die Grundlage für alle gängigen Trainingsumgebungen.

4.3.6 Training

In diesem Teilprozess wird das KNN für die anschließende Inferenz trainiert. Dieser Prozess wurde bereits in Kapitel 2.2.4 (Training) beschrieben. Der Prozess benötigt als Eingabe Ordner, die nach dem jeweiligen Label benannt sind und die die Bilder der jeweiligen Klasse enthalten. Die Anzahl der benötigten Bilder je Label hängt von der Datenqualität, der Eindeutigkeit der Merkmale und der gewählten Netzarchitektur ab. Hierzu lässt sich keine pauschale Aussage treffen. Grundsätzlich steigt jedoch die Genauigkeit, je mehr Trainingsdaten zur Verfügung stehen. Die Ausgabe des trainierten KNN ist kompatibel zur Inferenzplattform zu wählen.

4.3.7 Inferenz

In der Inferenz wird die Klasse eines neu aufgenommenen Bildes mithilfe des trainierten KNN vorhergesagt. Der Prozess muss als Ergebnis eine Wahrscheinlichkeit für jede Klasse liefern, mit der das Bild dieser zugeordnet werden kann. Die Klasse mit der höchsten Wahrscheinlichkeit ist dann das vorhergesagte Label des Bildes. Mit der Kenntnis darüber, welche Label eine unzureichende Qualität darstellen, kann die Gesamtqualität beurteilt werden. Dieses Ergebnis muss für nachfolgende Prozesse standardisiert zur Verfügung gestellt werden.

4.4 Zusammenfassung und Schlussfolgerung

Aufgrund der Ergebnisse dieses Kapitels müssen folgende allgemeine Anforderungen und Rahmenbedingungen fortan in der Softwareentwicklung berücksichtigt werden:

- Plug-and-Play Fähigkeit (Definition s. Kapitel 4.1)
- Kameraschnittstelle für Kameras der Cognex In-Sight Serie und Kameras mit GigE Vision Standard
- Ausschließlich Verwendung lizenzgebührenfreier Softwareprodukte und Software-Bibliotheken, die eine kommerzielle Nutzung erlauben
- Programmiersprache: Python
- Betriebssystem: Linux

Die Zerlegung des Gesamtprozesses in Teilprozesse hat gezeigt, dass eine Vielzahl von funktionalen Anforderungen für die Entwicklung und Inbetriebnahme eines IBV-Systems berücksichtigt werden müssen. Die beschriebenen Teilprozesse sind in Abb. 4.4 zusammenfassend mit einer funktionalen Beschreibung sowie den Ein- und Ausgaben dargestellt.

Im nächsten Schritt gilt es zu analysieren, welche Softwareprodukte und Open-Source Software-Bibliotheken lizenzgebührenfrei für die Teilprozesse verfügbar sind, um hieraus Rückschlüsse auf die verbleibenden Entwicklungsziele zu ziehen. Dazu wird die zusammenfassende Prozessübersicht aus Abb. 4.4 zur Einordnung weiterverwendet.

Pro-zess	Bildgewinnung			Datenauswertung (Deep Learning spezifisch)			
	Kamerakonfig.	Trigger	Bildaufnahme	Bildspeicher.	Labeln	Training	Inferenz
Beschreibung	Einstellen der Kamera für spezifische Prüfaufgabe	Auslösen des Prozesses (ereignisgesteuert oder kontinuierlich)	Aufnahme eines digitalen Bilds (visuelles Abbild des Prüfobjekts)	Bilder für Training speichern und bevorraten	Aufgenommene Bilder mit Label klassifizieren	Trainieren eines Künstlichen Neuronalen Netzes (KNN)	Klassifikation der Bilddaten mittels Deep Learning Inferenz
Eingabe	Einstellungsparameter	Messwert des Sensors oder Zeitintervall	Konfigurierte Kamera + Digitales Auslösesignal	Digitales Bild	Lokal abgespeicherte Bilder	Bilder abgespeichert in nach Labeln benannten Ordnen	Trainiertes KNN + digitales Bild
Ausgabe	Konfigurierte Kamera bereit zur Bildaufnahme	Digitales Auslösesignal	Digitales Bild	Lokal abgespeicherte Bilder	Bilder abgespeichert in nach Labeln benannten Ordnen	Trainiertes KNN	Zugewiesenes Label und Qualität des aktuellen Bilds/ Prüfobjekts

Abb. 4.4: Übersicht über Teilprozesse mit kurzer funktionaler Beschreibung sowie Ein- und Ausgaben

5 Softwarelösungen und Systemarchitektur

Zu jedem Teilprozess wurden mögliche Softwarebausteine und Open-Source Software-Bibliotheken recherchiert, um die in Kapitel 4 beschriebenen Funktionen und Anforderungen für den jeweiligen Prozess zu erfüllen. Diese werden nachfolgend kurz vorgestellt (s. Kapitel 5.1). Die Auswahl beschränkt sich, wie gefordert, auf lizenzgebührenfreie, kommerziell nutzbare Softwarelösungen. Auf Basis dieser Auswahl wird eine begründete Eingrenzung vorgenommen, die im weiteren Verlauf der Arbeit genutzt wird (s. Kapitel 5.2). Mit dem Wissen darüber, welche Softwarelösungen in den einzelnen Teilprozessen verwendet werden, wird anschließend betrachtet, wie die Teilprozesse zu einem Gesamtsystem verbunden werden können (s. Kapitel 5.3). Dazu wird sowohl die Softwarearchitektur als auch die primäre Kommunikationsweise in dieser Architektur betrachtet. Daraus werden die verbleibenden Entwicklungsschritte für Microservices abgeleitet (s. Kapitel 5.4). Alle verwendeten Softwarelösungen und Microservices werden auf Prozessebene zusammen mit den genutzten Kommunikationsprotokollen in Kapitel 5.5 zusammengefasst.

5.1 Analyse verfügbarer Softwareprodukte für Teilprozesse

Nachfolgend werden die Softwarelösungen für die Teilprozesse beschrieben. Die Reihenfolge entspricht dabei der bereits zur Beschreibung in Kapitel 4.3 verwendeten Abfolge. Für den Teilprozess Bildspeicherung werden keine verfügbaren Softwareprodukte vorgestellt, da dieser Teilprozess maßgeblich von der Gesamtsystemarchitektur abhängt und daher nach der Beschreibung der Systemarchitektur selbst passend entwickelt wird (s. Kapitel 6.2.3).

5.1.1 Kamerakonfiguration und Bildaufnahme

Da die Software, die für die Kamerakonfiguration verwendet wird, auch für die Bildaufnahme und vice versa verwendet wird, werden die beiden Teilprozesse gemeinsam betrachtet. Am DCC stehen für die Bachelorarbeit folgende Industriekameras zur Verfügung:

- Mako G-223C (Allied Vision Technologies GmbH, Stadtroda) mit GigE Vision Kameraschnittstellenstandard
- Mako G-503C (Allied Vision Technologies GmbH, Stadtroda) mit GigE Vision Kameraschnittstellenstandard

- In-Sight 2000-230 (Cognex Corporation, Natick, USA) ohne Kameraschnittstellenstandard; dafür allgemeine Schnittstellen wie Telnet und OPC UA

Wie bereits in den Anforderungen definiert, müssen diese Kameras von der Systemsoftware unterstützt werden. Nachfolgend werden für die Kameras Softwarelösungen zur Umsetzung der beiden Teilprozesse vorgestellt. Diese stellen jeweils eine API bereit, über jene die Kamera in der selbst entwickelten Applikation angesteuert werden kann. Das heißt sie stellen die Ressourcen und Werkzeuge bereit, um die Kameraansteuerung in das Gesamtsystem zu integrieren. Die beiden Kameras von Allied Vision werden zusammenbetrachtet, da sie die gleichen Schnittstellen haben.

Allied Vision Mako G-223C und Mako G-503C

Für die Kamerakonfiguration und Ansteuerung der Allied Vision Mako Kameras wird vom Hersteller die Software Vimba zur Verfügung gestellt. Mithilfe der Software kann die Kamera benutzerfreundlich mittels GUI konfiguriert werden. Da die Einstellungen gespeichert werden können, kann die Konfiguration an einem beliebigen Gerät mit Windows oder Linux erfolgen. Die Konfiguration ist somit entkoppelt vom Industrierechner, auf dem die spätere Bildverarbeitung ablaufen wird. Zum Zeitpunkt der Analyse bot Vimba jedoch lediglich eine API für C++ und C. Für Python existierte nur ein Python-Wrapper als Open-Source Software-Bibliothek, die privat entwickelt und unter MIT Lizenz zur Verfügung steht. Diese Software-Bibliothek mit dem Namen „pymba“ [Mor20] ist ein Wrapper für die Vimba API in C. Ein Wrapper ist eine Hülle, die eine andere Software umgibt, um diese kompatibel zur eigenen Programmiersprache zu machen. Mit Pymba kann somit eine Schnittstelle zum eigenen Code hergestellt werden.

Vimba basiert auf GenICam (s. Kapitel 3.1.2) und ist daher auch kompatibel mit den GenICam Hardwareschnittstellen GigE Vision und USB3 Vision (s. Kapitel 3.1.1). Das bietet den Vorteil, dass alle Kameras mit GenICam Standard nutzbar wären. Jedoch untersagen die Lizenzbedingungen von Vimba ausdrücklich die Nutzung fremder Hardware. Daher wäre eine Nutzung von Vimba als Kameraschnittstelle langfristig immer auf Kameras von Allied Vision beschränkt. EMVA bietet von offizieller Seite eine Möglichkeit zur herstellerunabhängigen GenICam Nutzung in Python durch Nutzung der Python-Software-Bibliothek „genicam“ [The20b] als GenICam-Python-Anbindung. Da hiermit nur einzelne Features gesteuert werden können, ist diese Bibliothek nicht intuitiv nutzbar. Es existiert aber ein Schwesterprojekt, auf das in der Bibliothek Genicam verwiesen wird und eine Referenzimplementierung des Moduls Genicam darstellt. Diese Open-Source Software-Bibliothek namens „harvesters“ [The20a] (auch als Harvester bezeichnet) erleichtert die Nutzung von Genicam, da Features zu Python-Funktionen zu-

sammengefasst wurden. Harvesters ist zusammen mit der Software-Bibliothek Genicam ein Python-Wrapper für den GenICam-Standard. Harvesters verwendet dazu die GenTL Producer aus den SDK der Kamerahersteller, da es keinen eigenen GenTL Producer besitzt (s. Kapitel 3.1.2). Die Zusammenhänge zwischen den einzelnen Modulen zum Aufbau einer Kameraverbindung sind in Abb. 5.1 dargestellt. Harvesters wird unter der Apache-Lizenz-2.0 angeboten und kann somit auch für kommerzielle Anwendungen frei verwendet, modifiziert und verteilt werden. Das Modul wird jedoch nicht offiziell durch den GenICam Betreuer EMVA bereitgestellt.

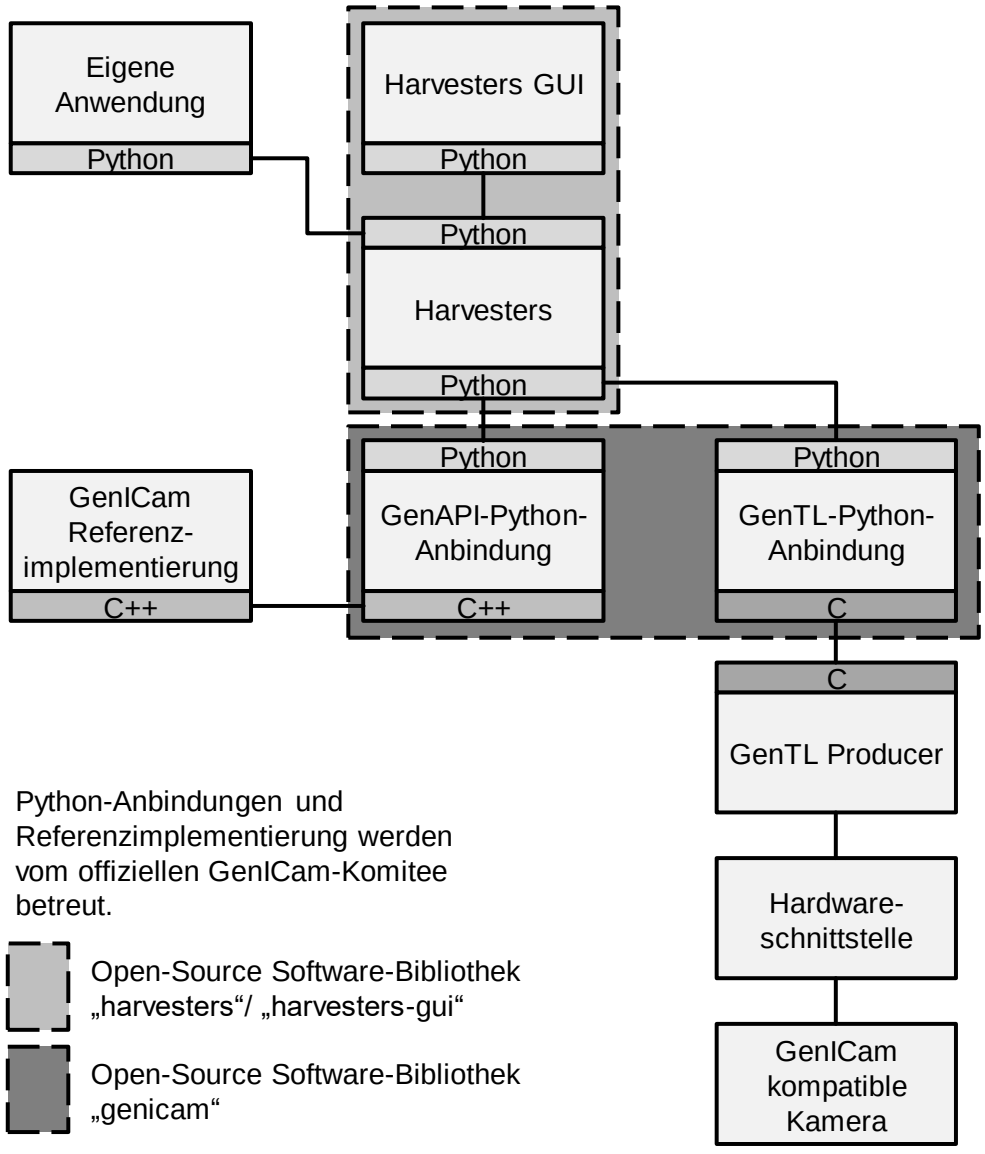


Abb. 5.1: Aufbau einer Kameraverbindung in Python mit den Modulen Harvesters und Genicam zu einer GenICam kompatiblen Kamera (Bildquelle: [The20a])

Da mit der Software-Bibliothek Harvesters Kameras beliebiger Hersteller, die entweder GigE Vision oder USB3 Vision als Hardwareschnittstelle verwenden, unterstützt werden, wird Harvesters zur Implementierung des Teilprozesses Bildaufnahme verwendet. Wie oben bereits beschrieben, wird dazu immer ein GenTL Producer benötigt. GenTL Producer stellen die Kamerahersteller mit ihren Kameras in den SDK zur Verfügung. Damit die SDK und der Pfad zum darin befindlichen GenTL Producer nicht bei jedem Kameraherstellerwechsel angepasst werden muss, bietet es sich an eine SDK beziehungsweise einen GenTL Producer zu nutzen, der erstens mit GenICam-Kameras anderer Hersteller funktioniert, zweitens die Nutzung mit Kameras anderer Hersteller laut Lizenz nicht ausschließt und drittens ohne Lizenzgebühren kommerziell nutzbar ist. Die MATRIX VISION GmbH, Oppenweiler, bietet ihr mvIMPACT Acquire SDK [Mat20] unter diesen Bedingungen an. Mit dem SDK von Matrix Vision wird gleichzeitig auch wxPropView und mvIPConfigure installiert. Zur Kamerakonfiguration wird die GUI wxPropView verwendet. mvIPConfigure wird nur benötigt, falls es Probleme mit der IP-Adresse oder dem Treiber gibt.

Es wäre auch möglich zur Kamerakonfiguration eine GUI eines anderen Herstellers zu verwenden. Um jedoch die Anzahl an zu installierenden Programmen gering zu halten, kann auf wxPropView auch die Nutzung der Open-Source Software-Bibliothek „harvesters-gui“ [The20c] in Betracht gezogen werden, da damit sämtliche GenICam Kameras unterstützt werden sollen und die Software unter einer Apache Lizenz angeboten wird. Jedoch hat diese Bibliothek derzeit noch einige ungelöste Softwareprobleme durch Versionsinkompatibilitäten mit dem Hauptmodul Harvesters.

Cognex In-Sight Vision 2000-230

Da die Cognex Kamera eine intelligente Kamera ist (s. Kapitel 2.3.4 Kameras), wird für die Konfiguration das Programm Cognex In-Sight Explorer [Cog20b] benötigt und für die Bildaufnahme die auf der Kamera befindliche Software verwendet. Cognex bietet seine Hardware nur in Kombination mit den eigenen Softwarelösungen an. Daher gibt es keine anderen Anbieter für die Software der Kamera. Die Kamera ist unter anderem über das In-Sight Native Mode Protokoll und OPC UA ansteuerbar. Das In-Sight Native Mode Protokoll ist ein ASCII Protokoll des Herstellers Cognex, mit dem ein In-Sight Sensor über benutzerdefinierte Anwendungsprogramme angesteuert werden kann. Der American Standard Code for Information Interchange (ASCII) ist eine Zeichenkodierung, mit der codierte Zeichensätze über eine Telnet-Verbindung gesendet werden können. Die Verbindung wird über Ethernet aufgebaut. Das Native Mode Protokoll bietet zwei verschiedene Arten von Befehlen (engl. Commands): Basic und Extended Native Commands. Die Basic Native Commands sind grundlegende Befehle zur Kamerasteuerung und Bildaufnahme. Diese werden von allen Cognex Kamerasensoren unterstützt.

Die höherpreisigen Kamerasensoren ab der In-Sight 5705 und höheren Serien bieten zusätzlich die Extended Native Commands, die an die Softwareerweiterung Spreadsheet geknüpft sind. Die Basic Native Commands sind ausreichend, sofern die Cognex Kamera mit einer selbst programmierten Anwendung genutzt wird. Für die in dieser Arbeit genutzte Cognex Kamera steht die Softwareerweiterung Spreadsheet nicht zur Verfügung. [Cog20a]

Tab. 5.1: Übersicht über die vorgestellten Softwarelösungen zur Kamerakonfiguration und Bildaufnahme

Allied Vision Mako G-223C und Mako G-503C	**Cognex In-Sight Vision 2000-230**
Vimba inkl. Python-API Pymba	Cognex In-Sight Explorer (nur Kamerakonfiguration)
Harvesters und GenICam (nur Bildaufnahme)	Cognex Native Mode Protokoll mit Basic Native Commands (nur Bildaufnahme)
mvIMPACT Acquire SDK (nur Kamerakonfiguration und GenTL Producer)	

5.1.2 Trigger

Ein Trigger kann einerseits ereignisgesteuert oder kontinuierlich mit festem Zeitintervall auslösen. Letzteres kann für Produktionen verwendet werden, bei denen es einen kontinuierlichen Materialfluss gibt. Da hier nur ein festes Zeitintervall im Code berücksichtigt werden muss, wird diese Form des Bildauslösens im Code selbst integriert.

Für das ereignisgesteuerte Auslösen wird keine Software im Rahmen dieser Bachelorarbeit erarbeitet, da der dafür benötigte Sensor individuell auf den vorliegenden Prozess abgestimmt sein muss. Stattdessen wird eine Schnittstelle zur Übergabe des digitalen Auslösesignals definiert. Die Validierung des Gesamtsystems wird auf einem Skript mit IO-Link Sensorschnittstelle basieren. Dieses Skript nutzt die Schnittstelle für das digitale Auslösesignal, die in dieser Arbeit definiert wird, und wird vom DCC Kooperationspartner ia: industrial analytics GmbH, Aachen, zur Verfügung gestellt. IO-Link ermöglicht die Verbindung mit intelligenten Sensoren und Aktoren. Für ausführlichere Informationen sei auf *IO-Link* von Uffelmann, Wienzek et al. [UWJ18] verwiesen.

5.1.3 Labeln

Es gibt eine Vielzahl von Möglichkeiten das Labeln durchzuführen. Klassisch kann dies händisch ohne zusätzliche Programme, also beispielsweise mit dem Explorer in Windows, durchgeführt werden. Es existieren auch viele Open-Source Software-Bibliotheken für diesen Prozess. Beispiele hierfür sind „image-sorter2“ [AB20] oder „tkteach“ [SM19], die auf dem Softwareverwaltungssystem GitHub des Unternehmens GitHub, Inc., San Francisco, USA, frei verfügbar sind. Ebenfalls ist die Durchführung in den Benutzeroberflächen von Microsoft Custom Vision, Google AutoML und allen weiteren kommerziellen Trainingsumgebungen vor dem eigentlichen Training möglich.

5.1.4 Training

Nachfolgende Tabelle gibt eine Orientierung für Softwarelösungen, die für das Training verwendet werden können (s. Tab. 5.2). Dabei werden drei verschiedene Arten an Trainingsumgebungen unterscheiden. Zum einen existieren die klassischen Open-Source Software-Bibliotheken, die einem Programmierer viele Freiheitsgrade lassen, aber auch Fachkenntnisse für die Umsetzung verlangen. Denn das Design der Algorithmen und die Hyperparameteroptimierung erfordern sehr viel Zeit und zahlreiche Versuche durch erfahrene Softwareingenieure und Datenwissenschaftler. Beispielsweise hat ein typisches 10-schichtiges Netzwerk etwa 10^{10} Kandidaten für das Design des KNN, die möglich wären. Aus diesem Grund entwickelte sich das automatisierte maschinelle Lernen, das häufig als AutoML abgekürzt wird. Dabei übernimmt AutoML die Auswahl des Algorithmus und die Hyperparameteroptimierung. Das ist besonders für Anwender ohne spezifische Erfahrung im Programmieren und Machine Learning hilfreich. Neben den kommerziellen Anbietern dieser AutoML Lösungen, entwickeln sich auch die Open-Source Bibliotheken in diesem Bereich immer weiter [FKE+19]. Die kommerziellen Anbieter bieten für erste Trainings kostenlose Testumfänge an und erfüllen damit noch die Anforderungen der Bachelorarbeit. Ob diese ausreichend sind, hängt von dem jeweiligen Anwendungsfall ab. Mit begrenztem Training wird allerdings eine geringere Vorhersagegenauigkeit erreicht. [Gan18; LZ17; Lub20]

Da im Rahmen dieser Bachelorarbeit das Training des KNN nicht detailliert betrachtet wird, wird auf eine ausführlichere Beschreibung der Softwareprodukte verzichtet. Tab. 5.2 zeigt jeweils drei Beispiele für populäre Softwarelösungen für die drei oben beschriebenen Kategorien. Aufgrund der stetigen Weiterentwicklung der Dienste und Bibliotheken ist dies jedoch nur eine Auswahl und Momentaufnahme. [War18; Klo19; Ung20]

Tab. 5.2: Übersicht zu Softwarelösungen für das Training von KNN

Klassische Open-Source Software-Bibliotheken für Training	Kommerzielle Anbieter von Plattformen für automatisiertes Training	Open-Source Software-Bibliotheken für automatisiertes Training
TensorFlow [Goo20c]	AutoML Vision (Google LLC, Mountain View, USA) [Goo20b]	Auto-WEKA [KTH+17]
Keras [Cho20]	Custom Vision (Microsoft Corporation, Redmond, USA) [Mic20]	Auto-sklearn [Mac19]
PyTorch [Pyt20]	AutoAI (International Business Machines Corporation, Armonk, USA) [Ibm20]	TPOT [WON+20]

5.1.5 Inferenz

Da die Inferenzplattformen für neuronale Netze als Server agieren, um Inferenzanfragen zu bearbeiten und die Ergebnisse dann an den Client zurückzusenden, wird die Inferenz im Rahmen der Bachelorarbeit mit einer Client-Server-Architektur umgesetzt. Daher muss zwischen der Plattform zur Inferenz, dem Server, und der Anforderung zur Inferenz aus der eigenen Applikation heraus, dem Client, unterschieden werden.

Für die Umsetzung des Servers existieren ebenfalls einige frei verfügbare Open-Source Software-Bibliotheken, die von verschiedenen Firmen im Hintergrund unterstützt werden. Drei Beispiele werden nachfolgend aufgezeigt. Alle drei wurden speziell für die Bereitstellung von Deep-Learning-Modellen in Produktionsumgebungen entwickelt:

- **TensorFlow Serving** [Goo20a] ist ein anpassbares, leistungsstarkes System zur Bereitstellung der Inferenz, das von Google entwickelt wurde. TensorFlow Serving bietet eine sofort einsatzbereite Integration mit TensorFlow Modellen, kann aber erweitert werden, um andere Arten von Modellen und Daten zu bedienen. Ebenfalls existieren andere Open-Source Software-Bibliotheken, die auf TensorFlow Serving aufbauen, um andere Modellarten zu unterstützen. TensorFlow Serving kann auf CPU, GPU und TPU ausgeführt werden und unterstützt gRPC und REST als Kommunikationsschnittstelle zwischen Client und Server [Goo20d].

- **NVIDIA Triton Inference Server** [Nvi20a] und **Intel OpenVINO Toolkit** [Int20b] ermöglichen beide die Nutzung einer Vielzahl an verschiedenen Modellarten (u. a. TensorFlow, PyTorch, TensorRT Plan, Caffe, MXNet). Beide Open-Source Bibliotheken bieten Methoden zur Modelloptimierung und anschließenden Inferenz an. Die Optimierung ist dabei auf die jeweilige Hardware von NVIDIA beziehungsweise Intel ausgelegt. NVIDIA Triton Inference Server unterstützt CPU- und GPU-basierte Inferenz sowie gRPC als Schnittstelle zwischen Client und Server [Nvi20b]. Intel OpenVINO kann auf CPU, GPU, FPGA und VPU ausgeführt werden und bietet als Schnittstelle gRPC und Rest [Int20a].

5.1.6 Zusammenfassung und Schlussfolgerung

Jeder Teilprozess des IBV-Systems wurde in Kapitel 5.1 adressiert. Die vorgestellten Softwarelösungen und Open-Source Bibliotheken der Teilprozesse lassen sich in die Gesamtprozessübersicht, die in Kapitel 4.4 (s. Abb. 4.4) vorgestellt wurde, einordnen. In der Darstellung wird zwischen drei Arten von Softwareprodukten unterschieden (s. Abb. 5.2):

- **SDK des Herstellers / kommerzielle Anbieter:** Dieser Kategorie werden alle SDK der Kamerahersteller inklusive der Schnittstellen sowie die kommerziellen Anbieter von Trainingsumgebungen für KNN mit begrenztem kostenfreien Trainingsumfang zugeordnet.
- **Open-Source Software-Bibliothek:** Die Software-Bibliotheken in dieser Kategorie stehen entweder unter MIT-Lizenz (Pymba), BSD-3-Lizenz (NVIDIA Triton Inference Server) oder Apache-2.0-Lizenz (alle anderen). Das bedeutet, dass alle Bibliotheken sowohl privat als auch kommerziell genutzt sowie verändert und verteilt werden dürfen. Damit können diese auch im Rahmen dieser Bachelorarbeit unter Angabe der verwendeten Bibliothek und dem Copyright-Vermerk genutzt werden. Alle Bibliotheken sind auf der Versionsverwaltungsplattform für Softwareprojekte GitHub verfügbar.
- **Selbstprogrammiert:** Hierunter ist sämtlicher Code zu verstehen, der selbst zur Nutzung von Schnittstellen (Bildaufnahme) oder auch für gesamte Teilprozesse (Bildspeicherung) sowie für die Verbindung der Teilprozesse zu einem Gesamtprozess geschrieben wird. Die Entwicklung findet in Kapitel 6 statt.

Aus den vorgestellten Softwareprodukten werden im nächsten Schritt jene ausgewählt, die für die zu entwickelnde Gesamtsoftware in dieser Bachelorarbeit genutzt werden. Zudem muss die Verbindung zwischen den einzelnen Prozessen hergestellt werden, um einen kontinuierlichen Inferenzbetrieb in einer Produktionslinie zu ermöglichen.

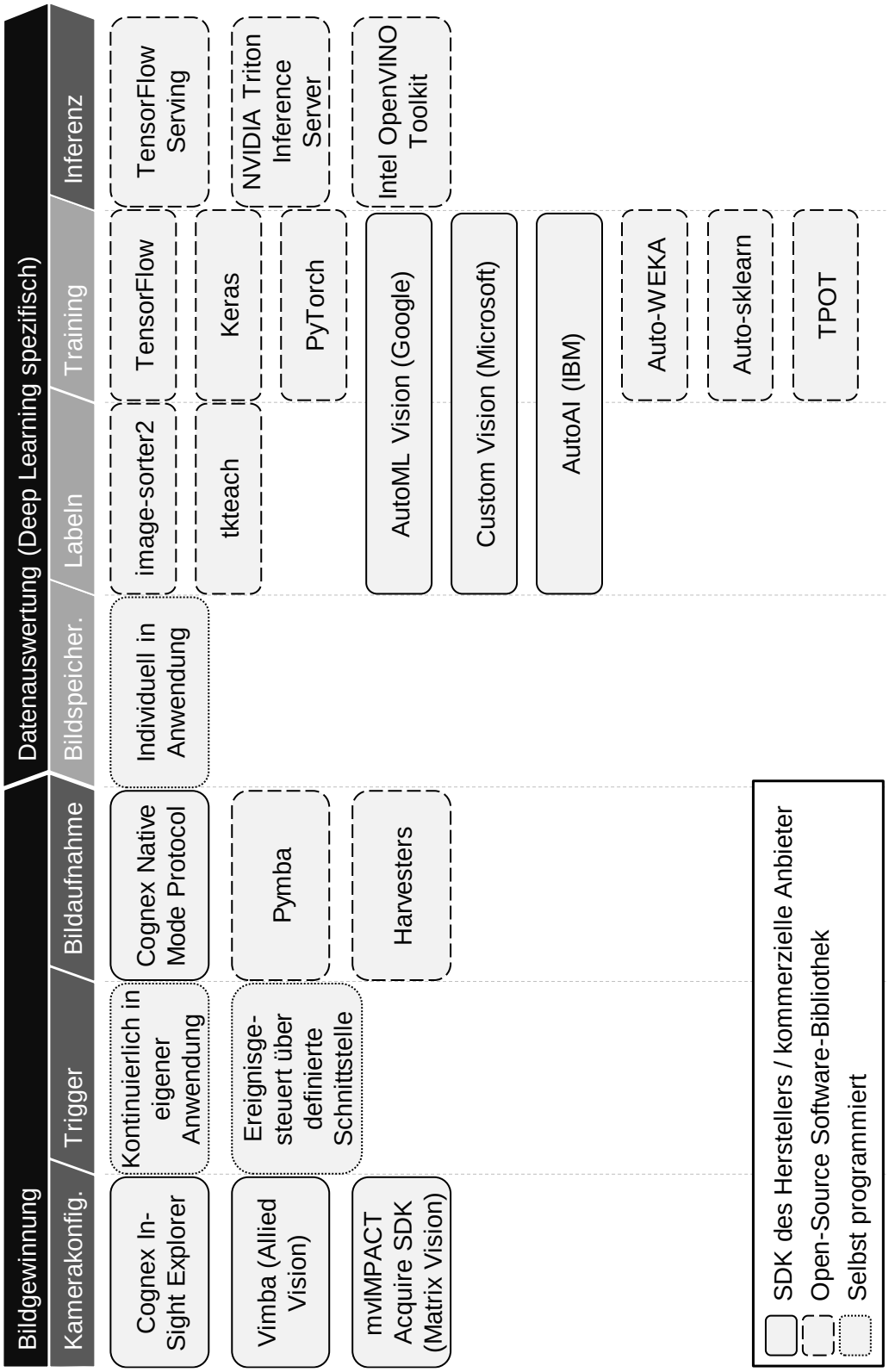


Abb. 5.2: Einordnung der vorgestellten Softwareprodukte in die Gesamtprozess-übersicht

5.2 Eingrenzung der Softwareprodukte

Da eine Verwendung und Implementierung aller vorgestellten Softwareprodukte aus Kapitel 5.1 den Rahmen dieser Bachelorarbeit übersteigen und nicht benötigte Redundanzen erzeugen würde, erfolgt nun eine Eingrenzung.

Für die **Kamerakonfiguration** und **Bildaufnahme** der Cognex Kamera werden der Cognex In-Sight Explorer und das Cognex Native Mode Protokoll genutzt, da diese alternativlos sind. Aufgrund der vielseitigeren Verwendungsmöglichkeiten (s. Kapitel 4.3.3) wird für die Bildaufnahme von GenICam-kompatiblen Kameras Harvesters zusammen mit dem GenTL Producer aus dem mvIMPACT Acquire SDK, das auch die GUI für die Konfiguration bereitstellt, genutzt. Unabhängig vom Kamerasystemhersteller wird für beide Fälle der Code zur Nutzung der API selbst geschrieben, um die passende Anbindung an eine System-Architektur zu ermöglichen.

Wie bereits in Kapitel 4.3.2 erklärt, wird der kontinuierliche **Trigger** im Code selbst programmiert. Für die ereignisgesteuerte Auslösung wird die Schnittstelle definiert und in der Validierung auf ein zu dieser Schnittstelle passendes Skript der ia: industrial analytics GmbH zurückgegriffen.

Die **Bildspeicherung** wird selbst im Gesamtprogramm implementiert und das **Labeln** erfolgt entweder händisch in der Ordnerstruktur des Betriebssystems (hier: Windows Explorer) oder in der Softwarelösung des kommerziellen Anbieters der Trainingsumgebung.

Da die Umsetzungen des **Trainings** und des **Inferenz-Servers** nicht Teil dieser Arbeit sind, werden sie als Annahmen von Seiten des DCC vorgegeben. Aufgrund einer bestehenden Partnerschaft mit Microsoft wird am DCC Custom Vision als Trainingsumgebung verwendet und ein Inferenz-Server auf Basis des Intel OpenVINO Toolkits mit gRPC Schnittstelle zur Verfügung gestellt. Der **Inferenz-Client** zur Nutzung des Servers wird zu einer System-Architektur passend selbst entwickelt.

5.3 Verbindung der Teilprozesse

Um die Teilprozesse zu einem Gesamtprozess für den kontinuierlichen Betrieb der Inferenz in der Produktionslinie zu verbinden, muss eine passende Systemarchitektur ausgewählt werden. Dabei sind insbesondere die Anforderungen aus Kapitel 4.1 für die Plug-and-Play Fähigkeit zu beachten. Darin ist definiert, dass die Bereitstellung durch eine automatische Installation von benötigten Treibern, Software-Bibliotheken und allgemeinen Einstellungen erfolgen und das System robust

gegenüber Fehlern sein soll. Zusätzlich ist das Ziel der Bachelorarbeit ein flexibles System bereitzustellen, das über standardisierte Schnittstellen verfügt.

5.3.1 Microservice-Architektur

Um diesen Anforderungen gerecht zu werden, wird eine Microservice-Architektur auf Basis der Containervirtualisierung-Software Docker gewählt (s. Kapitel 2.4 und 3.3). In einer solchen Architektur werden für die Teilprozesse eines Gesamtsystems einzelne Microservices parallel und unabhängig voneinander betrieben. Jeder Microservice erfüllt dabei eine bestimmte Funktion und kommuniziert über Protokolle mit den anderen Diensten. Die Vor- und Nachteile dieser Architektur sind in Tab. 5.3 aufgelistet.

Für den hiesigen Anwendungsfall überwiegen die Vorteile, da durch eine Microservice-Architektur die definierten Anforderungen abgedeckt werden und die Umsetzungskomplexität aufgrund der begrenzten Anzahl an Teilprozessen überschaubar ist. Im Folgenden wird daher eine Microservice-Architektur für die Umsetzung des IBV-Systems verwendet.

5.3.2 Ereignisgesteuerte Kommunikation zwischen Microservices

Für die Kommunikation kann grundsätzlich eine synchrone oder asynchrone Kommunikation genutzt werden. Bei der synchronen Kommunikation wartet der Nachrichtensender auf eine Antwort, bevor er die nächste Nachricht sendet. Im Gegensatz dazu wird bei der asynchronen Kommunikation die Nachricht gesendet, ohne auf die Antwort zu warten. Dieses Prinzip eignet sich besonders für verteilte Systeme. [Itz19] Das heißt mit einer asynchronen Kommunikation können die Services ihre Arbeit immer weiter verrichten, ohne auf die Rückmeldung eines anderen Prozesses zu warten, wie es in einer Client-Server-Architektur der Fall ist. Diese Form der Kommunikation wird als ereignisgesteuert bezeichnet, bei der ein Nachrichtenbroker mit Publisher/Subscriber-Pattern benötigt wird. Das heißt, dass Microservices an diesen Broker Ereignisnachrichten veröffentlichen und sich zum Empfang von neuen Nachrichten anmelden können (s. Abb. 5.3). Der Broker übernimmt die Verteilung der Nachrichten. Das erhöht die Effizienz des Gesamtprozesses und isoliert etwaige Probleme beim Empfänger, da der Sender davon nicht betroffen ist. [Jol18]

Tab. 5.3: Vor- und Nachteile einer Microservice-Architektur

Vorteile [LWR20; Ric15; Ven17; New15]	**Nachteile** [LWR20; Ric15]
Die Microservices können unabhängig entworfen, entwickelt und bereitgestellt werden. Dadurch können auch Updates einfacher durchgeführt werden. Die Flexibilität und die Geschwindigkeit der Entwicklung werden erhöht.	Im Gegensatz zu einer monolithischen Architektur, in der die Software „aus einem Stück“ besteht und direkt zusammenhängt, erfordern Microservice-Architekturen einen höheren Einsatz von globalen Ressourcen (Gesamtspeicher, Laufwerk und Netzwerk). Die dadurch entstehenden Kosten sind jedoch im Vergleich zu denen einer Skalierung und Weiterentwicklung einer monolithischen Anwendung zu vernachlässigen.
Zusätzlich lassen sich die einzelnen Dienste leichter in anderen Anwendungen wiederverwenden und sind durch ihre geschlossene Form einfach verständlich für Entwickler.	Durch die verteilte Anwendung muss zwischen den Diensten die Kommunikation durch geeignete Protokolle sichergestellt werden. Dies erfordert einen höheren Aufwand seitens der reinen Programmierung. Im Vergleich zum Erweitern oder Ändern von großen monolithischen Strukturen ist dieser Aufwand als gering einzuschätzen.
Tritt in einem Service ein Fehler auf, so ist nur dieser betroffen und kann isoliert von den anderen Microservices gedebuggt, geupdated und neugestartet werden. Das vereinfacht die Fehlerbehebung und erhöht die gesamte Stabilität des Systems, da alle anderen Microservices normal weiterlaufen können.	
Keine langfristige Bindung an eine Technologie oder ein Framework, da die Dienste unabhängig sind und somit jederzeit ein Dienst, der andere Technologien nutzt, isoliert neu entwickelt und bereitgestellt werden kann.	

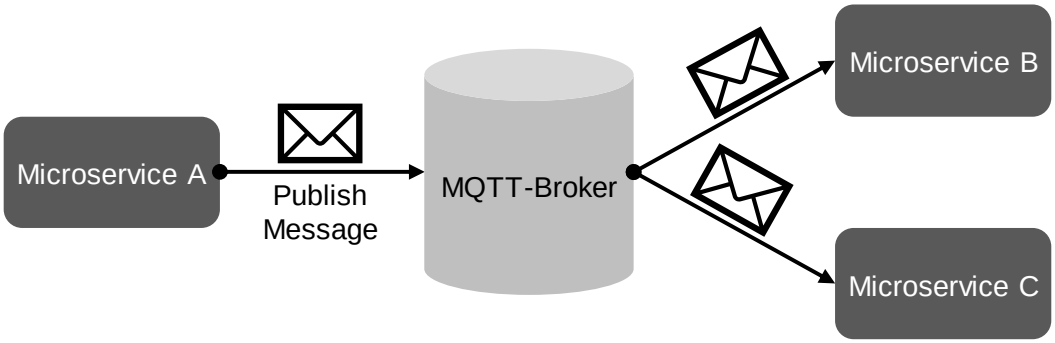


Abb. 5.3: Ereignisgesteuerte Kommunikation zwischen Microservices mit Publisher/Subscriber-Pattern (i. A. a. [LWR20])

Für die Umsetzung dieses Publisher/Subscriber-Pattern zur ereignisgesteuerten Kommunikation wird in dieser Bachelorarbeit MQTT verwendet. Die Funktionsweise und die Vorteile des MQTT-Protokolls wurden bereits in Kapitel 3.2.1 erläutert.

5.4 Ableitung der Microservices

Mit der Festlegung auf eine Microservice-Architektur, den Teilprozessbeschreibungen aus Kapitel 4.3 und den ausgewählten Softwareprodukten aus Kapitel 5.2 lassen sich nun die Microservices ausdefinieren. Diese dienen hauptsächlich der Einbindung der ausgewählten Softwarelösungen in die Systemarchitektur. Dazu wird im Folgenden abgeleitet, für welche Teilprozesse Microservices entwickelt werden müssen:

- Da die **Kamerakonfiguration** vollständig mit einer Herstellersoftware und außerhalb der Gesamtsoftware gemacht wird, wird hierfür kein Microservice benötigt.
- Für den ereignisgesteuerten **Trigger** wird der erste initiale Microservice benötigt. Dieser wird nicht im Rahmen der Bachelorarbeit entwickelt. Die Schnittstelle zum Prozess ist der MQTT-Broker, auf dem das Auslösesignal abgelegt wird. Der kontinuierliche Trigger wird direkt in die Microservices der Bildaufnahme integriert.
- Für die **Bildaufnahme** werden zwei Microservices zur Nutzung der beiden API benötigt. Ein Microservice nutzt dafür das Cognex Native Mode Protocol (s. Kapitel 6.2.1). Für GenICam Kameras wird ein Microservice unter Nutzung von Harvesters umgesetzt (s. Kapitel 6.2.2). Es werden getrennte Microservices für die beiden Kameraarten entwickelt, da die benötigten

Treiber und Software-Bibliotheken sehr unterschiedlich sind und auf nicht benötigte Installationen verzichtet wird. Beide Microservices werden über MQTT-Broker mit dem System verbunden, um über MQTT eine Bildaufnahme auslösen zu können und die digitalen Bilddaten auf dem Broker abzulegen.

- Die **Bildspeicherung** wird als einzelner Microservice implementiert, der beliebig durch Zu- und Wegschalten des Dienstes beim MQTT-Broker an- und abgemeldet werden kann, um die Bilddaten lokal zu speichern (s. Kapitel 6.2.3).
- Das **Training** ist, wie in Kapitel 5.2 erklärt, nicht Teil dieser Arbeit.
- Der **Inferenz-Client** wird ebenfalls als Microservice programmiert. Als Schnittstelle zum **Inferenz-Server** dient, wie bereits beschrieben, gRPC in einer synchronen Client-Server-Kommunikation. Der Inferenz-Client bietet auch mittels Umgebungsvariable wählbar eine Basisimplementierung einer REST API, die ebenfalls zur Verbindung zu Inferenz-Servern genutzt werden kann. Der Inferenz-Client erhält die digitalen Bilddaten vom MQTT-Broker und sendet die Ergebnisse aus der Inferenz zurück an diesen (s. Kapitel 6.2.4).

Auf den Ablauf innerhalb der einzelnen Microservices und deren Entwicklung wird in den verwiesenen Kapiteln detaillierter eingegangen. Abb. 5.4 zeigt alle erwähnten Microservices. Die Microservices sind in Prozessreihenfolge während des Inferenzbetriebs dargestellt. Die lokale Bildspeicherung und der Inferenz-Client werden parallel ausgeführt. Es wird zwischen dem Prozess mit Cognex und GenICam Kamera unterschieden. Wird ein kontinuierlicher Trigger verwendet, wird der Microservice Trigger nicht benötigt. Die Bildspeicherung ist optional zu- und wegschaltbar.

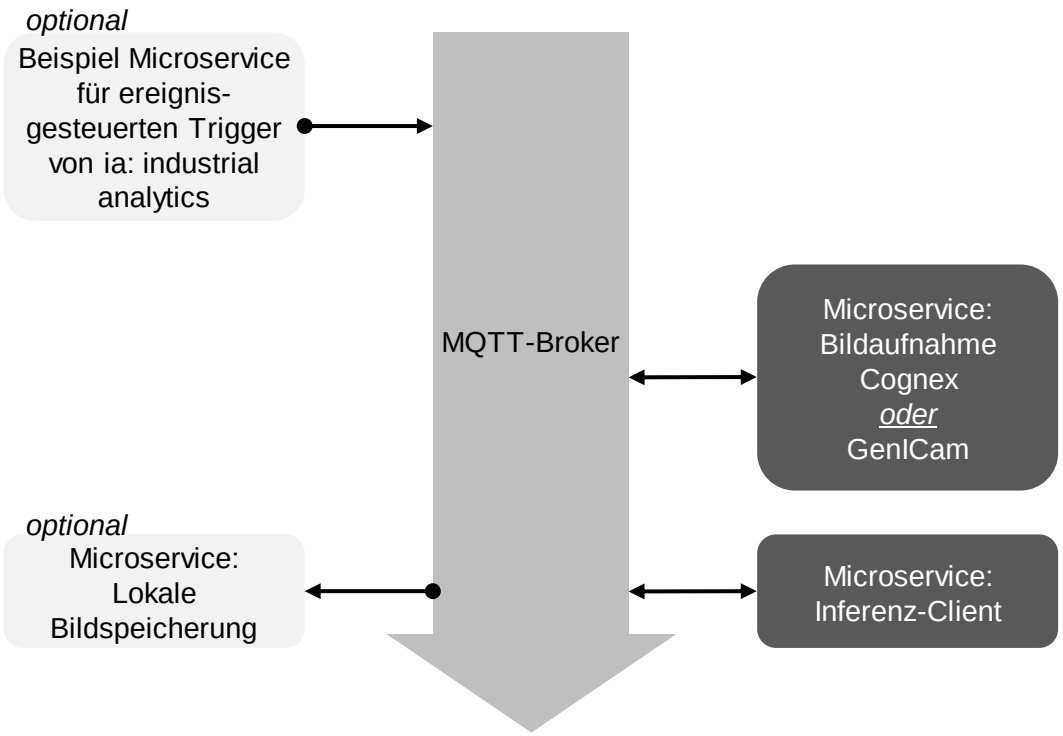


Abb. 5.4: Microservices im Inferenzbetrieb dargestellt in Prozessreihenfolge

5.5 Zusammenfassung der Ergebnisse auf Prozessebene

Wie im Kapitel 5.3 beschrieben und begründet, wird für die Software des IBV-Systems eine Microservice-Architektur mit ereignisgesteuerter MQTT-Kommunikation verwendet. Zusammen mit der Eingrenzung der Softwareprodukte aus Kapitel 5.2 und der Microservices aus Kapitel 5.4 ergibt sich damit eine neue Übersicht für den Gesamtprozess auf Basis von Abb. 5.2 und den in Kapitel 5.1.6 eingeführten Kategorien. In Abb. 5.5 sind alle Softwarelösungen und Microservices sowie deren Kommunikationsschnittstellen und der Broker dargestellt.

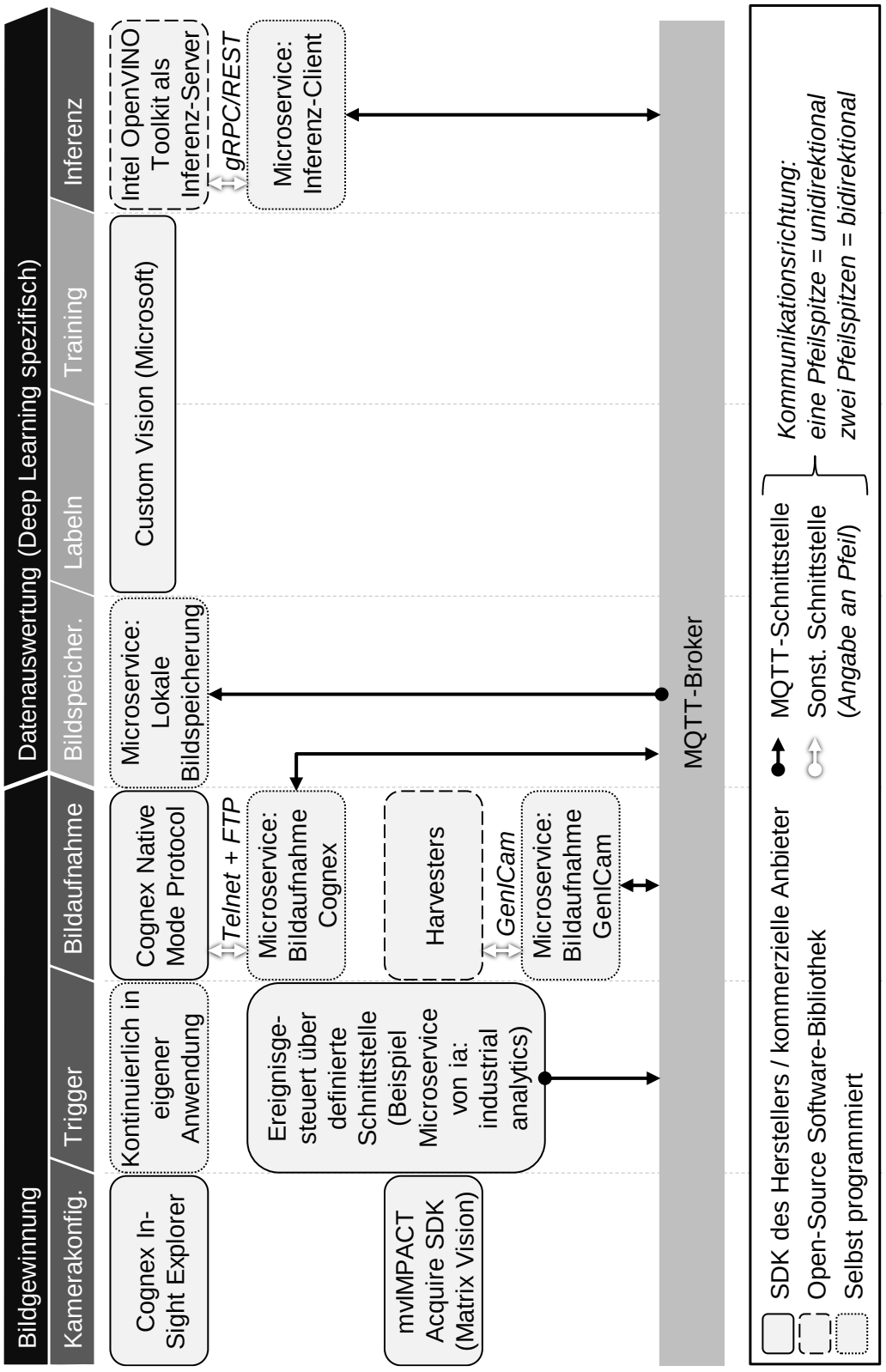


Abb. 5.5: Darstellung der Systemarchitektur inklusive der verwendeten Softwarelösungen und Schnittstellen als Gesamtprozessübersicht

6 Entwicklung der Microservices

Die in Kapitel 5.4 abgeleiteten Microservices werden nun entwickelt und implementiert. Zunächst wird hierzu auf den generellen Aufbau und die gemeinsamen Eigenschaften eingegangen (s. Kapitel 6.1). Für die Entwicklung der Microservices wird auf Prozessflussdiagramme zurückgegriffen, um die Abläufe im Microservice funktional zu beschreiben. In diesen werden verschiedene Symbole in Anlehnung an die ISO 5807 verwendet [ISO5807]. Die Erklärung der Symbole (s. Anhang 10.5) sowie die Prozessflussidagramme (s. Anhang 10.6.1, 10.7.1, 10.8.1, 10.9.1) sind im Anhang an diese Bachelorarbeit beigefügt. In der Definitionsphase der Anforderungen wurden diese mit den Stakeholdern des Entwicklungsprojekts abgestimmt. In Kapitel 6.2.1 bis 6.2.4 wird jeweils einer der Microservices und dessen Abläufe und Eigenschaften betrachtet.

6.1 Eigenschaften aller Microservices

Der Aufbau aller Microservices und Skripte ist weitestgehend ähnlich gestaltet, um anderen Programmierern eine schnelle Einarbeitung zu ermöglichen. Durch die Module mit Klassen wird eine Verwendung der Module außerhalb dieses Projekts vereinfacht. Zudem bietet sich eine einfache Möglichkeit weitere Klassen hinzuzufügen (beispielsweise weitere Kameraarten im Modul „cameras“ oder andere Triggerarten im Modul „trigger“).

In jedem Microservice wird die main.py Datei ausgeführt. In dieser werden die Umgebungsvariablen, die für diesen Dienst relevant sind, und die benötigten Software-Bibliotheken importiert. Basierend auf den Umgebungsvariablen werden danach Objekte der benötigten Klassen aus anderen Modulen instanziiert und deren Funktionalitäten aufgerufen. In den Kameraklassen sind die Kommunikationsaufrufe der Kameraschnittstellen in try-except Ausnahmebehandlungen eingebettet, um diese bei Verbindungsabbrüchen automatisch wiederherzustellen. Bei Fehlern, die aus einer falschen Eingabe von Umgebungsvariablen entstehen, bricht der Container ab und blendet dem Nutzer eine Fehlermeldung und Problemlösungsansätze in der Logdatei des Containers ein. Alle weiteren Fehler werden durch einen automatischen Neustart des Microservices probiert zu beheben, bis der Nutzer eingreift.

Für die Konfiguration der Microservices können verschiedene Umgebungsvariablen durch den Nutzer in einer Umgebungsdatei festgelegt werden. Dazu zählen beispielsweise Einstellungen für die Kamera, IP-Adressen und Ports. Nachfolgend werden die Umgebungsvariablen nicht separat im Text erklärt, da die Namen selbsterklärend gewählt wurden und alle Umgebungsvariablen alphabetisch mit

englischer Bezeichnung im Anhang 10.2 mit einer kurzen Beschreibung, einstellbaren Werten und deren Bedeutung sowie dem Standardwert aufgeführt sind. Umgebungsvariablen sind als solche aufgrund ihrer Kapitälchen-Schreibweise im Text zu erkennen (Bsp.: MQTT-PORT). Die Umgebungsvariablen müssen durch den Nutzer vor der Inbetriebnahme geprüft und entsprechend auf den Anwendungsfall hin angepasst werden.

Alle genannten Module, Klassen, Skripte und Funktionen, die im Anhang zu finden sind, sind fortan durch eine kursive Schreibweise hervorgehoben. Im Anhang 10.4 sind alle Module aufgeführt, die in die Microservices importiert werden. Die Urheberrechts- und Lizenzinformationen der verwendeten Open-Source Software-Bibliotheken (s. Anhang 10.14) und des SDK mvIMPACT Acquire von MATRIX VISION (s. Anhang 10.15) sind auf Englisch im Anhang aufgeführt.

Die Umsetzung eines MQTT-Brokers wird in dieser Arbeit nicht näher betrachtet, da dieser bereits auf dem zur Validierung verwendeten Rechner zur Verfügung steht. Mit Eclipse Mosquitto steht ein Open-Source MQTT-Broker frei zur Verfügung, auf dessen Anleitung und Dokumentation im Internet verwiesen sei [Lig20].

6.2 Betrachtung der einzelnen Microservices

Nachfolgend werden die vier Microservices, die für die Systemarchitektur selbst entwickelt werden, betrachtet. Für jeden wird der generelle Ablauf der Funktionsaufrufe sowie die Kommunikation mit dem MQTT-Broker beschrieben. Für weitere Details sind auch alle selbst entwickelten Skripte im Anhang auf Englisch auskommentiert beigefügt.

6.2.1 Bildaufnahme Cognex Kamera

Das Prozessflussdiagramm des Microservices zur Bildaufnahme mit einer Cognex Kamera befindet sich im Anhang 10.6.1. Dieser Microservice wird als image_acquisition_cognex innerhalb der Softwareentwicklung bezeichnet.

Ablauf der Funktionsaufrufe

Beim Aufruf des Skripts *main.py* (s. Anhang 10.6.2) werden zunächst alle benötigten Module und Umgebungsvariablen (s. Anhang 10.2.2) importiert. Anschließend wird eine Instanz der Klasse *Cognex* aus dem Modul *cameras* (s. Anhang 10.4.1) erzeugt, die der Bildaufnahme dient und dazu den nachfolgenden Trigger-Instanzen übergeben wird. Jedes Objekt dieser Kameraklasse verbindet sich direkt über Telnet und FTP mit der Cognex Kamera sowie über MQTT mit dem Broker. Die IP-ADRESSE DER COGNEX KAMERA kann im Cognex In-Sight Explorer im Menü-

punkt „Get Connected“ nach dem Verbinden eingesehen werden. Die Einstellungen und Bildverarbeitungsalgorithmen der Cognex Kamera werden als Jobs bezeichnet und auf der Kamera hinterlegt. Sofern eine JOB-ID übergeben wird, wird dieser Job geladen. Die Kamera muss für Bildaufnahmen im Online Modus sein. Ist dies nicht der Fall, wird ein Kameraneustart ausgeführt. Cognex Kameras sind nach dem Hochfahren standardmäßig immer im Online Modus, sofern dies nicht bewusst im In-Sight Explorer geändert wird. Der Neustart dauert mindestens 60 Sekunden.

Sofern ein kontinuierlicher TRIGGER mithilfe einer festen DURCHLAUFZEIT gesetzt wird, wird ein Objekt der Klasse *ContinuousTrigger* aus dem Modul *trigger* (s. Anhang 10.4.2) erzeugt. Darin wird eine Bildaufnahmeschleife ausgeführt, in der die Zeit gemessen wird. Durch die Berechnung einer künstlichen Pause wird die Prozesszeit gesteuert und damit die DURCHLAUFZEIT eingehalten. Ist die DURCHLAUFZEIT geringer als die Prozesszeit einer Schleife, wird eine Fehlermeldung erzeugt und abgebrochen.

Bei der Verwendung eines Triggers über MQTT wird stattdessen eine Instanz der Klasse *MqttTrigger* aus dem Modul *trigger* (s. Anhang 10.4.2) geschaffen. Diese Instanz abonniert die vom Nutzer einstellbare MQTT-TOPIC-TRIGGER beim MQTT-Broker, sodass bei jeder neuen veröffentlichten Nachricht unter diesem Topic eine neue Bildaufnahme ausgelöst wird. Es wird unabhängig vom Nachrichteninhalt immer eine neue Bildaufnahme ausgelöst. Das heißt, dass der Nachrichteninhalt grundsätzlich beliebig gewählt werden kann. In der Nachricht können aber auch Schlüsselwörter (engl. Keyword) genutzt werden. Die Nachricht muss dazu als Python-Dictionary im Datenformat JavaScript Object Notation (JSON) gesendet werden. Sofern das Keyword „timestamp“ enthalten ist, wird dieser Zeitstempel für die Berechnung der Zeit zwischen Auslösesignal und tatsächlicher Aufnahme verwendet. Ansonsten wird bei Eintreffen der Nachricht ein Zeitstempel erzeugt. Das hat aber den Nachteil, dass Schwankungen in der Übertragung vom Trigger-Microservice über den Broker zum Bildaufnahme-Dienst nicht berücksichtigt werden. Wenn das Keyword „job_id“ enthalten ist, wird geprüft, ob es eine andere Job-ID im Vergleich zur zuletzt verwendeten ist und falls ja wird der Job geladen. Diese Funktion ermöglicht das Prüfen verschiedener Objekte oder verschiedener Merkmale mit dem gleichen Inspektionssystem. Denn in jedem Job können verschiedene Kameraeinstellungen und Algorithmen hinterlegt werden. Das Keyword „conventional_image_processing“ kann genutzt werden, um auch konventionelle Bildverarbeitungsergebnisse zu erhalten. Auch wenn sich die vorliegende Bachelorarbeit auf Methoden des Deep Learnings beschränkt, wurde diese Funktion aufgrund der relativ einfachen Implementierung und der Möglichkeit, in zukünftigen Testszena-

rien konventionelle Bildverarbeitung gegen Methoden des Deep Learnings zu vergleichen, umgesetzt. Auch über die Umgebungsvariable kann die konventionelle Bildverarbeitung aktiviert werden. Die entsprechenden Verarbeitungsalgorithmen müssen dann im Cognex In-Sight Explorer im Job hinterlegt werden.

Unabhängig davon, ob ein kontinuierlicher oder ereignisgesteuerter MQTT-Trigger verwendet wird, wird in beiden Varianten die Funktion *get_image()* der Kamera-Instanz ausgeführt. Mit dieser Funktion wird per Cognex Native Command über Telnet eine Bildaufnahme ausgelöst und die Bilddatei über FTP zum Rechner übertragen. Die Bilddaten werden anschließend an den MQTT-Broker versendet. Sofern das System mit konventionellen Methoden genutzt wird, kann das Ergebnis der Bildverarbeitung ebenfalls direkt mitübertragen werden. Die Nachricht wird als Python-Dictionary im JSON-Format gesendet. Der Aufbau der Nachrichteninhalte wird nachfolgend unter Kommunikation mit dem MQTT-Broker vereinfacht beschrieben. Anhang 10.3 enthält eine Übersicht aller verwendeten Cognex Native Commands. Abb. 6.1 zeigt den vereinfachten Ablauf der Bildaufnahme und Übertragung zum Rechner mit einer Cognex Kamera. Nachdem die MQTT-Nachricht an den getrennt laufenden MQTT-Thread übergeben wurde, kann die nächste Bildaufnahme durch einen Trigger erfolgen.

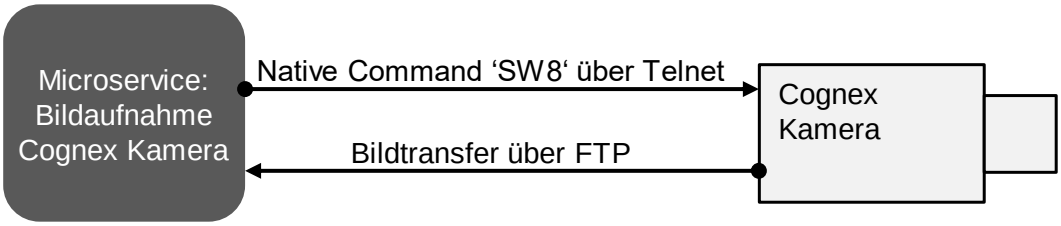


Abb. 6.1: Vereinfachte Darstellung der Kommunikation zur Bildaufnahme und Übertragung zwischen Microservice und Cognex Kamera

Kommunikation mit dem MQTT-Broker

In Abb. 6.2 ist die Kommunikation des Microservices mit dem MQTT-Broker dargestellt. Die Abbildung zeigt ein Beispiel mit ereignisgesteuertem Trigger. Zunächst abonniert der Dienst die MQTT-TOPIC-TRIGGER. Sobald der Broker einen neuen Trigger erhält, leitet er die Nachricht weiter. Diese Nachricht kann optional den Zeitstempel, eine Job-ID und eine Festlegung, ob konventionelle Methoden gerade verwendet werden, enthalten. Nach Ausführung des Microservices sendet dieser eine Nachricht unter dem MQTT-TOPIC-IMAGE an den Broker. Diese Nachricht enthält die SERIENNUMMER der Kamera, um bei Verwendung mehrerer Kameras die Nachrichten eindeutig zuordnen zu können, und einen neuen Zeitstempel, der den Zeitpunkt der Bildaufnahme beschreibt. Das JSON-Format, das bei MQTT-Nachrichten verwendet wird, unterstützt nur Byte-Arrays. Daher werden die Daten des Numpy-Arrays in Base64-Bytes enkodiert und anschließend in einen

String zum Versenden als Textnachricht dekodiert. Um beim Empfänger die Bilddaten wieder als Matrix aufzubauen, werden die Informationen Bildhöhe, Bildbreite und Bildkanalanzahl mitgesendet. Die konkrete Umsetzung ist im Code im Anhang beschrieben. Die Information zur Gesamtqualität (engl. Overall-Quality) ist zu diesem Zeitpunkt nur in der Nachricht enthalten, wenn konventionelle Bildverarbeitungsmethoden verwendet werden.

Bei einem kontinuierlichen Trigger würde der Microservice kein Topic beim Broker abonnieren und damit keine Nachricht von ihm erhalten, sondern nur die Nachricht mit den Bilddaten senden.

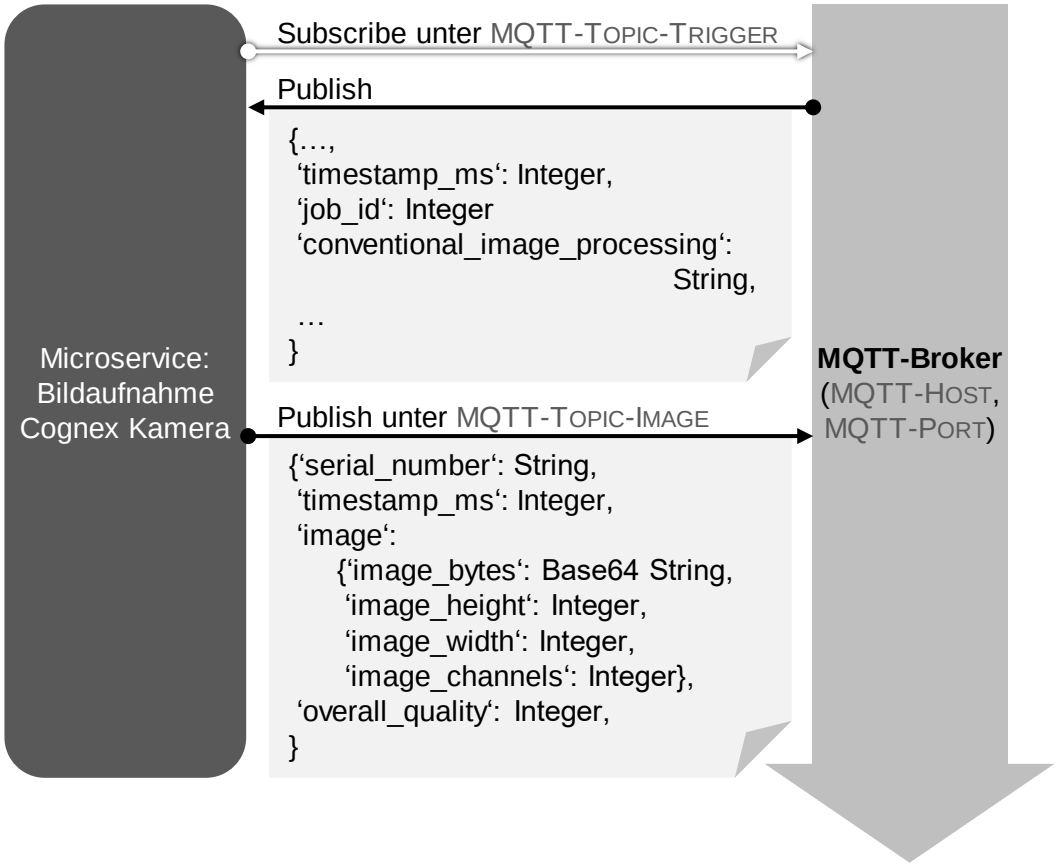


Abb. 6.2: Kommunikation des Microservices Bildaufnahme Cognex Kamera mit dem MQTT-Broker am Beispiel eines ereignisgesteuerten Triggers

Spracheinstellung Cognex In-Sight Explorer

Der Cognex In-Sight Explorer muss auf Englisch genutzt werden. Da die Cognex Native Mode Commands sprachsensitiv sind, kann eine Nutzung in Deutsch zu Kompatibilitätsproblemen führen.

6.2.2 Bildaufnahme GenICam Kamera

Das zugehörige Prozessflussdiagramm des GenICam Dienstes ist im Anhang 10.7.1 zu finden. Fortan wird auch image_acquisition_genicam als Bezeichnung für diesen Microservice genutzt.

Ablauf der Funktionsaufrufe

Die Bildaufnahme mit einer GenICam Kamera funktioniert vom Grundaufbau der Skripte und Funktionen her ähnlich zu der mit einer Cognex Kamera. Im ausgeführten Skript *main.py* (s. Anhang 10.7.2) wird stattdessen ein Objekt der Klasse *GenICam* (s. Anhang 10.4.1) als Instanz erzeugt. In dieser Klasse wird das Modul Harvesters genutzt, um Bildaufnahmen zu tätigen. Dazu wird zu Beginn ein Harvester-Objekt erzeugt, um alle Kameras im Netzwerk zu finden, die den hinterlegten GenTL-Producer unterstützen. Das sind in diesem Fall alle GigE Vision basierten Kameras. Das System ist derzeit auf die Nutzung einer Kamera im Netzwerk ausgelegt. Daher wird die erste erkannte Kamera verwendet. Zur Einrichtung der Kamera wird das durch den Nutzer hinterlegte User-Set, das in der Kamerakonfiguration erstellt wurde, geladen. Alternativ zum User-Set kann der Nutzer auch die wichtigsten Parameter selbst als Umgebungsvariablen hinterlegen (s. Anhang 10.2.3). Letzteres sollte nur genutzt werden, falls die Kamera die User-Set Verwendung nicht unterstützt, da dies kein verpflichtendes Feature des GenICam Standards ist. Neben den Einstellungen des Nutzers wird der Chunk Mode deaktiviert und die Anzahl an Bilder im Zwischenspeicher auf eins gesetzt, damit immer nur das aktuellste Bild abliegt. Wenn der Chunk Mode deaktiviert ist, werden dem Bilddatenstrom keine Chunk Daten angehangen. Das sind Metadaten über die Aufnahmeeinstellungen eines jeden Bildes. Mit aktiviertem Chunk Mode tritt ein Laufzeitfehler im Modul Genicam beim Nutzen der Kameras von Allied Vision auf, der sich nur durch Deaktivierung des Chunk Modes derzeit vermeiden lässt. Nach der Einrichtung wird die Kamera gestartet und liefert kontinuierlich mit der höchst möglichen Bildrate neue Aufnahmen. Diese Bilddaten werden in einem Zwischenspeicher abgelegt. Sobald ein Signal zum Aufnehmen eines neuen Bildes, wie bei der Cognex Kamera beschrieben, empfangen wird, lädt die Funktion *get_image()* das Bild aus dem Zwischenspeicher. Das dabei entstandene Bild wird anschließend je nach Einstellung entweder im Mono- oder RGB-Bilddatenformat, wie bei der Cognex Kamera, an den MQTT-Broker gesendet.

Kommunikation mit dem MQTT-Broker

Die Kommunikation zwischen dem Microservice für GenICam Kameras ist sehr ähnlich zu dem für Cognex Kameras. Daher sei auf die Beschreibung in Kapitel 6.2.1 (Kommunikation mit dem MQTT-Broker) verwiesen. Zur Veranschaulichung der Unterschiede dient Abb. 6.3. Die Cognex spezifischen Schlüsselwörter für die konventionellen Bildverarbeitungsmethoden fehlen hierin.

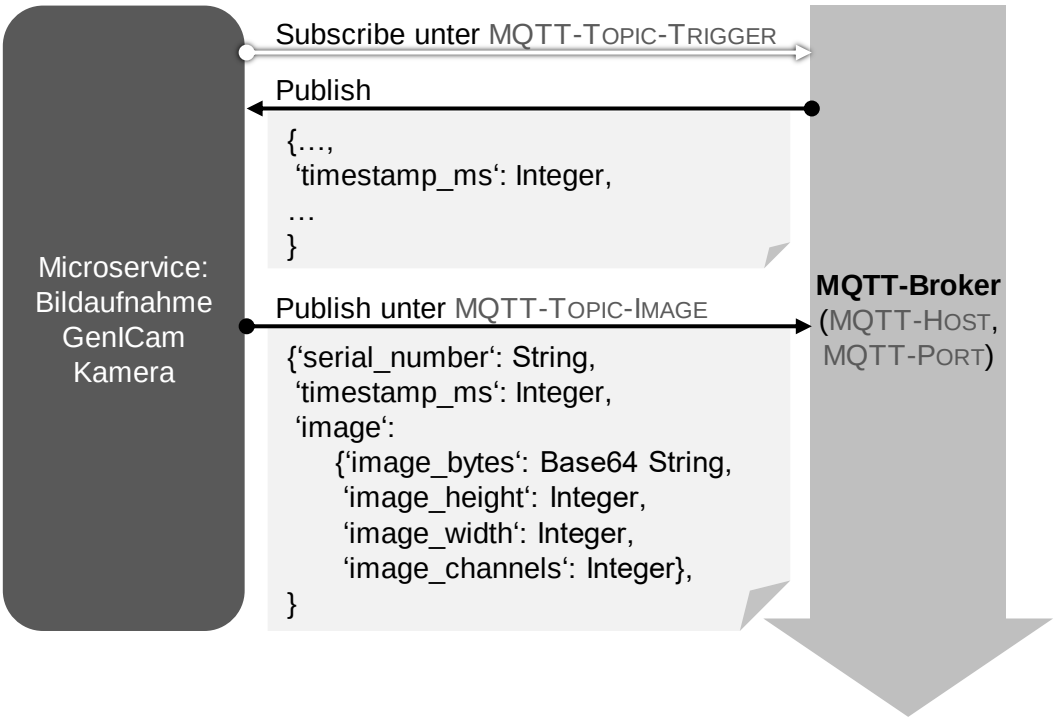


Abb. 6.3: Kommunikation des Microservices Bildaufnahme GenICam Kamera mit dem MQTT-Broker am Beispiel eines ereignisgesteuerten Triggers

6.2.3 Lokale Bildspeicherung

Anhang 10.8.1 enthält das Prozessflussdiagramm für den Microservice zur lokalen Bildspeicherung. Dieser Microservice wird als local_saving innerhalb der Softwareentwicklung bezeichnet.

Ablauf der Funktionsaufrufe

Zu Beginn der Ausführung werden auch bei diesem Dienst zunächst alle benötigten Bibliotheken und Umgebungsvariablen importiert. Alle Basis-Funktionalitäten werden im Modul *saving* (s. Anhang 10.4.3) durch die Klasse *LocalSaver* bereitgestellt. Die Klassenstruktur wurde an dieser Stelle trotz derzeit nur einer enthaltenen Klasse für einen konsistenten Aufbau der Microservices beibehalten. Auf Basis dieses Moduls können beispielsweise weitere Klassen geschrieben werden, die die Bilder nicht lokal, sondern in einer Datenbank speichern. Das Objekt der Klasse *LocalSaver* abonniert automatisch beim MQTT-Broker die MQTT-TOPIC-IMAGE. Mit der while-Schleife im Skript *main.py* (s. Anhang 10.8.2) wird sichergestellt, dass immer, sobald der Broker eine neue Nachricht mit Bilddaten weiterleitet, das Bild mit dem einstellbaren BILDNAMEN und Zeitstempel aus der Nachricht im Docker-Container gespeichert wird.

Um die Bilder auf dem Host-PC zu speichern, wird ein Docker-Volume verwendet. Ein Docker-Volume ist ein Ordner auf dem Host-PC, dass vom Docker-Container gemanagt und zwischen Docker-Container und Host-PC gemeinsam genutzt wird. Dadurch bleiben die Bilder auch nach Beenden des Docker-Containers erhalten. Sofern der Speicherort auf dem Host-PC geändert werden muss, ist diese Einstellung direkt in der Docker-Compose Datei (s. Kapitel 6.3) vorzunehmen. Diese Einstellung ist nicht über Umgebungsvariablen möglich.

Kommunikation mit dem MQTT-Broker

Bei der Kommunikation mit dem MQTT-Broker abonniert der Dienst zunächst die MQTT-TOPIC-IMAGE (s. Abb. 6.4). Sobald der Microservice eine Nachricht vom Broker erhält, werden die Informationen der Bildaufnahme aus dem Base64-Bytestring zurück in Bytes umkodiert, um anschließend die Matrix mit den Bilddaten als Numpy-Array zu erzeugen. Dieser Prozess stellt damit das genaue Gegenteil des Kodierungsprozesses bei der Bildaufnahme dar. Dieser Prozess muss auch bei allen anderen Services, die die Bilddaten verwenden, wie der Inferenz-Client, durchgeführt werden. Für zukünftige Entwicklungen kann dieser Code aus den Skripten im Anhang kopiert werden.

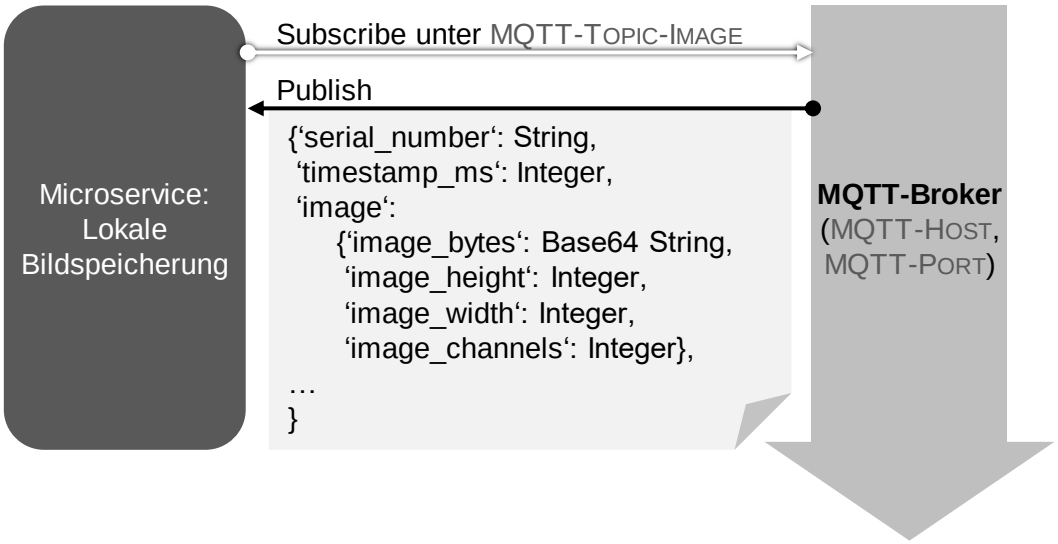


Abb. 6.4: Kommunikation des Microservices Lokale Bildspeicherung mit dem MQTT-Broker

6.2.4 Inferenz-Client

Das Prozessflussdiagramm des Inferenz-Clients ist im Anhang 10.9.1 dargestellt. In der Entwicklung wird dieser als inference_client bezeichnet.

Ablauf der Funktionsaufrufe

Auch bei diesem Dienst wird mit dem Ausführen des Skripts *main.py* (s. Anhang 10.9.2) der Prozess gestartet. Nach dem Import aller benötigten Bibliotheken und Umgebungsvariablen findet zunächst die Fallunterscheidung zwischen REST-API- und gRPC-basierter Kommunikation statt. Je nach Schnittstelle wird ein Client der jeweiligen Klasse *RestInferenceClient* oder *GrpcInferenceClient* aus dem Modul *inference* (s. Anhang 10.4.4) instanziiert. Dadurch wird die Schnittstelle automatisch konfiguriert und der Microservice über MQTT verbunden. Die nachfolgende while-Schleife im main-Skript stellt wieder sicher, dass der Prozess endlos läuft, um Bilder an den Inferenz-Server zu senden. Beide Schnittstellenvarianten funktionieren sehr ähnlich, da sie einerseits gemeinsame Methoden und anderseits eine abstrakte Methode von der Elternklasse, die jeweils individuell ausdefiniert werden muss, erben. In dieser Elternklasse sind die Basisfunktionalitäten zum Erhalt und Versand von MQTT-Nachrichten enthalten. Individuell ist dagegen die Kommunikation mit dem Inferenz-Server. Beim *RestInferenceClient* wird das kodierte Bild im JSON-Format und HTTP-Befehlen an den Server gesendet, um das Ergebnis zurückzuerhalten (s. Kapitel 3.2.4). Beim *GrpcInferenceClient* werden die Bilddaten mit Protobuf serialisiert und Funktionen aus der TensorFlow Bibliothek [Goo20d] verwendet (s. Kapitel 3.2.3). Weitere Informationen über die REST API und gRPC sind in den genannten Kapiteln und in den auskommentierten Skripten zu finden. Nachdem die MQTT-Nachricht mit den Ergebnissen der Inferenz an den getrennt laufenden MQTT-Thread übergeben wurde, ist die nächste Inferenz möglich.

Kommunikation mit dem MQTT-Broker

Der Inferenz-Client erhält vom MQTT-Broker aufgrund des Abonnements alle neuen Nachrichten der Bildaufnahmeprozesse unter dem MQTT-TOPIC-IMAGE. Die darin enthaltenen Bilddaten nutzt der Dienst zunächst, um diese zurück in einen Numpy-Array zu kodieren (s. Kommunikation mit dem MQTT-Broker Kapitel 6.2.3). Nach der Inferenz werden der empfangenen Nachricht die neuen Informationen hinzugefügt. Dazu gehören die Wahrscheinlichkeiten der einzelnen Labels und die daraus resultierende Klasse, also das Label mit der höchsten Wahrscheinlichkeit, sowie eine Beurteilung der Gesamtqualität in Form eines Integer-Werts, der mit 0 eine schlechte und mit 1 eine gute Qualität anzeigt. Diese Nachricht wird an den MQTT-Broker unter der MQTT-TOPIC-RESULTS veröffentlicht. Diese Nachricht enthält die Informationen für die Folgeprozesse, um beispielsweise die Qualität auf einem Dashboard zu visualisieren oder eine Signallampe zu steuern. Die Nachricht kann jederzeit, um weitere Informationen direkt im Code ergänzt werden.

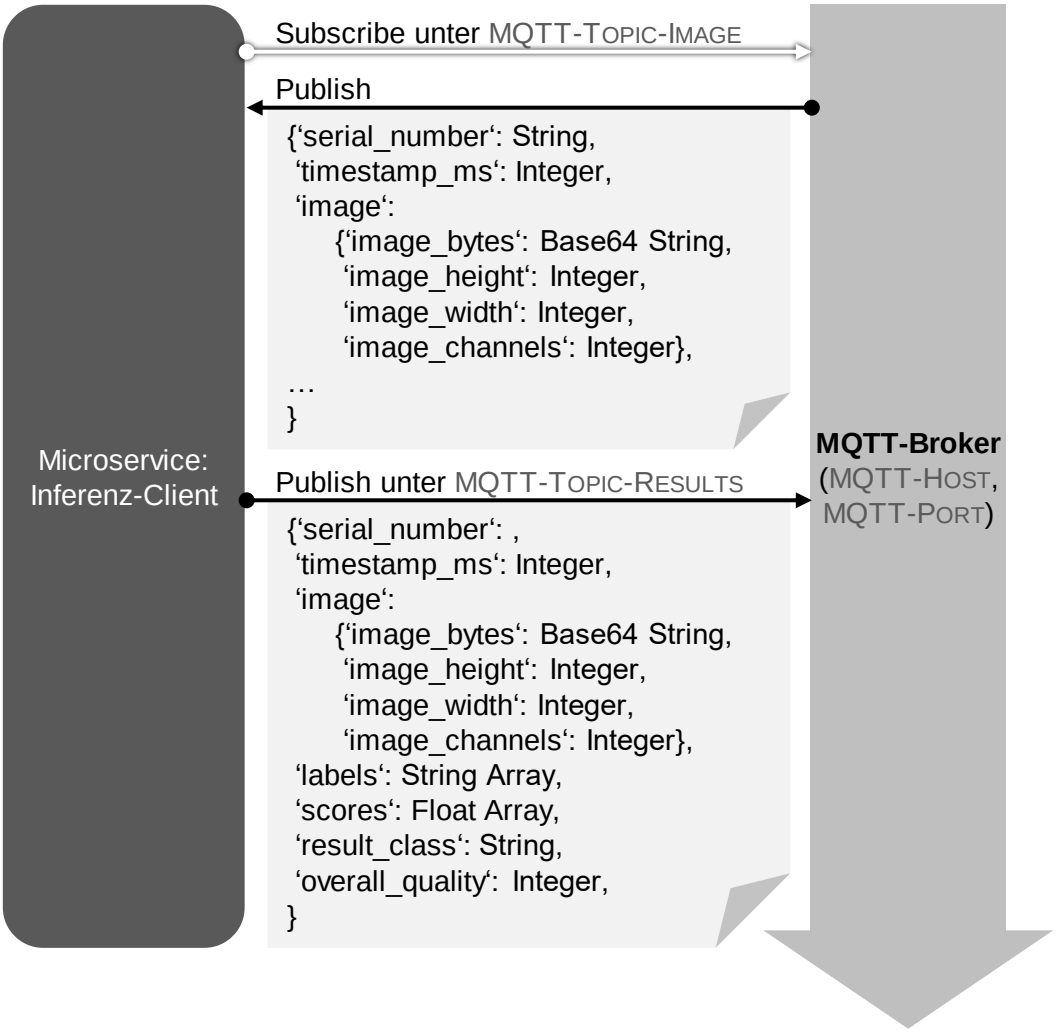


Abb. 6.5: Kommunikation des Microservices Inferenz-Client mit dem MQTT-Broker

6.3 Systeminbetriebnahme und -management mit Docker-Compose

Zum Ausführen der Microservices wird Docker-Compose von Docker (s. Kapitel 3.3) verwendet. Hinweise zur Installation und zur Ausführung der Microservices mit Docker-Compose sind im Anhang 10.16 beschrieben. Für die Installation ist stets eine Internetverbindung erforderlich, da die benötigten Software-Bibliotheken und Installationsdateien heruntergeladen werden müssen. Da zum Betrieb einer GenICam Kamera umfangreichere Installationen notwendig sind, wurden getrennte Docker-Compose Dateien und main-Skripte für Cognex (s. Anhang 10.10.2) und GenICam (s. Anhang 10.11.2) Kameras entwickelt. Die benötigten Dateien können der Verzeichnisstruktur im Anhang entnommen werden (s. Anhang 10.10.1 für Cognex und 10.11.1 für GenICam). Um die Übersichtlichkeit in

der env-Datei, in der die Umgebungsvariablen sind, zu erhöhen, existiert je eine Datei für jeden Kameratyp (s. Anhang 10.13 für Cognex und 10.12 für GenICam). Bei der Ausführung der Docker-Compose Datei werden die Container der Microservices Bildaufnahme (Cognex oder GenICam), Bildspeicherung und Inferenz-Client inklusive aller benötigten Software-Bibliotheken, Treibern, Umgebungsvariablen und anderen Einstellungen der Laufzeitumgebung geladen. Durch Auskommentieren einzelner Services in der Docker-Compose Datei kann beispielsweise der Microservice Bildspeicherung zu und weggeschaltet werden. Für jeden Microservice sind die zu installierenden Bibliotheken in einem separaten Dokument (requirements.py) zusammengefasst, um stets nur die individuell benötigten Module zu installieren (s. Anhang 10.6.3, 10.7.3, 10.8.3, 10.9.3). Die zum Bauen benötigten Dockerfiles befinden sich in den Anhängen 10.6.4, 10.7.4, 10.8.4 und 10.9.4.

7 Validierung der Systemsoftware

Für die Validierung der Systemsoftware werden zwei Anwendungsfälle in der Modellproduktion für textile Armbänder am DCC Aachen betrachtet.

7.1 Druckkontrolle

Die erste Implementierung des Codes erfolgt zur Prüfung der Druckqualität des Stoffs (s. Abb. 7.1 oben). Die Hardwareauswahl und -installation wurde bereits außerhalb der Bachelorarbeit vorgenommen. Als Kamera wird die Mako G-503C von Allied Vision Technologies mit dauerhaft eingeschaltetem Ringlicht verwendet (s. Abb. 7.1 mitte). Die Prüfung findet geschützt vor äußeren Einflüssen in einem Gehäuse statt (s. Abb. 7.1 unten). Als Recheneinheit wird ein ia: factorycube der ia: industrial analytics GmbH verwendet. Da diese Recheneinheit keine PoE-Funktionalität aufweist, wird die Kamera über einen PoE-Injektor und Ethernet Kabel mit dem Rechner verbunden. Auf dem Rechner ist bereits ein MQTT-Broker implementiert. Da es sich um eine kontinuierliche Produktion handelt, wird ein Trigger gewählt, der nach einer festgelegten Durchlaufzeit auslöst. Die Kamera wurde bereits konfiguriert und ein neuronales Netz trainiert, das nun im Inferenz-Server zur Verfügung steht. Alle verwendeten Umgebungsvariablen für dieses Setup sind im Anhang 10.12 aufgeführt.

Nachdem die benötigten Skripte über eine SSH-Verbindung auf den Rechner kopiert sind, wird Docker installiert und im Arbeitsverzeichnis die Docker-Compose Datei ausgeführt. Alle weiteren Schritte geschehen automatisch. Die Dauer der Installation hängt von der Netzverfügbarkeit und der Leistung des Rechners ab. Aufgrund der verhältnismäßig langen Installation der mvIMPACT Acquire SDK dauerte der Prozess auf dem Karbon 300 Mini-Rechner des ia: factorycubes rund 10 Minuten. Nach der erfolgten Installation ist das System direkt einsatzbereit.

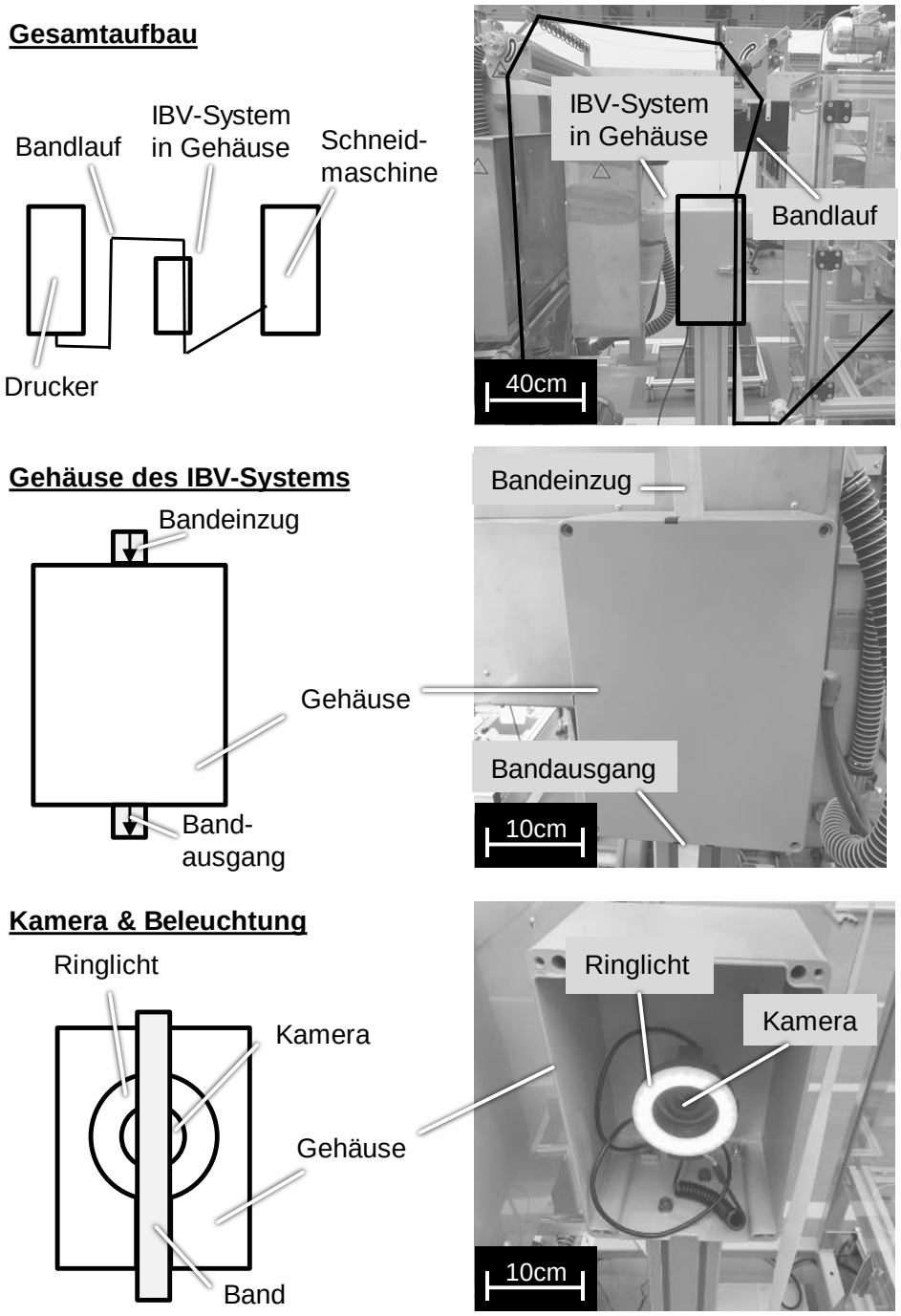


Abb. 7.1: Aufbau des Validierungsfalls Druckkontrolle

Das KNN klassifiziert in vier Klassen (s. Abb. 7.2):

1. Keine Fehler („qok")
2. Überlappungen der Druckaufträge („overlaps“)
3. Lücken zwischen den Druckaufträgen („gaps“)
4. Versatz der Druckposition („offset“)

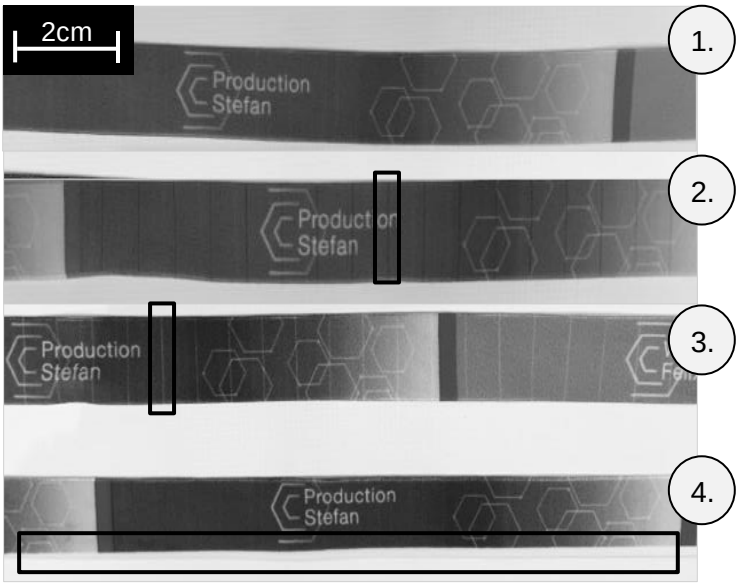


Abb. 7.2: Überblick über die Klassen und entsprechende Fehler

Ergebnis der Validierung

Der Docker-Container für die Bildaufnahme mit der GenICam Kamera startet ohne Probleme und beginnt auch die Bildaufnahme. Jedoch treten während des Programmablaufs Laufzeitfehler auf. Danach ist es nicht mehr möglich weitere Aufnahmen aus dem Bildzwischenspeicher abzurufen, da dieser nicht mehr von Harvesters automatisch gefüllt wird. Harvesters bricht aufgrund einer hierfür gesetzten Timeout-Ausnahme automatisch ab. Dieser Laufzeitfehler wurde untersucht. Es tritt umso schneller auf, je umfangreicher die ausgeführten Prozesse und je aufwendiger die Bildeinstellungen sind. Da das Problem nicht eindeutig reproduzierbar ist und nicht immer nach der exakt gleichen Anzahl an abgerufenen Bildern aus dem Zwischenspeicher auftritt, ist die Problemsuche erschwert. Teilweise tritt es erst nach mehreren hundert Bildaufnahmen ein. Das ist auch der Grund, wieso der Fehler in Tests während der Entwicklung nicht erkannt wurde. Der Laufzeitfehler könnte zum einen an der Bibliothek Harvesters selbst oder zum anderen an der Architektur des Karbon 300 liegen. Für Testzwecke stand kein anderes Gerät mit Linux-Betriebssystem am DCC zur Verfügung. Der Laufzeitfehler konnte nicht mehr im Rahmen der Bachelorarbeit gelöst werden. Da die Bildaufnahme nicht funktioniert, entfällt der Test der anderen Container in diesem Anwendungsfall.

7.2 Kontrolle des Nähprozesses

Die zweite Implementierung findet an der Kontrollstation des Nähprozesses für textile Armbänder am DCC statt (s. Abb. 7.3 oben). Dazu werden die Bänder mit der zu prüfenden Armbandtasche vor der Kamera aufgespannt (s. Abb. 7.3 unten).

Durch das Auslösen per Knopfleiste als ereignisgesteuerter Trigger wird eine Bildaufnahme erzeugt, diese in der Inferenz beurteilt und das Ergebnis zum Aktivieren einer Statuslampe genutzt. Wenn keine Fehler erkannt werden, leuchtet die Lampe grün (s. Abb. 7.3). Andernfalls leuchtet sie rot, damit der Nutzer das Armband aussortiert. Zum Einsatz kommt die In-Sight 2000-230 von Cognex mit integriertem Lichtring und Blitzlichtfunktion. Die verwendeten Skripte und die Hardware für das Auslösen per Knopfleiste und die Ergebnisverwertung per Lampe stammen von ia: industrial analytics. Die Kamera ist über Ethernet an einen ia: factorycube angebunden. Die Stromversorgung der Kamera erfolgt über ein separates Kabel.

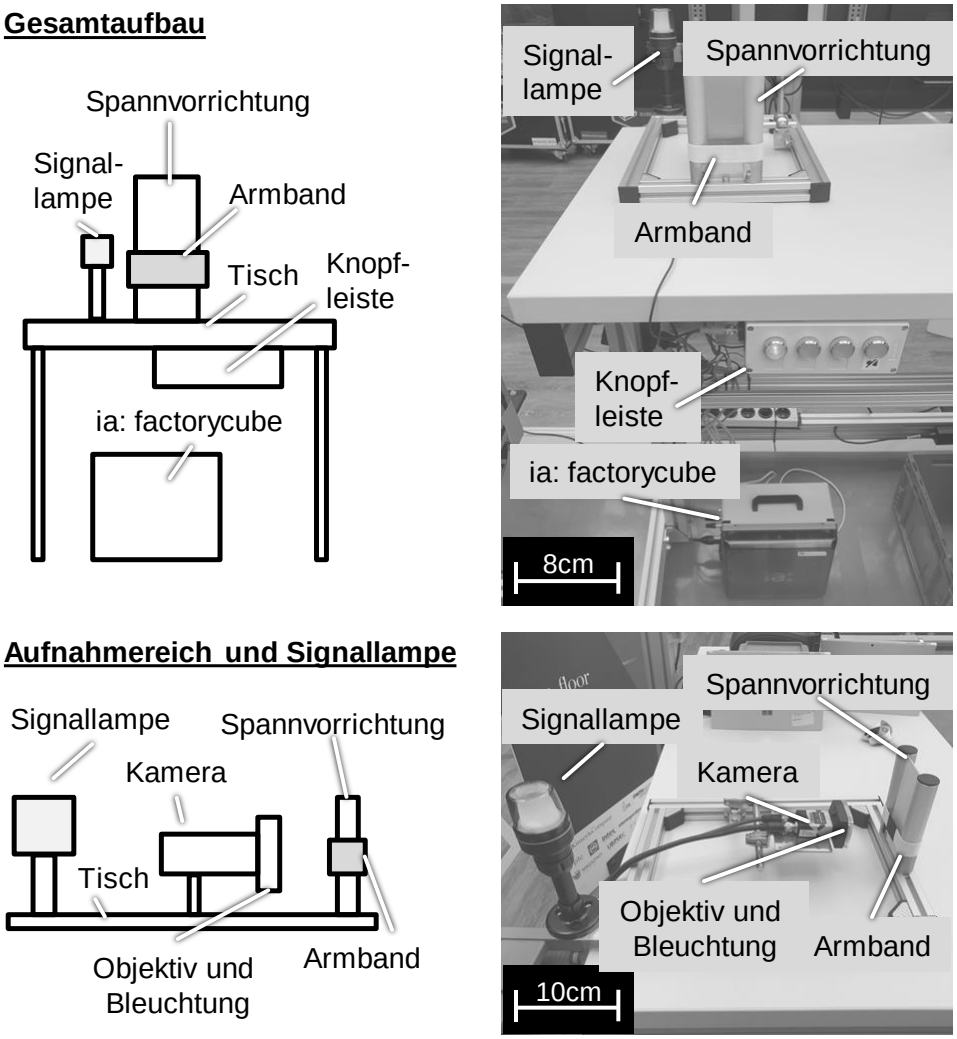


Abb. 7.3: Aufbau des Validierungsfalls Kontrolle des Nähprozesses

An dieser Station kann über die Knopfleiste beliebig zwischen konventionellen Bildverarbeitungsalgorithmen und Methoden des Deep Learnings zur Inspektion des Armbands gewechselt werden. Dies dient vor allem für Demonstrationszwecke

im Rahmen der Workshops im DCC. Die Einstellungen der Kamera und das Training des KNN wurden bereits vorgenommen und stehen für die Arbeit zur Verfügung. Die verwendeten Umgebungsvariablen sind im Anhang 10.13 aufgeführt.

Sofern das KNN verwendet wird, wird in dieser Validierung nur in die Fehlerklassen „qok“ bei keinen Fehlern und „qnok“ bei mangelhaften Produkten unterschieden. Mögliche Produktionsfehler, die beim Vernähen auftreten, sind in Abb. 7.4 dargestellt. Bei den konventionellen Methoden messen die Algorithmen nur die Höhe und Breite der Armbandtasche. Da die Nähte stets eine gewisse Variation aufweisen, ist diese nicht konventionell prüfbar.

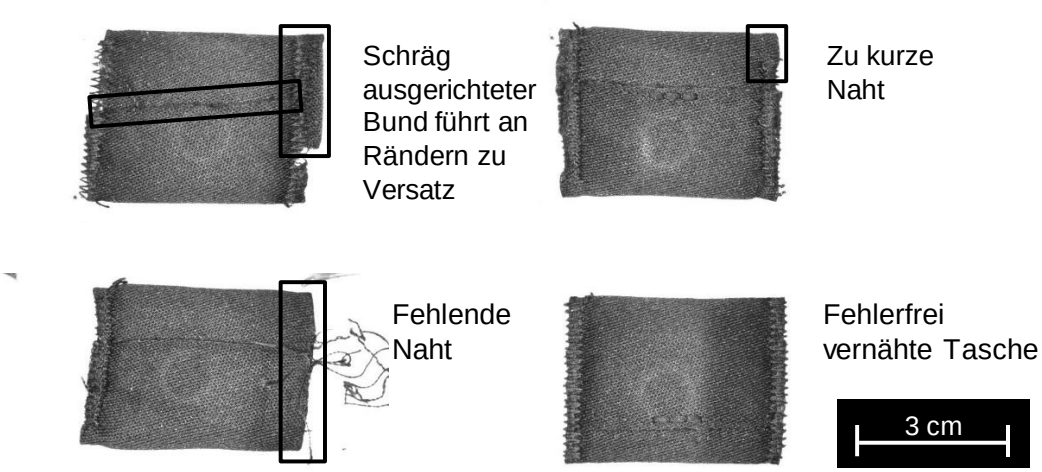


Abb. 7.4: Beispiele für Produktionsfehler an der Armbandtasche

Ergebnis der Validierung

Der Docker-Container für die Bildaufnahme mit der Cognex Kamera funktioniert ohne Probleme. Die Bildspeicherung funktioniert ebenfalls und erfüllt die Anforderungen. Der Inferenz-Client musste allerdings aufgrund einer Architekturinkompatibilität zwischen TensorFlow und dem Karbon 300 des ia: factorycube auf einem separaten Server, der mit dem MQTT-Broker des Karbon 300 verbunden ist, ausgeführt werden. Dieses Problem soll zukünftig durch eine selbst kompilierte Version von TensorFlow umgangen werden und ist bereits in Entwicklung.

7.3 Zusammenfassung der Validierungsergebnisse

In der Validierung sind Laufzeitfehler mit den Modulen Genicam und Harvesters sowie der TensorFlow-Bibliothek aufgetreten. Diese Laufzeitfehler waren während der reinen Entwicklung noch nicht erkennbar und traten erst bei der Implementierung in der Produktionsumgebung mit dem Karbon 300 des ia: factorycubes auf.

Probleme zwischen der Entwicklungs- und Produktionsumgebung sollten durch kontinuierliche Tests vermieden werden. Während den Zwischentests bei dieser Entwicklung ist das Problem mit der GenICam Kamera bis zur Validierung nicht aufgetreten und konnte daher nicht frühzeitig beachtet werden. Die Architekturinkompatibilität zwischen TensorFlow und dem ia: factorycube war bereits vorher bekannt und hat nicht im Aufgabenbereich der Bachelorarbeit gelegen. Eine Lösung dieses Problems steht weiterhin aus.

Der Vergleich der zwei verschiedenen Kameraschnittstellen hat gezeigt, dass die Python-Schnittstelle des sehr offenen GenICam Standards in der Validierung noch nicht stabil funktioniert. Ob dies primär am GenICam Standard an sich oder nur an der Python-Implementierung mit den Modulen Genicam und Harvesters liegt, konnte im Rahmen dieser Bachelorarbeit nicht evaluiert werden. Dagegen wurde mit dem Cognex-System gezeigt, dass es stabil und intuitiv über Telnet und FTP steuer- und nutzbar ist. Auch wenn GenICam ein Standard ist, der von vielen Herstellern genutzt wird, sollten die Ergebnisse der Validierung bei der Auswahl auch in Zukunft berücksichtigt werden, um zwischen Implementierungsaufwand und Nutzen abzuwägen. Die GenICam Schnittstelle ist durch ihre Struktur (s. Kapitel 3.1.2) wesentlich komplexer im Verständnis und der Kommunikation als eine Verbindung über Telnet im ASCII-Format und FTP (s. Kapitel 3.2.5). Dafür werden im GenICam Standard durch die Vielzahl an Features (s. SFNC Kapitel 3.1.2) dem Entwickler mehr Freiheiten in der Konfiguration und Ausführung der Kamera gelassen. Die Anbindung einer Cognex Kamera in eigene Applikationen beschränkt sich dagegen auf die Native Mode Commands [Cog20a], die in ihrem Umfang je nach Kameramodell variieren, aber insgesamt weniger Möglichkeiten bieten als die Features des GenICam Standards. Diese Ergebnisse und Erfahrungen aus der Implementierung der beiden Kameraschnittstellen sind in Tab. 7.1 gegenübergestellt. Als Kriterien werden hierzu die Interoperabilität, die Implementierung und der Funktionsumfang, der mit der Komplexität der Schnittstelle korreliert, verwendet.

Tab. 7.1: Gegenüberstellung der genutzten Kameraschnittstellen auf Basis eigener Erfahrungen aus der Entwicklung und Validierung der Software

	GenICam: Python-Implementierung mit den Modulen Genicam [The20b] und Harvesters [The20a]	**Cognex: Nutzung des Cognex Native Command Protokolls [Cog20a] über Telnet und Datenaustausch über FTP**
Interoperabilität	Herstellerunabhängig, weit verbreitete Schnittstelle für industrielle Kameras [Ste19b; AEJ+18]	Nur Cognex, Telnet und FTP keine dedizierten Standards für Kameras
Implementierung	Keine offizielle Dokumentation für Python, Laufzeitfehler	Ausführliche Dokumentation mit Hinweisen zur Fehlervermeidung
Funktionsumfang (Komplexität)	Meist über 100 Features änderbar und ausführbar, wobei die genaue Anzahl herstellerabhängig ist (Allied Vision Mako G-223C ca. 108 Features)	Beschränkung auf Native Mode Protokoll (33 Basic Native Commands, Extended Native Commands nicht verfügbar in niedriger Preisklasse)

8 Zusammenfassung und Ausblick

Die industrielle Bildverarbeitung (IBV) ermöglicht eine automatisierte zerstörungsfreie und berührungslose Qualitätskontrolle von Bauteilen und Produkten. Mithilfe der erlangten Daten über die Qualität kann entweder automatisch oder durch manuelles Eingreifen eines Operators die Qualität des Produkts beziehungsweise des Prozesses verbessert werden. Bisher mangelte es jedoch an systematischen Prozessbeschreibungen, welche die neuen Deep-Learning-basierten Bildverarbeitungsmethoden in den Gesamtprozess zur IBV integrieren und beschreiben. Zudem existierten neben den kommerziellen Softwareprodukten bisher keine ganzheitlichen Lösungen für die industrielle, Deep-Learning-basierte Bildverarbeitung, die lizenzgebührenfrei verfügbar sind. Eine solche Lösung würde sich insbesondere für das Prototyping und den Einsatz in kleinen und mittelständischen Unternehmen eignen, um die hohen Einführungskosten kommerzieller Lösungen zu meiden.

Das Ziel der Abschlussarbeit war die Entwicklung einer robusten und anwenderfreundlichen Software für ein System zur IBV, das Methoden des Deep Learnings anwendet. Der Nutzer dieser Software wird aufgrund der Plug-and-Play Fähigkeit und der standardisierten Schnittstellen in die Lage versetzt, ein IBV-System schnell und einfach in Betrieb zu nehmen. Die Systemsoftware basiert ausschließlich auf lizenzgebührenfreien Softwareprodukten.

Zur Erarbeitung des Gesamtsystems wurden relevante Standards, Schnittstellen und Softwarelösungen recherchiert und vorgestellt (s. Forschungsfrage 1). Durch die Einteilung des Systems in Teilprozesse wurden funktionale Anforderungen an die Software abgeleitet (s. Forschungsfrage 2) und in der Entwicklung mit den allgemeinen Anforderungen in einer Systemarchitektur umgesetzt (s. Forschungsfrage 3). Die Implementierung und anschließende Validierung erfolgt in der Modellproduktion für textile Armbänder am Digital Capability Center (DCC) Aachen.

Die detaillierte Betrachtung von Hardwarekomponenten des IBV-Systems wie Kamera, Beleuchtung oder Prozessor ist nicht Teil dieser Bachelorarbeit. Kompatible Kameras sind auf die Hersteller Cognex Corporation, Natick, USA, und Allied Vision Technologies GmbH, Stadtroda, mit GigE Vision Standard eingegrenzt. Das Training des Künstlichen Neuronalen Netz (KNN) wurde nur grundlegend anhand beispielhafter Softwareprodukte beschrieben und wird nicht durch die entwickelte Software selbst unterstützt. Der Inferenz-Server mit dem trainierten KNN wurde für die Arbeit bereitgestellt.

Das zentrale Ergebnis der Bachelorarbeit ist eine Gesamtprozessübersicht und eine Microservice-Architektur, mit deren Hilfe sich ein IBV-System nur durch Kon-

figuration der Kamera und Eingabe der Umgebungsvariablen softwareseitig in Betrieb nehmen lässt. Derzeit werden dabei Kameras des GenICam Standards mit GigE Vision Schnittstelle und Cognex Kameras unterstützt. Durch die offene Architektur wird eine Basisplattform für die Entwicklung weiterer Microservices und Folgeprozesse im Kontext der industriellen Bildverarbeitung geschaffen. Die zentrale Anforderung der Plug-and-Play Fähigkeit ist damit erfüllt worden, ebenso wie die Beschränkung auf lizenzgebührenfreie Software. Derzeit werden dabei Kameras des GenICam Standards mit GigE Vision Schnittstelle und Cognex Kameras unterstützt. Die Microservice-Architektur ist mit allen entwickelten Microservices, den verwendeten Schnittstellen und den über MQTT ausgetauschten Nachrichteninhalten in Abb. 8.1 dargestellt.

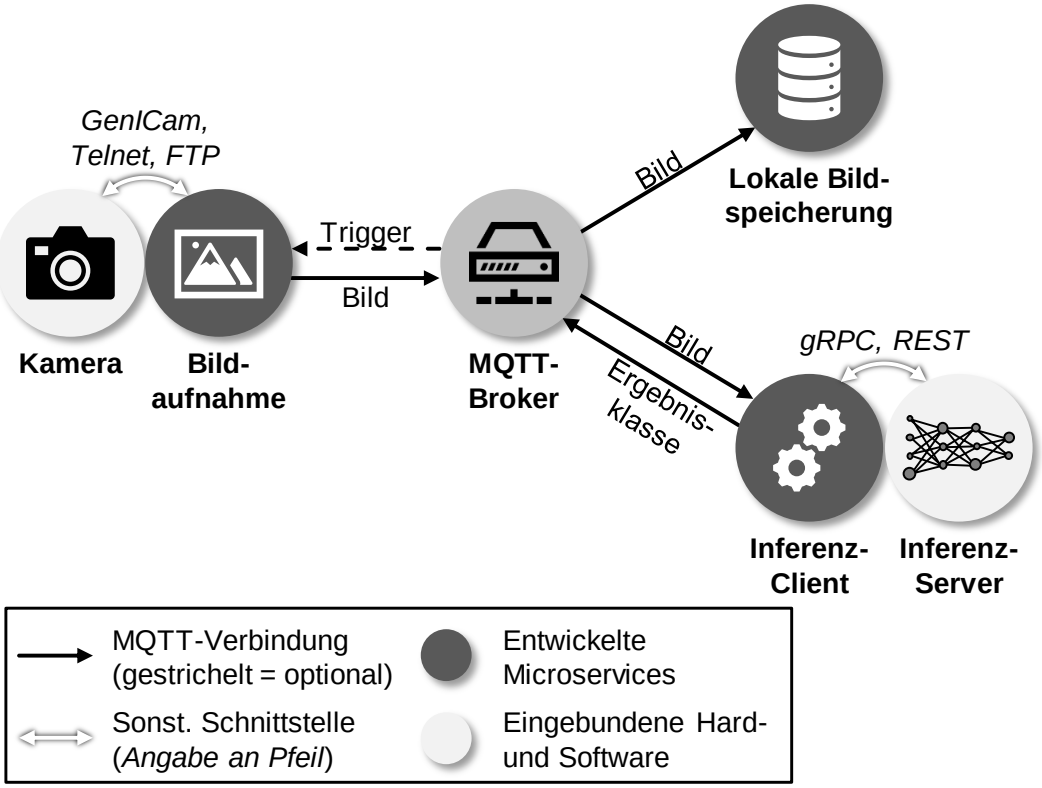


Abb. 8.1: Microservice-Architektur mit allen entwickelten Microservices, verwendeten Schnittstellen und ausgetauschten Nachrichteninhalten

Aufbauend auf den gesammelten Erkenntnissen aus der Betrachtung des Gesamtprozesses und der Entwicklung der Systemsoftware leiten sich abschließend weitere Potentiale für die Forschung und Entwicklung ab. Die nachfolgend genannten Themen sind dabei nach ihrem Potential und damit ihrer Priorität von hoch nach niedrig geordnet:

Stabilisierung der GenICam Schnittstelle

Um die GenICam Schnittstelle industrietauglich zu gestalten, muss der Laufzeitfehler von Harvesters weiter untersucht werden. Alternativ kann auch der Einsatz der Software Common Vision Blox von Stemmer Imaging geprüft werden [Ste19a]. Diese Software steht allerdings nur lizenzkostenfrei zur Verfügung, wenn die Kamera bei Stemmer Imaging, einem großen Händler für Bildverarbeitungshardware, gekauft wird. Zudem bietet Allied Vision seit Mai 2020 mit der Softwareversion 4.0 von Vimba auch einen eigenen Python-Wrapper an, sodass Vimba nun auch eine Python-API besitzt. Jedoch findet damit eine Festlegung auf Kameras des Herstellers Allied Vision für die Zukunft statt.

Automatisierung des Trainingsprozesses

In der momentanen Systemauslegung müssen die Bilder noch separat durch einen menschlichen „Lehrer“ gelabelt werden. Auch dieser Prozess kann durch eine entsprechende Systemauslegung mit Kameras vor und hinter dem menschlichen Prüfprozess automatisiert werden. Ebenfalls kann die Anwendung von Unsupervised Learning geprüft werden. Mit einem automatisierten Training würde auch Continuous Learning ermöglicht, was die Inspektionsqualität schrittweise immer weiter verbessern würde.

Entwicklung von Microservices für Folgeprozesse

Derzeit werden die Inspektionsergebnisse durch die entwickelten Microservices noch nicht weiterverwertet. Das bietet Potential für standardisierte Microservices zur Visualisierung, Datenbankspeicherung oder auch in begrenztem Maße zur Maschinensteuerung. Sofern eine Maschinensteuerung geplant wird, muss das System auf Echtzeitfähigkeit geprüft werden. Eine Umwandlung des Codes in C/C++ oder eine Implementierung von OPC UA können dann notwendig werden.

Objekterkennung und Objekttracking bei mehreren Objekten im Aufnahmebereich

Für Produktionen, bei denen sich mehrere Produkte gleichzeitig in der Bildszene befinden, ist die Implementierung einer Objekterkennung sinnvoll. Damit können die Objekte einzeln auf ihre Qualität geprüft werden.

Insgesamt wurde das grundlegende Ziel der Bachelorarbeit erreicht und die definierten Anforderungen durch die entwickelte Microservice-Architektur erfüllt. Die verbleibenden Herausforderungen wurden dargelegt und bilden die Basis für die Weiterentwicklung des IBV-Systems.

9 Literaturverzeichnis

[AB20] Arsenov, Nestor; Baumgartner, Christian
image-sorter2, Lizenztyp: Apache-2.0 License, 27.03.2020, https://github.com/Nestak2/image-sorter2, Zugriff am 18.06.2020
[Software]

[AEJ+18] AIA, EMVA, JIIA, VDMA und CMVU (Hrsg.):
Guide to Understanding Machine Vision Standards, 2018, https://ibv.vdma.org/documents/256550/15131362/Guide-to-Understanding-MV-Standards_1545405766669.pdf/dfe10d21-c68c-f356-57e3-6d567ab7cb0b, Zugriff am 19.06.2020

[Aia20a] AIA: GigE Vision. True Plug and Play Connectivity, www.visiononline.org/GigEVision, Zugriff am 10.05.2020

[Aia20b] AIA: USB3 Vision. High Bandwidth Yet Simple Connectivity, www.visiononline.org/USB3Vision, Zugriff am 10.05.2020

[Aia20c] AIA: Camera Link. The Only Real-Time Machine Vision Protocol, www.visiononline.org/cameralink, Zugriff am 10.05.2020

[Aia20d] AIA: Camera Link HS. The Machine Vision Protocol Moving Forward, www.visiononline.org/cameralinkHS, Zugriff am 10.05.2020

[Ant19] Antoniou, Manos: Predicting the future popularity of programming languages, https://towardsdatascience.com/predicting-the-future-popularity-of-programming-languages-4f28c80bd36f, Zugriff am 08.06.2020

[Bau17] Baun, C.
Virtualisierung.
In Baun, C.:
Betriebssysteme kompakt. - Berlin: Springer Vieweg, 2017

[BB20] Brüggemann, H.; Bremer, P.
Grundlagen Qualitätsmanagement.
3. Auflage. - Wiesbaden: Springer Vieweg, 2020

[BBB+19] Banks, Andrew; Briggs, Ed; Borgendale, Ken; Gupta, Rahul
MQTT Version 5.0, März 2019, https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html, Zugriff am 11.05.2020

[BPF16] Beyerer, J.; Puente León, F.; Frese, C.
Automatische Sichtprüfung.
2., erweiterte und verbesserte Auflage. - Berlin, Heidelberg: Springer Vieweg, 2016

[BPT15] Belshe, Mike; Peon, Roberto; Thomson, Martin: Hypertext transfer protocol version 2 (http/2), https://www.hjp.at/doc/rfc/rfc7540.html, Zugriff am 17.06.2020

[Cas19] Cass, Stephen: The Top Programming Languages 2019, https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019, Zugriff am 25.06.2020

[Cho20] Chollet, Francois
Keras, Version: 2.4.3, Lizenztyp: MIT License, 25.06.2020, https://pypi.org/project/Keras/, Zugriff am 26.06.2020
[Software]

[CKM18] Chui, Michael; Kamalnath, Vishnu; McCarthy, Brian
An executive's guide to AI
McKinsey & Company, Inc., 2018, https://www.mckinsey.com/~/media/McKinsey/Business%20Functions/McKinsey%20Analytics/Our%20Insights/An%20executives%20guide%20to%20AI/An-executives-guide-to-AI.ashx, Zugriff am 11.04.2020
[Firmenschrift]

[Cog18] Cognex Corporation (Hrsg.):
Deep Learning für die automatische Fertigung
Natick (USA): Cognex Corporation, 2018
[Firmenschrift]

[Cog20a] Cognex Corporation (Hrsg.):
Native Mode Communications, 23.04.2020, https://support.cognex.com/docs/is_590/web/EN/ise/Content/Communications_Reference/NativeModeCommunications.htm?tocpath=Communications%20Reference%7CNative%20Mode%20Communications%7C_____0, Zugriff am 12.06.2020
[Firmenschrift]

[Cog20b] Cognex Corporation (Hrsg.):
In-Sight Explorer, Version: 5.9.2, 15.06.2020, https://support.cognex.com/de-de/downloads/detail/insight/4088/1033, Zugriff am 18.06.2020
[Software]

[Cop16] Copeland, Michael: What's the Difference Between Deep Learning Training and Inference?, https://blogs.nvidia.com/blog/2016/08/22/difference-deep-learning-training-inference-ai/, Zugriff am 09.06.2020

[DIN EN ISO9000] DIN EN ISO 9000:2015
Qualitätsmanagementsysteme – Grundlagen und Begriffe (ISO 9000:2015)

[DIN55350] DIN 55350 Teil 17
Begriffe der Qualitätssicherung und Statistik. Begriffe der Qualitätsprüfungsarten

[Doc20] Docker Inc. (Hrsg.):
What is a Container? A standardized unit of software, 2020, https://www.docker.com/resources/what-container, Zugriff am 05.04.2020
[Firmenschrift]

[Dri17] Drilling, Thomas: Web-Service-APIs Contest - Teil 3. Konzept, Aufbau und Funktionsweise von REST, https://www.dev-insider.de/konzept-aufbau-und-funktionsweise-von-rest-a-603152/, Zugriff am 17.06.2020

[DSS11] Demant, C.; Streicher-Abel, B.; Springhoff, A.
Industrielle Bildverarbeitung.
3. Aufl. - Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg, 2011

[Emv19] EMVA (Hrsg.):
GenICam Standard FeaturesNaming Convention(SFNC), 27.05.2019, https://www.emva.org/wp-content/uploads/GenICam_SFNC_v2_5.pdf, Zugriff am 18.06.2020

[Ert16] Ertel, W.
Grundkurs Künstliche Intelligenz.
4., überarbeitete Auflage. - Wiesbaden: Springer Vieweg, 2016

[Ett19] Ettun, Yochay: How to apply continual learning to your machine learning models. Improve model accuracy and strengthen performance with continual learning, https://towardsdatascience.com/how-to-apply-continual-learning-to-your-machine-learning-models-4754adcd7f7f, Zugriff am 24.06.2020

[Fie00] Fielding, R. T.
Architectural Styles and the Design of Network-based Software Architectures. Irvine, University of California, Irvinve, Dissertation, 2000, https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf, Zugriff am 16.06.2020

[FKE+19] Feurer, M.; Klein, A.; Eggensperger, K.; Springenberg, J. T.; Blum, M.; Hutter, F.
Auto-sklearn: Efficient and Robust Automated Machine Learning.
In Hutter, F.; Kotthoff, L.; Vanschoren, J.:
Automated Machine Learning. - Cham: Springer International Publishing, 2019

[Gan18] Gangele, Jeetendra: How does AutoML works?, https://medium.com/@gangele397/how-does-automl-works-b0f9e45fbb24, Zugriff am 12.04.2020

[GB04] Graves, M.; Batchelor, B. (Hrsg.)
Machine vision for the inspection of natural products.
2nd printing 2004. - London: Springer, 2004

[GBC16] Goodfellow, I.; Bengio, Y.; Courville, A.
Deep learning. - Cambridge, Massachusetts, London, England: MIT Press, 2016

[Gei18] Geißler, Otto: Was ist Plug-and-Play? Schneller, flexibler und bequemer, https://www.datacenter-insider.de/was-ist-plug-and-play-a-688240/, Zugriff am 08.06.2020

[Goo20a] Google Inc. (Hrsg.):
TensorFlow Serving, Version: 2.2.0, Lizenztyp: Apache-2.0 License, 03.06.2020, https://github.com/tensorflow/serving, Zugriff am 24.06.2020
[Software]

[Goo20b] Google Inc. (Hrsg.):
AutoML Vision, 2020, https://cloud.google.com/vision/automl/docs?hl=de, Zugriff am 26.06.2020
[Software]

[Goo20c] Google Inc. (Hrsg.):
tensorflow, Version: 2.2.0, Lizenztyp: Apache-2.0 License, 07.05.2020, https://pypi.org/project/tensorflow/, Zugriff am 26.06.2020
[Software]

[Goo20d] Google Inc. (Hrsg.):
TensorFlow Serving Models. Introduction, 31.03.2020, https://www.tensorflow.org/tfx/guide/serving, Zugriff am 13.06.2020
[Firmenschrift]

[Grp20] gRPC Authors: About gRPC. Who is using it, why they're using it, and which platforms support it, https://grpc.io/about/, Zugriff am 18.05.2020

[Hol17] Holmes, N.
Camera Computer Interfaces.
In Hornberg, A.:
Handbook of machine and computer vision. Second, revised and updated edition. - Weinheim, Germany: Wiley-VCH, 2017

[Hub16] Huber, W.
Industrie 4.0 in der Automobilproduktion. - Wiesbaden: Springer Vieweg, 2016

[Ibm20] IBM Corporation (Hrsg.):
AutoAI, 2020, https://www.ibm.com/de-de/cloud/watson-studio/autoai, Zugriff am 26.06.2020
[Software]

[Int20a] Intel Corporation (Hrsg.):
OpenVINO™ Toolkit Documentation, 2020, https://docs.openvinotoolkit.org/latest/index.html, Zugriff am 13.06.2020
[Firmenschrift]

[Int20b] Intel Corporation (Hrsg.):
OpenVINO™ Toolkit - Deep Learning Deployment Toolkit repository, Version: 2020.3 LTS, Lizenztyp: Apache-2.0 License, 03.06.2020, https://github.com/openvinotoolkit/openvino, Zugriff am 24.06.2020
[Software]

[ISO5807] ISO 5807:1985
Information processing; Documentation symbols and conventions for data, program and system flowcharts, program network charts and system resources charts

[Itz19] Itzkovich, Sefi: Redis, Kafka or RabbitMQ: Which Micro-Services Message Broker To Choose?, https://otonomo.io/blog/redis-kafka-or-rabbitmq-which-microservices-message-broker-to-choose/, Zugriff am 14.06.2020

[Jah17] Jahr, I.
Lighting in Machine Vision.
In Hornberg, A.:
Handbook of machine and computer vision. Second, revised and updated edition. - Weinheim, Germany: Wiley-VCH, 2017

[Jii20] JIIA: What is CoaXPress? - Technical Summary, www.coaxpress.com, Zugriff am 10.05.2020

[Jol18] Joliveau, Matteo: Microservices communications. Why you should switch to message queues., https://dev.to/matteo-joliveau/microservices-communications-why-you-should-switch-to-message-queues--48ia, Zugriff am 14.06.2020

[Kam20] Kamaruzzaman, Md: Top 10 In-Demand programming languages to learn in 2020, https://towardsdatascience.com/top-10-in-demand-programming-languages-to-learn-in-2020-4462eb7d8d3e, Zugriff am 25.06.2020

[Klo19] Klostermann, Aiko: An ML showdown in search of the best tool, https://www.thoughtworks.com/insights/blog/machine-learning-showdown-best-tool, Zugriff am 19.05.2020

[Kof18] Kofler, T.
Das digitale Unternehmen. - Berlin: Springer Vieweg, 2018

[KS19] Kreutzer, R. T.; Sirrenberg, M.
Künstliche Intelligenz verstehen. - Wiesbaden: Springer Gabler, 2019

[KTH+17] Kotthoff, Lars; Thornton, Chris; Hutter, Frank; Hoos, Holger; Leyton-Brown, Kevin
Auto-WEKA, Version: 2.6, Lizenztyp: GNU General Public License version 3, Juli 2017, http://www.cs.ubc.ca/labs/beta/Projects/autoweka/, Zugriff am 26.06.2020
[Software]

[Lig20] Light, Roger
Eclipse Mosquitto™ An open source MQTT broker, Lizenztyp: Dual licensed under the Eclipse Public License 1.0 and the Eclipse Distribution License 1.0, 2020, https://mosquitto.org, Zugriff am 20.06.2020
[Software]

[Lit18] Litzenberger, Gudrun: Machine Vision Germany - from record to record since 2010!, https://rua.vdma.org/documents/256550/24333082/Chart_MV_Germany_development.jpg/a3dd77ed-df83-4ca6-b58e-9f0783affaed?t=1517839154747, Zugriff am 10.04.2020

[Liu17] Liu, B.
Lifelong machine learning: a paradigm for continuous learning
Frontiers of Computer Science. Bd. 11 (2017) 3, S. 359–361

[LS18] Luoto, A.; Systä, K.
IoT Application Deployment Using Request-Response Pattern with MQTT.
In Garrigós, I.; Wimmer, M.:
Current trends in web engineering. 10544. - Cham: Springer, 2018

[Lub16] Luber, Stefan: Was ist Machine Learning?, https://www.bigdata-insider.de/was-ist-machine-learning-a-592092/, Zugriff am 11.04.2020

[Lub20] Luber, Stefan: Was ist automatisiertes Machine Learning (AutoML)?, https://www.bigdata-insider.de/was-ist-automatisiertes-machine-learning-automl-a-896975/, Zugriff am 12.04.2020

[Lüb20] Lübkemeier, Thomas: GenICam, https://www.emva.org/standards-technology/genicam/introduction-new/, Zugriff am 10.05.2020

[LWR20] La Torre, Cesar de; Wagner, Bill; Rousos, Mike
.NET Microservices: Architecture for Containerized .NET

Applications
Redmond (USA), 2020, https://docs.microsoft.com/de-de/dotnet/architecture/microservices/, Zugriff am 14.06.2020
[Firmenschrift]

[LZ17] Le, Quoc; Zoph, Barret: Using Machine Learning to Explore Neural Network Architecture, https://ai.google-blog.com/2017/05/using-machine-learning-to-explore.html, Zugriff am 12.04.2020

[Mac19] Machine Learning Professorship Freiburg (Hrsg.):
auto-sklearn, Version: 0.7.0, Lizenztyp: BSD-3-Clause License, 07.05.2019, https://automl.github.io/auto-sklearn/master/, Zugriff am 26.06.2020
[Software]

[Mat16] Matzer, Michael: Big Data und Deep Learning. So spürt Deep Learning Datenmuster auf, https://www.elektronikpraxis.vogel.de/so-spuert-deep-learning-datenmuster-auf-a-569865/, Zugriff am 12.04.2020

[Mat17] Mattfeldt, H.
Camera Systems in Machine Vision.
In Hornberg, A.:
Handbook of machine and computer vision. Second, revised and updated edition. - Weinheim, Germany: Wiley-VCH, 2017

[Mat20] MATRIX VISION GmbH (Hrsg.):
mvIMPACT Acquire SDK, Version: 2.37.1, 2020, http://static.matrix-vision.com/mvIMPACT_Acquire/2.37.1/, Zugriff am 05.06.2020
[Software]

[Mic20] Microsoft Corporation (Hrsg.):
Custom Vision, 2020, https://azure.microsoft.com/de-de/services/cognitive-services/custom-vision-service/, Zugriff am 26.06.2020
[Software]

[Mor20] morefigs (Hrsg.):
pymba, Version: 0.3.7, Lizenztyp: MIT License, 16.06.2020, https://pypi.org/project/pymba/, Zugriff am 18.06.2020
[Software]

[MS13] Meinel, C.; Sack, H.
Internetworking. - Berlin, Heidelberg, s.l.: Springer Berlin Heidelberg, 2013

[Myl98] Myler, H. R.
Fundamentals of engineering programming with C and Fortran. - Cambridge: Cambridge University Press, 1998

[NB10] Nufer, D.; Braunsteiner, A.
Gute Qualitätssicherung oder niedrige Herstellkosten?
Laser Technik Journal. Bd. 7 (2010) 6, S. 18–21

[New15] Newman, S.
Building microservices.
First edition. - Sebastopol, CA: O'Reilly Media, 2015

[New19] Newton-King, James
Compare gRPC services with HTTP APIs | Microsoft Docs, 05.12.2019, https://docs.microsoft.com/en-us/aspnet/core/grpc/comparison?view=aspnetcore-3.1, Zugriff am 18.05.2020
[Firmenschrift]

[Nvi20a] NVIDIA Corporation (Hrsg.):
NVIDIA Triton Inference Server, Version: 1.13.0, Lizenztyp: BSD-3-Clause License, 01.06.2020, https://github.com/NVIDIA/triton-inference-server, Zugriff am 24.06.2020
[Software]

[Nvi20b] NVIDIA Corporation (Hrsg.):
NVIDIA Triton Inference Server, 2020, https://developer.nvidia.com/nvidia-triton-inference-server, Zugriff am 13.06.2020
[Firmenschrift]

[OCC+20] O'Mahony, N.; Campbell, S.; Carvalho, A.; Harapanahalli, S.; Hernandez, G. V.; Krpalkova, L.; Riordan, D.; Walsh, J.
Deep Learning vs. Traditional Computer Vision.
In Arai, K.; Kapoor, S.:
Advances in computer vision. 943. - Cham: Springer, 2020

[PG74] Popek, G. J.; Goldberg, R. P.
Formal requirements for virtualizable third generation architectures
Communications of the ACM. Bd. 17 (1974) 7, S. 412–421

[Pre00] Prechelt, L.
An empirical comparison of seven programming languages
Computer. Bd. 33 (2000) 10, S. 23–29

[Pyt20] PyTorch Team (Hrsg.):
torch (PyTorch), Version: 1.5.1, Lizenztyp: BSD-3-Clause License, 18.06.2020, https://pypi.org/project/torch/, Zugriff am 26.06.2020
[Software]

[Ras17] Raschbichler, Florian: MQTT - Leitfaden zum Protokoll für das Internet der Dinge, https://www.informatik-aktuell.de/betrieb/netzwerke/mqtt-leitfaden-zum-protokoll-fuer-das-internet-der-dinge.html, Zugriff am 16.06.20

[Ric15] Richardson, Chris: Microservices: From Design to Deployment. Introduction to Microservices, https://www.nginx.com/blog/introduction-to-microservices/, Zugriff am 14.06.2020

[Rog02] Rogers, H.
Theory of recursive functions and effective computability. 5. print. - Cambridge, Mass.: MIT Press, 2002

[Roj93] Rojas, R.
Theorie der neuronalen Netze. - Berlin, Heidelberg: Springer, 1993

[RSD+19] Rocha, M. S.; Sestito, G. S.; Dias, A. L.; Turcato, A. C.; Brandão, D.; Ferrari, P.
On the performance of OPC UA and MQTT for data exchange between industrial plants and cloud servers
ACTA IMEKO. Bd. 8 (2019) 2, S. 80

[Sch03] Schnabel, P.
Computertechnik-Fibel. - Ludwigsburg, Im Hafer 6: P. Schnabel, 2003

[Sid19] Sidana, Utkarsh: Python vs C: Know what are the differences, https://www.edureka.co/blog/python-vs-c/, Zugriff am 08.06.2020

[SM19] Shekhovtsov, Serhiy; Mones, Ryan
tkteach, Lizenztyp: Apache-2.0 License, 11.12.2019, https://github.com/Serhiy-Shekhovtsov/tkteach, Zugriff am 18.06.2020
[Software]

[Spe18] Spectra GmbH Co. KG (Hrsg.):
INDUSTRY 4.0 & IIOT. Smart machine data make the difference, 2018, https://cms.spectra.de/fileadmin/user_upload/Downloads_PDFs/Flyer-IIoT_and_Industry4.0.pdf, Zugriff am 21.06.2020
[Firmenschrift]

[Spi12] Spinnler, K.
Leitfaden zur industriellen Bildverarbeitung. Bd. 13. - Stuttgart: Fraunhofer Verlag, 2012

[Sro17] Srocke, Dirk: Definition: Representational State Transfer (REST) Application Programming Interface (API). Was ist eine REST API?, https://www.cloudcomputing-insider.de/was-ist-eine-rest-api-a-611116/, Zugriff am 17.06.2020

[Ste19a] Stemmer Imaging AG (Hrsg.):
Common Vision Blox (CVB) 2019, Version: 13.02.002,

2019, https://www.commonvisionblox.com/en/cvb-download/, Zugriff am 22.06.2020
[Software]

[Ste19b] Stemmer Imaging AG (Hrsg.):
Das Handbuch der Bildverarbeitung
Puchheim, 2019
[Firmenschrift]

[SUW18] Steger, C.; Ulrich, M.; Wiedemann, C. (Hrsg.)
Machine vision algorithms and applications.
2nd, completely revised and enlarged edition. - Weinheim: Wiley-VCH, 2018

[Tel17] Telljohann, A.
Introduction to Building a Machine Vision Inspection.
In Hornberg, A.:
Handbook of machine and computer vision. Second, revised and updated edition. - Weinheim, Germany: Wiley-VCH, 2017

[The20a] The GenICam Committee (Hrsg.):
harvesters, Version: 1.2.6, Lizenztyp: Apache-2.0 License, 11.06.2020, https://pypi.org/project/harvesters/, Zugriff am 17.06.2020
[Software]

[The20b] The GenICam Committee (Hrsg.):
genicam, Version: 1.0.1, Lizenztyp: Other/Proprietary License (The GenICam License Version 1.6), 10.06.2020, https://pypi.org/project/genicam/, Zugriff am 17.06.2020
[Software]

[The20c] The GenICam Committee (Hrsg.):
harvesters-gui, Version: 1.0.1, Lizenztyp: Apache-2.0 License, 08.03.2020, https://pypi.org/project/harvesters-gui/, Zugriff am 18.06.2020
[Software]

[Ung20] Unger, Patrick: AutoML. So automatisiert KI die Entwicklung von KI-Modellen, https://www.bigdata-insider.de/so-automatisiert-ki-die-entwicklung-von-ki-modellen-a-897404/, Zugriff am 12.04.2020

[UWJ18] Uffelmann, J. R.; Wienzek, P.; Jahn, M.
IO-Link.
2. Auflage 2018. - Essen: Deutscher Industrieverlag, 2018

[VDD+18] Voulodimos, A.; Doulamis, N.; Doulamis, A.; Protopapadakis, E.
Deep Learning for Computer Vision: A Brief Review
Computational intelligence and neuroscience. Bd. 2018 (2018) Special Issue, S. 9–21

[VDI/VDE2628] VDI/VDE 2628 Blatt 1
Automatisierte Sichtprüfung. Beschreibung der Prüfaufgabe

[Vdm16] VDMA Machine Vision Group within the Robotics + Automation Association (Hrsg.):
Machine Vision 2017/2018. Key Technology for Automation Solutions
Frankfurt, 2016

[Vdm18] VDMA Fachabteilung Industruelle Bildverarbeitung im Fachverband Robotik + Automation (Hrsg.):
Industrielle Bildverarbeitung 2018/2019. Schlüsseltechnologie für die Automatisierung
Frankfurt, 2018

[Ven17] Venugopal, M.V.L.N.
Containerized Microservices architecture
International Journal of Engineering and Computer Science. Bd. 6 (2017) 11, S. 23199–23208

[Voi18] Voigt, Kai-Ingo: Qualitätssicherung, https://wirtschaftslexikon.gabler.de/definition/qualitaetssicherung-44396/version-267707, Zugriff am 05.04.2020

[War18] Wartala, R.
Praxiseinstieg Deep Learning.
1. Auflage. - Heidelberg: O'Reilly, 2018

[WDK+18] Wollschlaeger, Martin; Debes, Thomas; Kalhoff, Johannes; Wickinger, Jens; Dietz, Holger; Feldmeier, Günter; Michels, Jan; Scholing, Heinz; Billmann, Meik
Kommunikation im Industrie-4.0-Umfeld
Frankfurt, April 2018, https://www.zvei.org/fileadmin/user_upload/Presse_und_Medien/Publikationen/2018/April/Kommunikation_im_Industrie-4.0-Umfeld/Kommunikation_im_Industrie-4.0-Umfeld_Download-Neu.pdf

[Wen18] Wendel, Anne: VDMA: Industrielle Bildverarbeitung auf Rekordniveau. - Stuttgart, https://ibv.vdma.org/viewer/-/v2article/render/27036181, Zugriff am 17.06.2020

[Whe11] Wheen, A.
Dot-Dash to Dot.Com. - New York, NY: Springer Science+Business Media LLC, 2011

[WMZ+18] Wang, J.; Ma, Y.; Zhang, L.; Gao, R. X.; Wu, D.
Deep learning for smart manufacturing: Methods and applications
Journal of Manufacturing Systems. Bd. 48 (2018) 46, S. 144–156

[WON+20] Weixuan Fu (Hrsg.):; Olson, Randy; Nathan (Hrsg.):; Grishma Jena (Hrsg.):; PGijsbers (Hrsg.):; Augspurger,

Tom; Romano, Joe; PRONOjit Saha (Hrsg.):; Sahil Shah (Hrsg.):; Raschka, Sebastian; Sohnam (Hrsg.):; DanKoretsky (Hrsg.):; Kadarakos (Hrsg.):; Jaimecclin (Hrsg.):; Bartdp1 (Hrsg.):; Bradway, Geoffrey; Ortiz, Jose; Jorijn Jacko Smit (Hrsg.):; Jan-Hendrik Menke (Hrsg.):; Ficek, Michal; Akshay Varik (Hrsg.):; Chaves, Anderson; Myatt, James; Ted (Hrsg.):; Badaracco, Adrian Garcia; Kastner, Christian; Crypto Jerônimo (Hrsg.):; Hristo (Hrsg.):; Rocklin, Matthew; Carnevale, Randy
EpistasisLab/tpot, Version: 0.11.5, Lizenztyp: LGPL-3.0 License, 02.06.2020, https://github.com/EpistasisLab/tpot, Zugriff am 26.06.2020
[Software]

[WRA96] Watters, A.; van Rossum, G.; Ahlstrom, J. C.
Internet programming with Python. - New York, NY: M&T Books, 1996

[WSS16] Weimer, D.; Scholz-Reiter, B.; Shpitalni, M.
Design of deep convolutional neural network architectures for automated feature extraction in industrial inspection
CIRP Annals. Bd. 65 (2016) 1, S. 417–420

[Xie18] Xie, Yuan: Computer Architecture Today. A Brief Guide of xPU for AI Accelerators, https://www.sigarch.org/a-brief-guide-of-xpu-for-ai-accelerators/, Zugriff am 13.06.2020

[Yua17] Yuan, Michael: Getting to know MQTT. Why MQTT is one of the best network protocols for the Internet of Things, May 2017, https://developer.ibm.com/articles/iot-mqtt-why-good-for-iot/, Zugriff am 11.05.2020

[ZZL+19] Zhang, S.; Zhao, H.; Lu, X.; Qu, J.
Design of IoT Platform Based on MQTT Protocol.
In Krömer, P.; Zhang, H.; Liang, Y.; Pan, J.-S.:
Proceedings of the Fifth Euro-China Conference on Intelligent Data Analysis and Applications. 891. - Cham: Springer, 2019

10 Anhang

10.1 Beschreibung der Hardwareschnittstellen

Dieser Anhang ist eine Ergänzung zu Kapitel 3.1.1.

Einer der ersten Standards war FireWire, auch bekannt als IEEE 1394. Dieser Standard hat jedoch im Laufe der Zeit deutlich an Bedeutung verloren und wurde mehr und mehr durch neue abgelöst. Diese bieten eine deutlich höhere Performance. So ist die Übertragungsrate bei FireWire geringer als 100 MB/s (Megabyte pro Sekunde). Heutige Übertragungsstandards leisten über 5000 MB/s. Daher wird dieser Standard auch im Folgenden nicht mehr berücksichtigt. Auch die G3-Initiative führt diesen Standard lediglich zu Vergleichszwecken weiter mit. Eine Verwendung wird jedoch nicht empfohlen und durch am Markt verfügbare Produkte auch kaum mehr unterstützt. [AEJ+18] Nachfolgend werden die Eigenschaften der fünf derzeit anerkannten und geförderten Hardwarestandards kurz beschrieben.

GigE Vision

GigE Vision ist ein globaler Kameraschnittstellen-Standard, der mit dem weit verbreiteten Gigabit-Ethernet-Kommunikationsprotokoll entwickelt wurde. GigE Vision bietet eine schnelle Bildübertragung über sehr große Distanzen und Multi-Kamera-Funktionalitäten, um gleichzeitig mehrere Kameras anzuschließen. Die Verbindung erfolgt über kostengünstige Standardkabel (Cat 5e/6a/7-Netzwerkkabel). Mit GigE Vision können Hard- und Software von verschiedenen Herstellern nahtlos über Ethernet-Verbindungen zusammenarbeiten und über die Netzwerkkarte an Standard-PCs direkt angeschlossen werden. Durch PoE (Power over Ethernet) können Kameras dieses Standards mit nur einem Kabel installiert werden. [Aia20a; Vdm18] GigE Vision ist die meistverbreitete Schnittstelle in der IBV [Ste19b].

USB3 Vision

USB3 Vision basiert auf der USB 3.x-Schnittstelle, die an allen gängigen PCs und vielen eingebetteten Systemen vorhanden ist. Dadurch sind die Produkte, die diesen Standard erfüllen, mit vielen Produkten anderer Hersteller kompatibel. Der Standard bietet ein einfach zu verwendendes Protokoll, Stromübertragung über das Datenkabel sowie eine einfache Plug-and-Play Installation. [Aia20b; Vdm18]

Camera Link

Camera Link ist eine robuste Kommunikationsschnittstelle, welche die Verbindung zwischen Kameras und Framegrabbern vereinfacht. Dazu wird eine Camera Link spezifische Kabelverbindung und ein einfaches standardisiertes Kommunikations-

protokoll verwendet. Der Standard definiert eine vollständige Hardware-Spezifikation, in der Regelungen für den Datentransfer, die Zeitsteuerung der Kamera, die serielle Kommunikation und die Echtzeit-Signalübertragung enthalten sind. Camera Link wurde für die parallele Kommunikation in Echtzeit und mit hoher Bandbreite (maximal 850 MB/s) entwickelt. Auch hier ermöglicht PoCL (Power over Camera Link) die Übertragung von Strom über das Datenkabel. [Aia20c; Vdm18]

Camera Link HS

Camera Link HS ist speziell für Bildverarbeitungsanwendungen entwickelt worden. Es ermöglicht eine sehr schnelle und zuverlässige Echtzeit-Datenübertragung mit geringer Latenz. Es ist eine Weiterentwicklung des Camera Link Standards und ergänzt diesen um weitere standardisierte Funktionen. [Aia20d; Vdm18]

CoaXPress

CoaXPress verwendet Koaxialkabel für eine sehr schnelle Übertragung von seriellen Daten. Es ist möglich, Geschwindigkeiten von bis zu 12,5 Gb/s pro Kabel zu erreichen sowie die Kamerasteuerung und Stromversorgung („Power-over-Coax“) über ein Kabel umzusetzen. [Jii20; Vdm18]

10.2 Umgebungsvariablen

Nachfolgend sind alle Umgebungsvariablen alphabetisch aufgelistet. Sie sind nach den Verwendungszwecken gegliedert (s. Anhang 10.2.2 für Bildaufnahme Cognex, 10.2.3 für Bildaufnahme GenICam, 10.2.4 für Bildspeicherung und 10.2.5 für Inferenz-Client). Variablen, die Microservice übergreifend genutzt werden, befinden sich im Anhang 10.2.1. Für die Nutzung des Codes kann auf die env-Dateien aus der Implementierung zurückgegriffen werden. Für Anwendungsfälle mit Cognex siehe Anhang 10.13. Für Anwendungsfälle mit GenICam Kamera siehe Anhang 10.12. In diesen Dateien können die Parameter passend zum neuen Anwendungsfall gesetzt werden. Die bereits genutzten Werte aus der Implementierung in dieser Arbeit dienen dabei als Orientierung. Für jede Umgebungsvariable folgt zunächst eine Beschreibung, anschließend die möglichen Eingabewerte und zuletzt der Standardwert, der im Programm hinterlegt ist, falls diese Umgebungsvariable nicht gesetzt wird.

10.2.1 Allgemein

ACQUISITION_DELAY

- Wenn als TRIGGER MQTT verwendet wird, kann mit dieser Variablen, die Zeit zwischen dem Triggersignal und der tatsächlichen Bildaufnahme gesteuert werden. Sofern ein Zeitstempel in der MQTT-Nachricht in Millisekunden enthalten ist, wird dieser für die Berechnung genutzt. Sofern keiner enthalten ist, wird einer generiert, sobald die Nachricht beim Subscriber empfangen wurde. Der Zeitstempel gilt dabei stets seit 01.01.1970 00:00:00 (UTC), also dem Linux Weltalter. Diese Bildaufnahmeverzögerung kann genutzt werden, um schwankende Übertragungslatenzen zu vermeiden und Prozesseinstellungen zu ermöglichen. Sofern es sich um einen stationären Prozess handelt, bei dem die Bildaufnahme so schnell wie möglich nach Triggersignaleingang ausgelöst werden soll, muss 0.0 eingegeben werden. Dann wird das Programm zur Bildaufnahme so schnell wie möglich ohne Verzögerungen ausgeführt.
- *Mögliche Werte*: Gleitkommazahl in Sekunden
- *Standardwert*: 2.0

CAMERA_INTERFACE

- Zur Wahl steht eine Kameraschnittstelle für Cognex Kameras der In-Sight Serie und alle GigE Vision Kameras mit GenICam.
- *Mögliche Werte*: GenICam, Cognex
- *Kein Standardwert*

CYCLE_TIME

- Wenn ein kontinuierlicher Trigger verwendet wird, wird mit der Taktzeit der Abstand zwischen jeder Bildaufnahme eingestellt. Wenn die Taktzeit kürzer als die einmalige Ausführung des Programms ist, wird eine Fehlermeldung an den Nutzer ausgegeben.
- *Mögliche Werte*: Gleitkommazahl in Sekunden
- *Standardwert*: 10.0

MQTT_HOST

- Hostnamen oder IP-Adresse des MQTT-Brokers.
- *Mögliche Werte*: Zeichenfolge ohne Anführungszeichen
- *Kein Standardwert*

MQTT_PORT

- Port des MQTT-Brokers, reservierte Ports für MQTT sind 1883 und 8883 für SSL.
- *Mögliche Werte*: Vierstellige Ganzzahl

- *Standardwert*: 1883

MQTT_TOPIC_IMAGE

- MQTT-Topic zu der Bildaufnahmen gesendet werden.
- *Mögliche Werte*: Beliebiger Pfad mit Slash zur Trennung der Ordnerebenen
- *Standardwert*: qualityInspection/image

MQTT_TOPIC_RESULTS

- MQTT-Topic zu der die Ergebnisse der Inferenz gesendet werden.
- *Mögliche Werte*: Beliebiger Pfad mit Slash zur Trennung der Ordnerebenen
- *Standardwert*: qualityInspection/results

MQTT_TOPIC_TRIGGER

- MQTT-Topic zu der Triggersignale gesendet werden.
- *Mögliche Werte*: Beliebiger Pfad mit Slash zur Trennung der Ordnerebenen
- *Standardwert*: qualityInspection/trigger

SERIAL_NUMBER

- Sofern verfügbar kann hiermit die Seriennummer der Kamera der MQTT Nachricht beigefügt werden. Sofern kein Wert eingegeben wird, wird ein leerer String als Seriennummer verwendet.
- *Mögliche Werte*: Zeichenfolge ohne Anführungszeichen
- *Standardwert*: (leer)

TRIGGER

- Zur Wahl steht ein kontinuierlicher Trigger mit fester Taktzeit oder ein ereignisgesteuerter Trigger über MQTT (MQTT_TOPIC_TRIGGER)
- *Mögliche Werte*: Continuous, MQTT
- *Kein Standardwert*

10.2.2 Bildaufnahme Cognex

CONVENTIONAL_IMAGE_PROCESSING

- Sofern der Job im Cognex In-Sight Explorer hierfür vorbereitet wurde, kann hiermit das Ergebnis der konventionellen Bildverarbeitung abgerufen werden. Nähere Informationen zum Einrichten der Bildverarbeitungsmethoden und zur Formatzeichenkette (FormatString) zur Übergabe der Ergebnisse sind der Cognex Dokumentation online zu entnehmen.
- *Mögliche Werte*: True or False
- *Standardwert*: False

IP_ADDRESS_COGNEX

- IP-Adresse des In-Sight-Sensors. Diese kann im Cognex In-Sight Explorer unter Connection nach dem Verbinden ermittelt werden.
- *Mögliche Werte*: Zeichenfolge ohne Anführungszeichen
- *Kein Standardwert*

JOB_ID

- Diese Variable ermöglicht das Auswählen unterschiedlicher Jobs auf der Cognex Kamera. Ein Job beinhaltet dabei die Kameraeinstellungen und optional auch Bildverarbeitungsmethoden. Um die Funktion Job-ID-Nummer zu verwenden, muss der zu ladende Job mit einem numerischen Präfix von 0 bis 999 auf der Kamera gespeichert werden. Die Job-ID entspricht diesem Präfix. Wird keine Job_ID hinterlegt, wird der Job verwendet, der im In-Sight Explorer als Standardjob eingestellt ist.
- *Mögliche Werte*: Ganzzahl von 0 bis 999 oder default
- *Standardwert*: default

10.2.3 Bildaufnahme GenICam

BALANCE_WHITE_AUTO (nur wenn kein User Set verwendet wird)

- Diese Variable kann genutzt werden, damit die Kamera den Weißabgleich automatisch einstellt. Diese Einstellungen werden nur ausgeführt, wenn die Kamera dies unterstützt. Der Nutzer muss nicht selbst überprüfen, ob die Kamera dies unterstützt.
- *Mögliche Werte*:
    - Off: Keine automatische Anpassung
    - Once: Einmalige automatische Anpassung (Achtung: Dies führt zu einer Änderung der Bildaufnahmeeinstellungen und somit zu Variabilität, die im Künstlichen Neuronalen Netz berücksichtigt werden muss.)
    - Continuous: Kontinuierliche automatische Anpassung (Achtung: Das kann die Bildaufnahmerate durch die häufigen automatischen Nachregelungen deutlich verringern und zu einer hohen Prozessorauslastung führen. Zudem besteht die Gefahr der Variabilität wie bei Once.)
- *Standardwert*: Off

DEFAULT_GENTL_PRODUCER_PATH

- Der Pfad muss nur geändert werden, sofern der GenTL Producer in der Docker-Compose Datei geändert wird. Sofern an diesem nichts geändert wird, kann stets der Standardwert verwendet werden.

- *Mögliche Werte*: Beliebiger Pfad mit Slash zur Trennung der Ordnerebenen
- *Standardwert*: /opt/mvIMPACT_Acquire/lib/x86_64/mvGenTLProducer.cti

EXPOSURE_AUTO (nur wenn kein User Set verwendet wird)

- Diese Variable kann genutzt werden, damit die Kamera die Belichtungszeit automatisch einstellt. Diese Einstellungen werden nur ausgeführt, wenn die Kamera dies unterstützt. Der Nutzer muss nicht selbst überprüfen, ob die Kamera dies unterstützt.
- *Mögliche Werte*:
    - Off: Keine automatische Anpassung
    - Once: Einmalige automatische Anpassung (Achtung: Dies führt zu einer Änderung der Bildaufnahmeeinstellungen und somit zu Variabilität, die im Künstlichen Neuronalen Netz berücksichtigt werden muss.)
    - Continuous: Kontinuierliche automatische Anpassung (Achtung: Das kann die Bildaufnahmerate durch die häufigen automatischen Nachregelungen deutlich verringern und zu einer hohen Prozessorauslastung führen. Zudem besteht die Gefahr der Variabilität wie bei Once.)
- *Standardwert*: Off

EXPOSURE_TIME (nur wenn kein User Set verwendet wird)

- Zur manuellen Eingabe der Belichtungszeit.
- *Mögliche Werte*: Gleitkommazahl oder None
- *Standardwert*: None

GAIN_AUTO (nur wenn kein User Set verwendet wird)

- Diese Variable kann genutzt werden, damit die Kamera die Verstärkung automatisch einstellt. Diese Einstellungen werden nur ausgeführt, wenn die Kamera dies unterstützt. Der Nutzer muss nicht selbst überprüfen, ob die Kamera dies unterstützt.
- *Mögliche Werte*:
    - Off: Keine automatische Anpassung
    - Once: Einmalige automatische Anpassung (Achtung: Dies führt zu einer Änderung der Bildaufnahmeeinstellungen und somit zu Variabilität, die im Künstlichen Neuronalen Netz berücksichtigt werden muss.)
    - Continuous: Kontinuierliche automatische Anpassung (Achtung: Das kann die Bildaufnahmerate durch die häufigen auto-

matischen Nachregelungen deutlich verringern und zu einer hohen Prozessorauslastung führen. Zudem besteht die Gefahr der Variabilität wie bei Once.)

- *Standardwert*: Off

IMAGE_CHANNELS (nur wenn kein User Set verwendet wird)

- Anzahl der Bildkanäle (Bytes pro Pixel), die im Array (dritte Dimension des Bilddatenarrays) verwendet werden. Dieser Wert muss nicht eingestellt werden. Wenn None, wird die beste Anzahl von Kanälen für die Variable PIXEL_FORMAT verwendet.
- *Mögliche Werte*: 1, 3 or None
- *Standardwert*: None

IMAGE_HEIGHT (nur wenn kein User Set verwendet wird)

- Mit der Bildhöhe in Pixeln wird die Region of Interest (ROI) festgelegt. Die ROI wird immer im Kamerasensor zentriert sein. Ein Wert, der höher als die maximale Auflösung der Kamera ist, setzt Maximalwerte anstelle der hier eingegebenen Werte. Um den höchsten Wert herauszufinden, kann nach der Auflösung in den Spezifikationen der Kamera gesucht werden. Die Spezifikationen sind im Handbuch oder auf der Website abrufbar, auf der die Kamera gekauft wurde.
- *Mögliche Werte*: Ganzzahl
- *Standardwert*: 800

IMAGE_WIDTH (nur wenn kein User Set verwendet wird)

- Mit der Bildbreite in Pixeln wird die Region of Interest (ROI) festgelegt. Die ROI wird immer im Kamerasensor zentriert sein. Ein Wert, der höher als die maximale Auflösung der Kamera ist, setzt Maximalwerte anstelle der hier eingegebenen Werte. Um den höchsten Wert herauszufinden, kann nach der Auflösung in den Spezifikationen der Kamera gesucht werden. Die Spezifikationen sind im Handbuch oder auf der Website abrufbar, auf der die Kamera gekauft wurde.
- *Mögliche Werte*: Ganzzahl
- *Standardwert*: 800

PIXEL_FORMAT (nur wenn kein User Set verwendet wird)

- Dieses Programm ermöglicht die Aufnahme von Bildern in monochromen Pixelformaten (Verwende: "Mono8") und RGB/BRG-Farbpixelformaten (Verwende: "RGB8Packed" oder "BGR8Packed"). Wenn eine Kamera mit nur einem Bildsensor verwendet und die Interpolation der Bayer-Formate

in RGB/BGR-Farbpixelformate nicht auf der Kamera unterstützt wird, können nur monochrome Bilder aufgenommen werden. Die Umrechnung von Bayer-Bildformaten ist im Programm derzeit nicht vorgesehen.

- *Mögliche Werte*: Mono8, RGB8Packed, BGR8Packed
- *Standardwert*: Mono8

USER_SET_SELECTOR

- Es kann ein User Set gewählt werden, das zuvor in der GUI der SDK des Kameraherstellers erstellt und gespeichert wurde. Sofern das nicht möglich oder gewünscht ist, muss Default als Wert gesetzt werden. Dann können die markierten Einstellungen, die nur genutzt werden, wenn kein User Set verwendet wird, definiert werden.
- *Mögliche Werte*: Default, UserSet1, UserSet2, UserSet3, UserSet4, UserSet5 (genaue Anzahl an User Sets ist Hersteller abhängig)
- *Standardwert*: Default

10.2.4 Bildspeicherung

FILE_NAME

- Ein beliebiger Dateiname kann zum Speichern verwendet werden. Hinter jeden Dateinamen wird der Zeitstempel des Aufnahmezeitpunkts in Millisekunden seit 01.01.1970 00:00:00 (UTC) angehängt. Der Dateiname kann auch leer gelassen werden, sodass nur der Zeitstempel verwendet wird.
- *Mögliche Werte*: Zeichenfolge ohne Anführungszeichen
- *Standardwert*: (leer)

FILE_TYPE

- Zur Wahl des Dateityps der lokal abgespeicherten Bilder.
- *Mögliche Werte*: jpg, png oder bmp
- *Standardwert*: png

10.2.5 Inferenz-Client

INPUT_TENSOR_NAME (nur für gRPC)

- Name des Input Tensors für Inferenz.
- *Mögliche Werte*: Zeichenfolge ohne Anführungszeichen
- *Standardwert*: inputs

INPUT_TENSOR_SHAPE_1ST_DIMENSION (nur für gRPC)

- Erste Dimension des Input Tensors (Batch-Größe).

- *Mögliche Werte*: Ganzzahl
- *Kein Standardwert*

INPUT_TENSOR_SHAPE_2ND_DIMENSION (nur für gRPC)

- Zweite Dimension des Input Tensors (Anzahl Pixel Höhe).
- *Mögliche Werte*: Ganzzahl
- *Kein Standardwert*

INPUT_TENSOR_SHAPE_3RD_DIMENSION (nur für gRPC)

- Dritte Dimension des Input Tensors (Anzahl Pixel Weite).
- *Mögliche Werte*: Ganzzahl
- *Kein Standardwert*

INPUT_TENSOR_SHAPE_4TH_DIMENSION (nur für gRPC)

- Vierte Dimension des Input Tensors (Anzahl Bildkanäle).
- *Mögliche Werte*: Ganzzahl
- *Kein Standardwert*

LABEL_1 (nur für gRPC)

- Erstes Label das in der Inferenz als Klasse verwendet wird. Die Labels müssen in der gleichen Reihenfolge wie beim Training hinterlegt werden. Ungenutzte Label müssen leer gelassen werden. Bis zu fünf Label sind möglich.
- *Mögliche Werte*: Zeichenfolge ohne Anführungszeichen
- *Standardwert*: (leer)

LABEL_2 (nur für gRPC)

- Zweites Label das in der Inferenz als Klasse verwendet wird. Die Labels müssen in der gleichen Reihenfolge wie beim Training hinterlegt werden. Ungenutzte Label müssen leer gelassen werden. Bis zu fünf Label sind möglich.
- *Mögliche Werte*: Zeichenfolge ohne Anführungszeichen
- *Standardwert*: (leer)

LABEL_3 (nur für gRPC)

- Drittes Label das in der Inferenz als Klasse verwendet wird. Die Labels müssen in der gleichen Reihenfolge wie beim Training hinterlegt werden. Ungenutzte Label müssen leer gelassen werden. Bis zu fünf Label sind möglich.
- *Mögliche Werte*: Zeichenfolge ohne Anführungszeichen
- *Standardwert*: (leer)

LABEL_4 (nur für gRPC)

- Viertes Label das in der Inferenz als Klasse verwendet wird. Die Labels müssen in der gleichen Reihenfolge wie beim Training hinterlegt werden. Ungenutzte Label müssen leer gelassen werden. Bis zu fünf Label sind möglich.
- *Mögliche Werte*: Zeichenfolge ohne Anführungszeichen
- *Standardwert*: (leer)

LABEL_5 (nur für gRPC)

- Fünftes Label das in der Inferenz als Klasse verwendet wird. Die Labels müssen in der gleichen Reihenfolge wie beim Training hinterlegt werden. Ungenutzte Label müssen leer gelassen werden. Bis zu fünf Label sind möglich.
- *Mögliche Werte*: Zeichenfolge ohne Anführungszeichen
- *Standardwert*: (leer)

MODEL (nur für gRPC)

- Name des Models (Künstliches Neuronales Netz).
- *Mögliche Werte*: Zeichenfolge ohne Anführungszeichen
- *Standardwert*: model

QUALITY_LABEL_1 (nur für gRPC)

- Aussage über die Qualität des ersten Labels, das in der Inferenz als Klasse verwendet wird. Die Labels und die Qualitätsaussagen müssen in der gleichen Reihenfolge wie beim Training hinterlegt werden. Ungenutzte Label müssen auch hier leer gelassen werden. Bis zu fünf Label und damit auch Qualitätsaussagen sind möglich. Für jedes hinterlegte Label muss eine Qualitätsaussage getroffen werden, ob dieses Label einer guten oder schlechten Qualität entspricht.
- *Mögliche Werte*: Bad, Good
- *Standardwert*: (leer)

QUALITY_LABEL_2 (nur für gRPC)

- Aussage über die Qualität des zweiten Labels, das in der Inferenz als Klasse verwendet wird. Die Labels und die Qualitätsaussagen müssen in der gleichen Reihenfolge wie beim Training hinterlegt werden. Ungenutzte Label müssen auch hier leer gelassen werden. Bis zu fünf Label und damit auch Qualitätsaussagen sind möglich. Für jedes hinterlegte Label muss eine Qualitätsaussage getroffen werden, ob dieses Label einer guten oder schlechten Qualität entspricht.

- *Mögliche Werte*: Bad, Good
- *Standardwert*: (leer)

QUALITY_LABEL_3 (nur für gRPC)

- Aussage über die Qualität des dritten Labels, das in der Inferenz als Klasse verwendet wird. Die Labels und die Qualitätsaussagen müssen in der gleichen Reihenfolge wie beim Training hinterlegt werden. Ungenutzte Label müssen auch hier leer gelassen werden. Bis zu fünf Label und damit auch Qualitätsaussagen sind möglich. Für jedes hinterlegte Label muss eine Qualitätsaussage getroffen werden, ob dieses Label einer guten oder schlechten Qualität entspricht.
- *Mögliche Werte*: Bad, Good
- *Standardwert*: (leer)

QUALITY_LABEL_4 (nur für gRPC)

- Aussage über die Qualität des vierten Labels, das in der Inferenz als Klasse verwendet wird. Die Labels und die Qualitätsaussagen müssen in der gleichen Reihenfolge wie beim Training hinterlegt werden. Ungenutzte Label müssen auch hier leer gelassen werden. Bis zu fünf Label und damit auch Qualitätsaussagen sind möglich. Für jedes hinterlegte Label muss eine Qualitätsaussage getroffen werden, ob dieses Label einer guten oder schlechten Qualität entspricht.
- *Mögliche Werte*: Bad, Good
- *Standardwert*: (leer)

QUALITY_LABEL_5 (nur für gRPC)

- Aussage über die Qualität des fünften Labels, das in der Inferenz als Klasse verwendet wird. Die Labels und die Qualitätsaussagen müssen in der gleichen Reihenfolge wie beim Training hinterlegt werden. Ungenutzte Label müssen auch hier leer gelassen werden. Bis zu fünf Label und damit auch Qualitätsaussagen sind möglich. Für jedes hinterlegte Label muss eine Qualitätsaussage getroffen werden, ob dieses Label einer guten oder schlechten Qualität entspricht.
- *Mögliche Werte*: Bad, Good
- *Standardwert*: (leer)

SERVING_HOST

- Hostname oder IP-Adresse des Inferenz-Servers.
- *Mögliche Werte*: Zeichenfolge ohne Anführungszeichen
- *Kein Standardwert*

SERVING_INTERFACE

- Zur Wahl steht eine Kommunikationsschnittstelle über REST oder gRPC zur Kommunikation mit dem Inferenz-Server.
- *Mögliche Werte*: REST, gRPC
- *Kein Standardwert*

SERVING_PORT

- Port des Inferenz-Servers, normalerweise wird Port 8505 für REST und 8506 für gRPC verwendet.
- *Mögliche Werte*: Vierstellige Ganzzahl
- *Kein Standardwert*

SIGNATURE_NAME (nur für gRPC)

- Name der Serving Signature. Die Serving Signature spezifiziert den Typ des Models, das beim Training mit TensorFlow exportiert wird.
- *Mögliche Werte*: Zeichenfolge ohne Anführungszeichen
- *Standardwert*: serving_default

10.3 Genutzte Cognex Native Commands

Nachfolgend sind alle im Programm verwendeten Native Commands mit einer kurzen Beschreibung, der Syntax, den Eingabeparametern und den Ausgabestatuscodes aufgelistet. Detailliertere Informationen sowie weitere Native Commands sind in der Cognex Dokumentation zum Native Mode Protokoll zu finden [Cog20a].

Get Online

Gibt den Online-Status des In-Sight Vision-Systems zurück.

Syntax: GO

Eingabeparameter: keine

Ausgabestatuscodes:

- 0: Das In-SightVision-System ist derzeit offline.
- 1: Das In-SightVision-System ist derzeit online.

Reset System

Setzt den In-Sight-Sensor zurück. Dieser Befehl ähnelt dem physischen Ein- und Ausschalten des Sensors.

Syntax: RT

Eingabeparameter: keine

Ausgabestatuscodes:

- -6: Der Benutzer hat keinen Vollzugriff zum Ausführen des Befehls.

Set Online

Setzt den In-Sight-Sensor in den Online- oder Offline-Modus.

Syntax: SO[Int]

Eingabeparameter:

- 0: Setzt den In-Sight-Sensor offline
- 1: Setzt den In-Sight-Sensor online

Ausgabestatuscodes:

- 1: Der Befehl wurde erfolgreich ausgeführt.
- 0: Nicht erkannter Befehl.
- -1: Der für Int angegebene Wert liegt entweder außerhalb des Bereichs oder ist keine gültige ganze Zahl.
- -2: Der Befehl konnte nicht ausgeführt werden.
- -5: Die Kommunikationsflagge war erfolgreich, aber der Sensor wurde nicht online geschaltet, da der Sensor manuell über die Benutzeroberfläche von In-Sight Explorer oder durch ein diskretes E/A-Signal offline geschaltet wurde.
- -6: Der Benutzer hat keinen Vollzugriff zum Ausführen des Befehls.

Set Job

Lädt einen Job aus einem der Job-Slots im Flash-Speicher des In-Sight-Sensors und macht ihn damit zum aktiven Job.

Syntax: SJ[ID]

Eingabeparameter:

- ID: ID-Nummer des zu ladenden Jobs (0 bis 999)

Ausgabestatuscodes:

- 1: Der Befehl wurde erfolgreich ausgeführt.
- 0: Nicht erkannter Befehl.
- -1: Die ID ist kleiner als 0 oder ist keine ganze Zahl.
- -2: Der Auftrag konnte nicht geladen werden, der Sensor ist Online oder die Datei wurde nicht gefunden, so dass der Befehl nicht ausgeführt werden konnte.
- -4: Der In-Sight-Sensor hat keinen Speicherplatz mehr.
- -6: Der Benutzer hat keinen Vollzugriff zum Ausführen des Befehls.

Set and Wait

Löst ein bestimmtes Ereignis aus und wartet, bis der Befehl abgeschlossen ist, um eine Antwort zurückzugeben.

Syntax: SW[Int]

Eingabeparameter:

- 0 bis 7: Gibt einen Soft-Trigger an (Soft 0, Soft 1, ... Soft 7)
- 8: Erfassen eines Bildes. Für diese Option muss der Trigger-Parameter der Funktion AcquireImage auf Extern, Manuell oder Netzwerk eingestellt sein.

Ausgabestatuscodes:

- 1: Der Befehl wurde erfolgreich ausgeführt.
- 0: Nicht erkannter Befehl.
- -1: Die Zahl liegt entweder außerhalb des Bereichs (0 bis 8) oder ist keine ganze Zahl.
- -2: Der Befehl konnte nicht ausgeführt werden, oder der Sensor ist offline.
- -6: Der Benutzer hat keinen Vollzugriff zum Ausführen des Befehls.

Get Value

Gibt den Inhalt eines angegebenen symbolischen Tags zurück, z.B. ein EasyBuilder Positions- oder Inspektions-Tool-Ergebnis oder Jobdaten.

Syntax: GV[„Symbolic Tag“]

Eingabeparameter:

- „Symbolic Tag“: Der Name des symbolischen Tags, z. B. ein Positions- oder Inspektions-Tool-Ergebnis oder Job-Daten (z. B. "Job.Robot.FormatString.", "Job.FormatString").

Ausgabestatuscodes:

- 1: Der Befehl wurde erfolgreich ausgeführt.
- 0: Nicht erkannter Befehl.
- -1: Der "Symbolic Tag" ist ungültig.
- -2: Der Befehl konnte nicht ausgeführt werden.

10.4 Module

In Tab. 10.1 sind alle Module mit ihren entsprechenden Klassen aufgelistet. Abstrakte Basisklassen vererben abstrakte Methoden an ihre Erben, die von diesen ausdefiniert werden müssen. Die Klassendiagramme sind ebenfalls nachfolgend

zur Veranschaulichung der Attribute, der Methoden und einer etwaigen Vererbung dargestellt.

Tab. 10.1: Modulübersicht mit enthaltenen Klassen

Modul	**Klassen**		
cameras	CamGeneral	GenICam	Cognex
trigger	MqttTrigger	ContinuousTrigger	
saving	LocalSaver		
inference	BaseInferenceClient	RestInferenceClient	GrpcInferenceClient

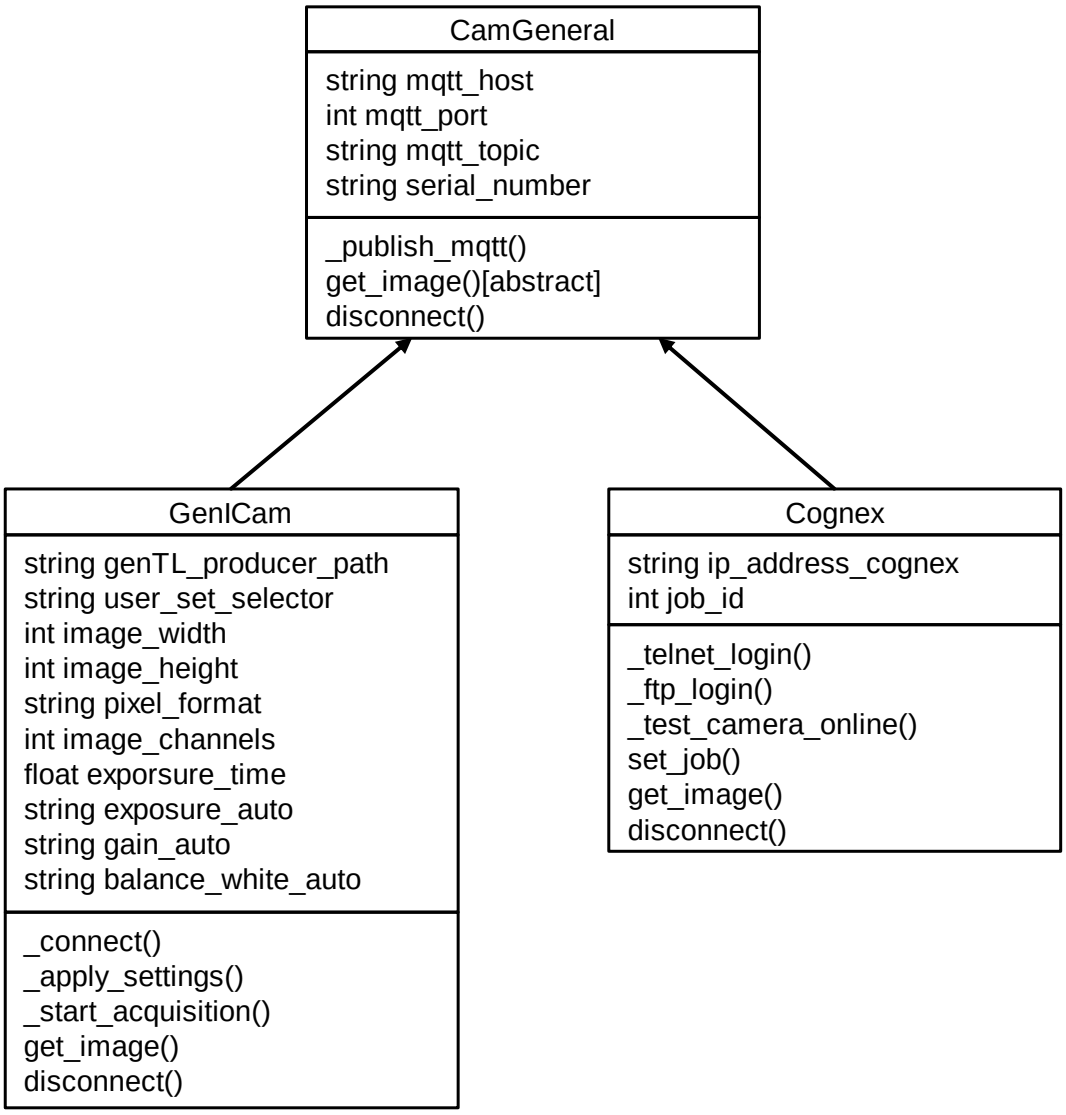


Abb. 10.1: Vereinfachtes Klassendiagramm für Modul cameras.py

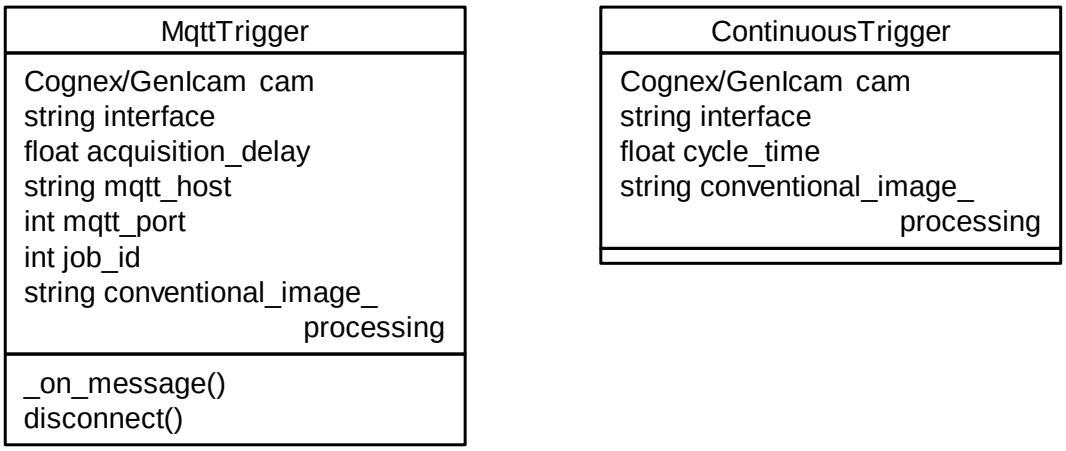


Abb. 10.2: Vereinfachtes Klassendiagramm für Modul trigger.py

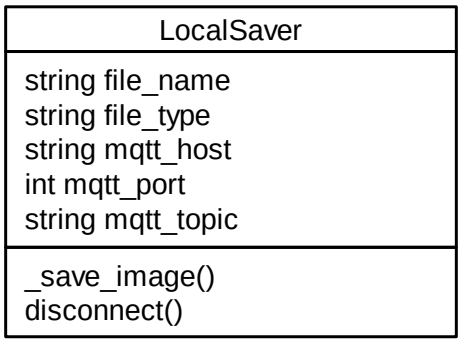


Abb. 10.3: Vereinfachtes Klassendiagramm für Modul saving.py

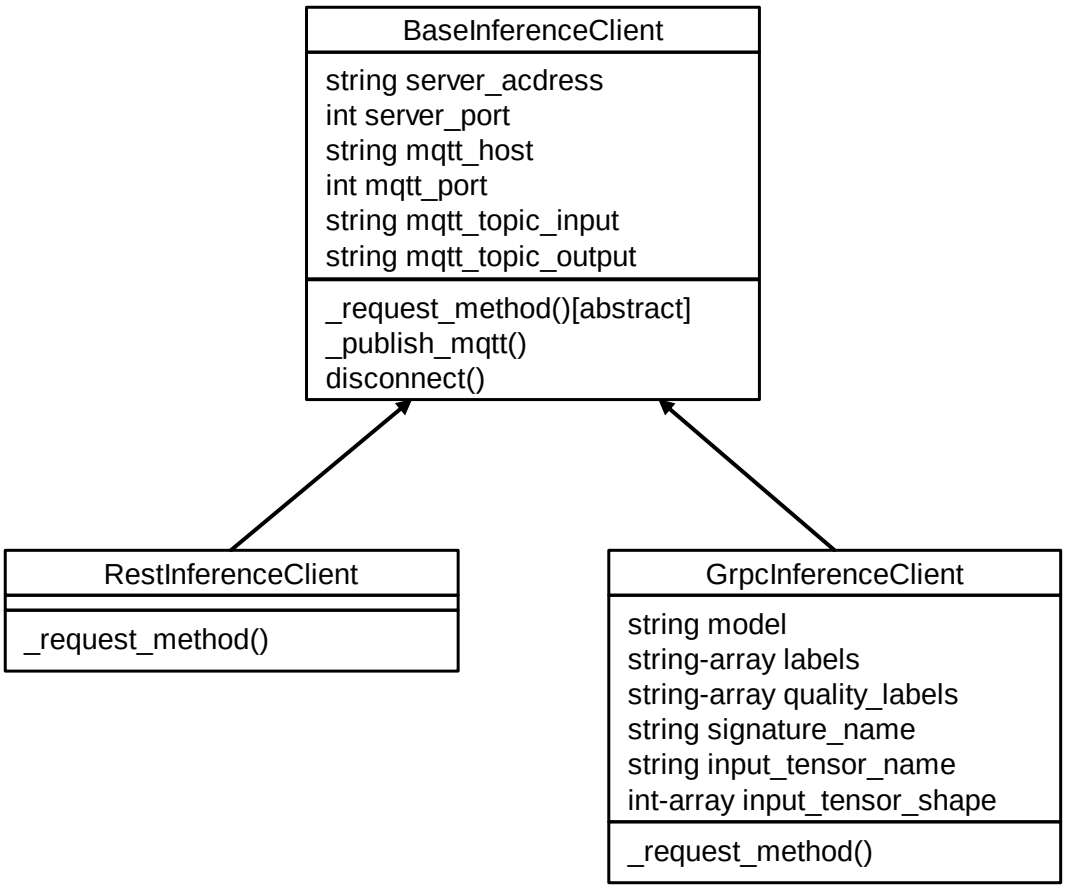


Abb. 10.4: Vereinfachtes Klassendiagramm für Modul inference.py

10.4.1 cameras.py

```
"""
Classes to connect, configurate and get image data from cameras.

The module provides two classes:
- GenICam: for all GenICam compatible cameras. Cameras with GigE
            Vision or USB3 Vision transport layer always support
            GenICam.
- Cognex: for all cognex vision sensors and vision systems of
            the in-sight series.

"""

# Import python in-built libraries
from abc import ABC, abstractmethod
import time
import base64
import json
import telnetlib
from ftplib import FTP
```

```
import os
import sys

# Import libraries that had been installed with pip install
import paho.mqtt.client as mqtt
import cv2
import numpy as np

# Import libraries that are only needed for GenICam
from genicam.gentl import TimeoutException
from harvesters.core import Harvester

class CamGeneral(ABC):
    """
    Abstract base clase for the different cameras.
    This class defines only the basic constructor, the
    method _publish_mqtt() to publish the results to the MQTT
    broker, the method disconnect() and the abstract method
    get_image(). Children must define the get_image() method.

    Args of constructor:
        mqtt_host[string]:      Hostname or IP address of the MQTT
 broker
        mqtt_port[int]:         Network port of the server host to
 connect to
        mqtt_topic[string]:     Topic on MQTT Broker where trigger
 signal is send to
                                (e.g. "test/trigger/")
        serial_number[string]:  Serial number of the camera set
                                by user

    Returns of constructor:
        See inheritors
    """

    def __init__(self,mqtt_host,mqtt_port,mqtt_topic,serial_number
) -> None:
        """
        Base class constructor configures the object with the
        MQTT host settings.

        Args:
            see class description

        Returns:
            see class description
        """
```

```
        self.mqtt_host = mqtt_host
        self.mqtt_port = mqtt_port
        self.mqtt_topic = mqtt_topic
        self.serial_number = serial_number

        # Connect to the Broker, default port for MQTT 1883
        self.client = mqtt.Client()
        self.client.connect(self.mqtt_host, self.mqtt_port)
        print("Connected to MQTT broker.")
        self.client.loop_start()

    def _publish_mqtt(self,image,serial_number,overall_quality=Non
e) -> None:
        """
        Sends the timestamp of the time at at which the image was
        taken, the image itself, and optionally for Cognex cameras
        the overall quality based on conventional image processing
        to the MQTT broker. Therefore, the image is first
        converted from a numpy array into a byte array and from
        a byte array into a string.

        overall_quality is only needed if using conventional
        image processing methods with cognex.
        If overall_quality is None, this key/information is not
        put into the MQTT message.

        The MQTT message contains in one json:
            - serial number of the camera
            - timestamp of the acquisition time in ms since epoch
            - image information:
                - string containing the image data (image_bytes)
                - image height, width and channels
            - overall quality (only if using conventional image
                processing on cognex)

        Json format:
            {
            'serial_number': serial number,
            'timestamp_ms': timestamp_ms,
            'image':
                    {'image_bytes': encoded_image,
                    'image_height': image.shape[0],
                    'image_width': image.shape[1],
                    'image_channels':image.shape[2]},
            'overall_quality': overall_quality,
            }
```

```
        Args:
            image[np.ndarray]:      Array of image in BGR color
                                    format with array size
                                    N x M x image_channels
                                    where N is height, M is width
                                    and image channels the number
                                    of bytes per pixel
            overall_quality[int]:   Use 0 for quality not ok and
                                    1 for quality ok

        Returns:
            None
        """
        # Get timestamp of time  when trigger was received.
        #   Measured in ms since epoch. Epoch is defined as
        #   January 1, 1970, 00:00:00 (UTC)
        timestamp_ms = int(round(time.time() * 1000))

        # Encode numpy array in byte array
        # Use decode() to convert the bytes to a string to send
        #   them in a json message
        encoded_image = base64.b64encode(image).decode()
        # Preparation of the message that will be published
        prepared_message = {
            'serial_number': self.serial_number,
            'timestamp_ms': str(timestamp_ms),
            'image':
                    {'image_bytes': encoded_image,
                    'image_height': image.shape[0],
                    'image_width': image.shape[1],
                    'image_channels':image.shape[2]},
        }

        if overall_quality is not None:
            print("Overall quality: " + str(overall_quality))
            # Add overall_quality to prepared_message dictionary
            prepared_message['overall_quality'] = str(overall_qual
ity)

        # Get json formatted string, convert python object into
        #   json object
        message = json.dumps(prepared_message)
        # Publish the message
        ret = self.client.publish(self.mqtt_topic, message,qos=0)

        print("Image sent to MQTT broker.")
```

```
    @abstractmethod
    def get_image(self) -> None:
        """
        Must be defined by children.

        Function to get an image from the camera and uses
        _publish_mqtt to send it to MQTT broker.

        Args:
            None

        Returns:
            None
        """
        pass

    def disconnect(self) -> None:
        """
        Disconnects from MQTT broker.

        Args:
            None

        Returns:
            None
        """
        self.client.loop_stop()
        self.client.disconnect()
        print("Disconnected from MQTT broker.")

class GenICam(CamGeneral):
    """
    This class is for all GenICam compatible cameras. Cameras
    with GigE Vision or USB3 Vision transport layer always
    support GenICam. Each instance of this class will be
    automatically connected to the device.

    The class inherits all methods from the base class CamGeneral.

    (see description of CamGeneral) Because of that this class has
    to define the abstract method get_image(). In this class
    get_image() fetchs an image out of the buffer/image stream and
    makes sure that data format is BGR8 to send it to MQTT broker
    with _publish_mqtt().

    Additional methods of the GenICam class:
    The first method _connect() establishs a connection to the
```

```
GenICam camera. Afterwards, _apply_settings() applies either
a configurated user set of configurations or the entered
settings in the arguments for the class instance. The user
set of configurations can be created in the matrix vision
wxPropView or in most SDK which is provided by the camera
manufacturer. If no user set of configurations is used
and no settings are provided in the arguments, the default
settings of the camera will be used.
The method start _start_acquisition() starts the image stream
of the camera.
The last method deactivate() disconnects from camera.

Some of the comments in this class are copied from:
https://github.com/genicam/harvesters/blob/master/README.rst

Args of constructor:
    mqtt_host[string]:          Hostname or IP address of the
                                MQTT broker
    mqtt_port[int]:             Network port of the server
                                host to connect to
    mqtt_topic[string]:         Topic on MQTT Broker where
                                trigger signal is send to
                                (e.g. "test/trigger/")
    serial_number[string]:      Serial number of the camera
                                set by user
    genTL_producer_path[string]:
                                Path to the *.cti file that
                                is used to connect to camera
    (opt.) user_set_selector[string]:
                                Use an already pre-configu-
                                rated user set.
                                Possible values: "Default",
                                    "UserSet1", "UserSet2",
                                    "UserSet3", "UserSet4",
                                    "UserSet5" (The number of
                                    user sets is camera
                                    dependent.)
                                Default value: "Default"
    (opt.) image_width[int]:    Determine in pixels the region
                                of interest (ROI). ROI will be
                                always centered in camera
                                sensor.
                                A value higher than maximum
                                resolution of the camera will
                                set maximum values instead of
                                the values entered here.
```

```
                                To find out the highest value,
                                search for the resolution in
                                the specifications of the
                                camera.
                                Specifications are available
                                in the manual or on the
                                website where you bought the
                                camera.
                                Default: None
(opt.) image_height[int]:       see image_width
                                Default: None
(opt.) pixel_format[string]:
                                Set the pixel format you want
                                to use. This Programm allows
                                you to take pictures in
                                monochrome pixel formats
                                (use: "Mono8") and RGB/BRG
                                color pixel formats (use:
                                "RGB8Packed" or "BGR8Packed")
                                If you only have a camera with
                                only one image sensor, you can
                                only take monochrome images.
                                Possible values: "Mono8",
                                    "RGB8Packed", "BGR8Packed"
                                Default value: None
(opt.) image_channels[int]: Number of channels (bytes per
                                pixel) that are used in the
                                array (third dimension of the
                                image data array).You do not
                                have to set this value.
                                If None, the best number of
                                channels for your set pixel
                                format will be used
                                Possible Values: 1, 3
                                Default value: None
(opt.) exposure_time[float]:
                                Set the exposure time manually
                                Default value: None
(opt.) exposure_auto[string]:
                                Determine if camera should
                                automatically adjust the
                                exporsure time.
                                Your settings will only be
                                executed if  camera supports
                                this. You do not have to check
                                if the camera supports this.
                                Possible values are:
```

```
                                        - "Off":  No automatic
                                            adjustment
                                        - "Once": Adjusted once
                                        - "Continuous": Continuous
                                            adjustment (not
                                            recommended,
                                            Attention: This could
                                            have a big impact on
                                            the frame rate of your
                                            camera)
                                    Default value: None
        (opt.) gain_auto[string]:   Determine if camera should
                                    automatically adjust the gain.
                                    Your settings will only be
                                    executed if  camera supports
                                    this. You do not have to check
                                    if the camera supports this.
                                    Possible values are:
                                        see exposure_auto
                                    Default value: None
        (opt.) balance_white_auto[string]:
                                    Determine if camera should
                                    automatically adjust the
                                    white balance.
                                    Your settings will only be
                                    executed if  camera supports
                                    this. You do not have to check
                                    if the camera supports this.
                                    Possible values are:
                                        see exposure_auto
                                    Default value: None

    Returns of constructor:
        A configurated and conected instance of GenICam ready to
        fetch an image.
    """

    def __init__(self,mqtt_host,mqtt_port,mqtt_topic,serial_number
,genTL_producer_path,user_set_selector="Default",image_width=None,
image_height=None,pixel_format=None,image_channels=None,exposure_t
ime=None,exposure_auto=None,gain_auto=None,balance_white_auto=None
) -> None:
        """
        Defines the settings for the camera configuration and
        establish a connection to the GenICam camera.

        Args:
```

```
            see class description

        Returns:
            see class description
        """
        super().__init__(mqtt_host=mqtt_host,
                         mqtt_port=mqtt_port,
                         mqtt_topic=mqtt_topic,
                         serial_number=serial_number)

        self.genTL_producer_path = genTL_producer_path
        self.user_set_selector = user_set_selector
        self.image_width = image_width
        self.image_height = image_height
        self.pixel_format = pixel_format
        self.image_channels = image_channels
        self.exposure_time = exposure_time
        self.exposure_auto = exposure_auto
        self.gain_auto = gain_auto
        self.balance_white_auto = balance_white_auto

        # Connect to camera
        self._connect()

        # Apply configurations
        self._apply_settings()

        # Start acquisition
        self._start_acquisition()

    def _connect(self) -> None:
        """
        Establishes with the set GenTL Producer a connection to
        the GenICam camera. Also some default settings are done.

        Args:
            None
        Returns:
            None
        """

        # Instantiate a Harvester object to use harvesters core
        self.h = Harvester()

        # Add path of GenTL Producer
        self.h.add_file(self.genTL_producer_path)
        # Check if cti-file available, stop if none found
```

```
        if len(self.h.files) == 0:
            sys.exit("No valid cti file found at %s" % self.genTL_
producer_path)
        print("Currently available genTL Producer CTI files: ", se
lf.h.files)

        # Update the list of remote devices; fills up your device
        #   information list; multiple devices in list possible
        self.h.update()
        # If no remote devices in the list that you can control
        if len(self.h.device_info_list) == 0:
            sys.exit("No compatible devices detected.")
        # Show remote devices in list
        print("Available devices List: ", self.h.device_info_list)


        # Create an image acquirer object specifying a target
        #   remote device
        # As argument also user_defined_name, serial_number,
        #   vendor, model, etc. possible
        # If multiple cameras in device list, choose the right
        #   one by changing the list_index or by using another
        #   argument
        # Multiple devices not supported yet in this programm
        self.ia = self.h.create_image_acquirer(list_index=0)
        print("Using device: ", self.h.device_info_list[0])

        ## Set some default settings
        # This is required because of a bug in the harvesters
        #   module. This should not affect our usage of image
        #   acquirer. Only change if you know what you are doing
        self.ia.remote_device.node_map.ChunkModeActive.value = Fal
se

        # The number of buffers that is prepared for the image
        #   acquisition process. The buffers will be announced
        #   to the target GenTL Producer. Need this so that we
        #   always get the correct actual image.
        self.ia.num_buffers = 1

    def _apply_settings(self) -> None:
        """
        Applies the settings for the camera.

        Either a configurated user set of configurations or the
        entered settings in the arguments for the class instance.
        The user set of configurations can be created in the
```

```
        matrix vision wxPropView or in most SDK which is provided
        by the camera manufacturer.
        If no user set of configurations is used and no settings
        are provided in the arguments, the default settings of
        the cameras will be used.

        The automatic adjust settings are only applied if camera
        supports these features.

        Args:
            None

        Returns:
            None
        """

        # If camera was already configurated and configurations
        #   has been saved in user set, then set and load user
        #   set here and return
        if self.user_set_selector is not "Default":
            self.ia.remote_device.node_map.UserSetSelector.value =
 self.user_set_selector
            self.ia.remote_device.node_map.UserSetLoad.execute()
            # Do not execute the code afterwards in this function
            #   if user-set is used
            return

        # Set Width
        if self.image_width is not None:
            if self.image_width > self.ia.remote_device.node_map.W
idthMax.value:
                # Value given in settings higher than max
                #   -> set max
                self.ia.remote_device.node_map.Width.value = self.
ia.remote_device.node_map.WidthMax.value
            else:
                # Set value given in settings
                self.ia.remote_device.node_map.Width.value = self.
image_width

        # Set Height
        if self.image_height is not None:
            if self.image_height > self.ia.remote_device.node_map.
HeightMax.value:
                # Value given in settings higher than max
                #   -> set max
                self.ia.remote_device.node_map.Height.value = self
.ia.remote_device.node_map.HeightMax.value
            else:
```

```
                # Set value given in settings
                self.ia.remote_device.node_map.Height.value = self
.image_height

        # Set ROI always centered in camera sensor
        # Therefor calculate Offset X and Offset Y where the
        #   readout region should start and assign it to features
        if self.user_set_selector is not "Default":
            self.ia.remote_device.node_map.OffsetX.value = int((se
lf.ia.remote_device.node_map.WidthMax.value - self.ia.remote_devic
e.node_map.Width.value)/2)
            self.ia.remote_device.node_map.OffsetY.value = int((se
lf.ia.remote_device.node_map.HeightMax.value - self.ia.remote_devi
ce.node_map.Height.value)/2)

        # Set PixelFormat
        if self.pixel_format is not None:
            self.ia.remote_device.node_map.PixelFormat.value = sel
f.pixel_format

        # Set Exposure time
        if self.exposure_auto is not None:
            self.ia.remote_device.node_map.ExposureTimeAbs.value =
 self.exposure_time

        # Get list of all available features of the camera
        node_map = dir(self.ia.remote_device.node_map)

        # Set ExposureAuto, GainAuto and BalanceWhiteAuto;
        #   it always first check if connected camera supports
        #   this function
        if self.exposure_auto is not None:
            if "ExposureAuto" in node_map:
                self.ia.remote_device.node_map.ExposureAuto.value
= self.exposure_auto
            else:
                print("Camera does not support automatic adjustmen
t of exposure time")
        if self.gain_auto is not None:
            if "GainAuto" in node_map:
                self.ia.remote_device.node_map.GainAuto.value = se
lf.gain_auto
            else:
                print("Camera does not support automatic adjustmen
t of gain")
        if self.balance_white_auto is not None:
            if "BalanceWhiteAuto" in node_map:
                self.ia.remote_device.node_map.BalanceWhiteAuto.va
lue = self.balance_white_auto
```

```
            else:
                print("Camera does not support automatic adjustmen
t of white balance")

    def _start_acquisition(self) -> None:
        """
        Activate an image stream from camera to be able to fetch
        images out of stream.

        Args:
            None

        Returns:
            None
        """
        # Starts image acquisition with harvesters
        self.ia.start_acquisition()
        print("Acquisition started.")

    # Get image out of image stream
    def get_image(self) -> None:
        """
        Fetch an image out of the image stream and make sure if
        colored image that BGR pixel format is used.

        Args:
            None

        Returns:
            None
        """

        # Try to fetch a buffer that has been filled up with an
        #   image
        try:
            # Default value
            retrieved_image = None

            # To solve the problem that buffer is already filled
            #   with an old image, but we want the newest image,
            #   This here is probably not the best way to solve
            #   the problem. It is a workaround.
            with self.ia.fetch_buffer(timeout=20) as buffer:
                # Do not use this buffer, use the next one
                pass

            # Due to with statement buffer will automatically be
            #   queued
```

```
            with self.ia.fetch_buffer(timeout=20) as buffer:
                print("Image fetched.")
                # Create an alias of the 2D image component:
                component = buffer.payload.components[0]
                # Note that the number of components can be vary.
                #   If your target remote device transmits a
                #   multi-part information, then you'd get two or
                #   more components in the payload. However, this
                #   programs works with a remote device that
                #   transmits only a 2D image. So we manipulate
                #   only index 0 of the list object, components.

                # As we record only two-dimensional pictures the
                #   third position in shaps shows the number of
                #   color values.
                #  Mono8 is only black and white -> one color
                #   value for each pixel (one byte per pixel)
                #   -> image_channel = 1
                # RGB/BGR is red, green and blue -> three color
                #   values for each pixel to determine its
                #   overall color (three bytes per pixel)
                #   -> image_channel = 3

                # Check if Mono or RGB/BGR to determine argument
                #   for shape of the array of retrieved image
                #   If number of channels is set in arguments,
                #   do not change anything, only change if None
                data_format = component.data_format
                if self.image_channels is None:
                    if data_format =="Mono8":
                        self.image_channels = 1
                    elif data_format == "RGB8" or data_format == "
BGR8":
                        self.image_channels = 3
                    else:
                        sys.exit("Unsupported pixel format: %s" %
data_format)

                # Generate out of buffer data, which is an
                #   [1x(N+M+image_channels)x1] array, an
                #   [N x M x image_channels] array where N is
                #   height and M is width of image
                retrieved_image = np.ndarray(buffer=component.data
.copy(),
                                             dtype=np.uint8,
                                             shape=(component.height, compo
nent.width, self.image_channels))
```

```
                # Adjust the order of red, blue and green color
                #   to BGR which is default in opencv
                if data_format == "RGB8":
                    retrieved_image = cv2.cvtColor(retrieved_image
, cv2.COLOR_RGB2BGR)

            print("Image converted. Ready to publish.")

            self._publish_mqtt(retrieved_image,self.serial_number)

        # If TimeoutException because no image was fetchable,
        #   restart the acquisition process
        except TimeoutException:
            print("Timeout ocurred during fetching an image. Camer
a reset and restart.")
            self.ia.destroy()
            self.h.reset()
            self._connect()
            self._apply_settings()
            self._start_acquisition()
            print("Camera restarted. Ready to fetch an image.")


    def disconnect(self) -> None:
        """
        Deactivate acquisition and disconnect from camera.

        Args:
            None

        Returns:
            None
        """
        # Close the connection to ...
        # ... MQTT (and stop loop)
        CamGeneral.disconnect(self)

        # ... GenICam (destroy image acquirer)
        self.ia.destroy()
        self.h.reset()

        print("Disconnected from GenICam camera.")

class Cognex(CamGeneral):
    """
    This class is for all Cognex cameras of the In-Sight series.
```

```
    Every instance is automatically connected to MQTT to publish
    images to the MQTT broker.

    This class provides a _telnet_login() and _ftp_login() method
    to establish connections to Telnet and FTP. With
    test_camera_online() the camera is set in online mode and
    with set_job() the job_id is used to configurate the camera.
    The get_image() function to acquire a single image with
    telnet and ftp is defined. The last method is disconnect().

    Some comments are copied from:
    https://support.cognex.com/docs/is_590/web/EN/ise/Content/Comm
unications_Reference/NativeMode_Commands.htm

    Args of constructor:
        mqtt_host[string]:          Hostname or IP address of the
                                    MQTT broker
        mqtt_port[int]:             Network port of the server
                                    host to connect to
        mqtt_topic[string]:         Topic on MQTT Broker where
                                    trigger signal is send to
                                    (e.g. "test/trigger/")
        serial_number[string]:      Serial number of the camera
                                    set by user
        ip_adress_cognex[string]:   IP adress of cognex camera
        job_id[int]:                Job ID number

    Returns of constructor:
        A configurated and conected instance of Cognex ready to
        acquire an image.
    """

    def __init__(self,mqtt_host,mqtt_port,mqtt_topic,serial_number
,ip_adress_cognex,job_id):
        """
        Defines the settings for the camera configuration and
        connects to MQTT.
        Calls _telnet_login(), _ftp_login(), test_camera_online()
        and set_job() to establish a valid connection to the
        camera and to configurate the camera settings.

        Args:
            see class description

        Returns:
            see class description
        """
```

```
        super().__init__(mqtt_host=mqtt_host,
                         mqtt_port=mqtt_port,
                         mqtt_topic=mqtt_topic,
                         serial_number=serial_number)

        # Default user name of all cognex cameras
        self.user = 'admin'
        # Default password of all cognex cameras
        self.password = ''
        self.ip_adress_cognex = ip_adress_cognex
        self.job_id = job_id

        # Telnet login
        self._telnet_login()
        # Test if camera is in online mode
        self._test_camera_online()

        # Set job
        if job_id is not None:
            self.set_job(self.job_id)

        # FTP login
        self._ftp_login()

    def _telnet_login(self):
        """
        Establish telnet connection with device at given IP adress

        and log in with default log-in credentials of cognex
        (User: "Admin", Password: "")
        Afterwards tn (obj/class: Telnet)is available as logged
        in Telnet interface.

        Args:
            None

        Returns:
            None
        """

        # Get new telnet object, default port of Cognex cameras
        #   is port=23
        self.tn = telnetlib.Telnet(self.ip_adress_cognex,port=23,t
imeout=None)
        # Read the line to make sure not to use it in further
        #   steps
        self.tn.read_until(b'\n')
        print("Telnet connection established.")
```

```
        # Enter login data
        # Add enter after admin for telnet
        telnet_user = self.user+'\r\n'
        # Send the user name in ASCII format
        self.tn.write(telnet_user.encode('ascii'))
        # Add enter after password for telnet
        telnet_password = self.password + '\r\n'
        # Send the password in ASCII format
        self.tn.write(telnet_password.encode('ascii'))
        # Read the line to make sure not to use it in further
        #   steps
        self.tn.read_until(b'\n')
        print("Telnet logged in.")

    def _ftp_login(self) -> None:
        """
        Establish ftp connection with device at given IP adress
        and log in with default log-in credentials of cognex
        (User: "Admin")

        Args:
            None

        Returns:
            None
        """

        # Connect to remote server (cognex camera) and ftp login
        # Use ip_adress_cognex to connect
        self.ftp = FTP(host=self.ip_adress_cognex,timeout=None)
        # Use cognex default login credentials (User: "admin")
        self.ftp.login(self.user)
        print("FTP logged in.")

    def _test_camera_online(self) -> None:
        """
        Makes sure camera is online. Otherwise resets camera and
        logs in again.

        Args:
            None

        Returns:
            None
```

```
        """

        # Send native command GO to get online state of camera
        self.tn.write(b'GO\r\n')
        # Read next line -> information about online state
        line = self.tn.read_until(b'\n')
        # Check return about online state
        if line.decode().strip() == "1":
            print("Camera is online")
        elif line.decode().strip() == "0":
            print("Camera is offline. Reset camera.")
            # Reset camera to restart (default after restart
            #   should be online)
            self.tn.write(b'RT\r\n')
            # Give time to shutdown and restart again
            print("Sleeping to wait for restart.")
            time.sleep(60)
            print("Time over! Continuing in code after restart.")

            # Again Telnet login
            self.tn = self._telnet_login()

            # Send again native command GO to get online state
            #   of camera
            self.tn.write(b'GO\r\n')
            # Read next line -> information about online state
            line = self.tn.read_until(b'\n')
            # Check return about online state
            if line.decode().strip() == "1":
                print("Camera is online")
            elif line.decode().strip() == "0":
                disconnect()
                sys.exit("Cognex Error: Reset to set online not su
ccesful ||| You have to set the camera online manually in Cognex I
nsight Explorer. You can avoid this error by selecting the box for
 'Start the Sensor in Online Mode' in the 'Startup Options' under
'Save Job' in the In-Sight Explorer.")

    def set_job(self,job_id) -> None:
        """
        To load a job, the camera first must be set offline.
        Then load the job and set online again.

        This command only works on a job saved on the flash memory
        on the In-Sight vision system. If the job is stored on the
        configured Job Server or the SD Card installed to the
        In-Sight vision system, this command does not work.
```

```
        To use the job ID number feature, the job to be loaded
        must be saved with a numerical prefix of 0 to 999.

        Args:
            job_id[int]:    Job ID number

        Returns:
            None
        """
        # Proof if valid job id
        if not job_id in range(0,1000):
            sys.exit("Environment Error: JOB_ID not allowed ||| Th
e job to be loaded must be saved with a numerical prefix of 0 to 9
99. Make sure to use a job_id in this range.")

        # Set offline
        self.tn.write(b'SO0\r\n')
        # Read next line -> information about status code
        line = self.tn.read_until(b'\n')  # Check for new line
        # Check if executed correctly
        if line.decode().strip() == "1":
            print("Camera set offline")
        else:
            sys.exit("Cognex Error: Native command to set offline
failed ||| Make sure that the command is not blocked by In-
Sight Explorer.")

        # Set Job
        prepared_command = 'SJ'+str(job_id)+'\r\n'
        self.tn.write(prepared_command.encode('ascii'))
        # Read next line -> information about status code
        line = self.tn.read_until(b'\n')  # Check for new line
        # Check if executed correctly
        if line.decode().strip() == "1":
            print("Job " + str(job_id) + " is set.")
        else:
            sys.exit("Cognex Error: Native command to set job fail
ed ||| Make sure the job is configurated with In-
Sight Explorer and saved with the set job_id as prefix on the came
ra.")

        # Set online
        self.tn.write(b'SO1\r\n')
        # Read next line -> information about status code
        line = self.tn.read_until(b'\n')  # Check for new line
        # Check if executed correctly
        if line.decode().strip() == "1":
            print("Camera set online")
```

```
        else:
            sys.exit("Cognex Error: Native command to set online f
ailed ||| Make sure the camera is set online after restart. Check
if the box for 'Start the Sensor in Online Mode' in the 'Startup O
ptions' under 'Save Job' in the In-Sight Explorer is selected.")

    def get_image(self,conventional_image_processing=False) -
> None:
        """
        Trigger a new image over Telnet and get the new image
        over FTP to send it to the publsihing function.

        If conventional_image_processing is True, it gets the
        contents of the symbolic tag, FormatString from the
        Cognex. Define the values in the FormatString in Cognex
        Insight Explorer. The values in the FormatString should
        be only 0 or 1 where 0 stands for quality is bad and 1
        for quality is good. You can put a few values into the
        string of different Cognex inspection tools. The program
        will evaluate the string. If one value is stated as 0,
        the overall_quality is declared as not okay. Only if
        all values are 1, the quality is good. The
        overall_quality is then also put into the MQTT message.
        If conventional_image_processing is False, nothing is
        get from Cognex camera or put into MQTT. Default is
        False. Only use it if FormatString is defined in Cognex
        Insight Explorer. If FormatString is defined in Insight
        Explorer but you do not need it, set False to make code
        faster in execution. Do not use the German version of
        Cognex Insight Explorer. Because the native commands are
        language sensitive.

        Args:
            conventional_image_processing[bool]:
                            Set True to get value from cognex,
                            default: False

        Returns:
            None
        """

        # If telnet connection timed out, establish new
        #   connection
        try:
            # First trigger an image with native command
            self.tn.write(b'SW8\r\n')
            # Read next line -> information about status code
            line = self.tn.read_until(b'\n')
```

```
            # Check if executed correctly
            if line.decode().strip() == "1":
                print("Image acquired")

            overall_quality=None
            if conventional_image_processing == True:
                print("Conventional image processing activated")
                # Get "FormatString" which is defined in Cognex
                #   Vision Insight Explorer with native command
                self.tn.write(b'GVJob.FormatString\r\n')
                # Read next line -> information about status
                #   code
                line = self.tn.read_until(b'\n')
                # Check if executed correctly
                if line.decode().strip() == "1":
                    print("Get value executed successfully")
                else:
                    print("Symbolic Tag is invalid. Check the nati
ve command. Check if Insight Explorer is used in English version."
)

                # Read next line -> value of specified smybolic
                #   tag
                line = self.tn.read_until(b'\n')
                print("Value from cognex: " + str(line))
                # Remove \r\n at the end and split the values
                #   of the string
                values=line.decode().strip()
                values_splitted =values.split(",")
                # Quality level (0: bad quality, 1: good quality)
                # If one value is 0, overall_quality is 0
                for value in values_splitted:
                    if value == "1":
                        overall_quality = 1
                    elif value == "0":
                        # As soon as one value is 0 the
                        #   overall_quality is 0. Stop the
                        #   for-loop
                        overall_quality = 0
                        break
        except socket.timeout:
            print("Telnet connection error. Login again.")
            self._telnet_login()

        # Grab file from cognex with ftp command
        filename = 'image.bmp'
        # Generate file in current working dircetory and load
```

```
        #    file
        # w: open for writing
        # b: binary mode
        lf = open(filename, "wb")

        # If ftp connection timed out, establish new connection
        try:
            # ftp-command "RETR": Retrieve a copy of the file
            #   from the server (here: cognex camera)
            # First argument for retrbinary should look like
            #   'RETR filename'
            # Second argument for retrbinary: callback function
            #   is called for each block of data received
            self.ftp.retrbinary("RETR " + filename, lf.write)
        except IOError as e:
            print("FTP connection error. Login again.")
            self._ftp_login()

        # Close the file
        lf.close()

        # Load image to publish it
        image = cv2.imread('image.bmp')

        print("Image loaded, Ready to publish.")

        # Call publishing method
        self._publish_mqtt(image,self.serial_number,overall_qualit
y)

    def disconnect(self) -> None:
        """
        Disconnect from camera (Telnet and FTP) and remove
        the temporary image file.

        Args:
            None

        Returns:
            None
        """
        # Close the connection to ...
        # ... MQTT (and stop loop)
        CamGeneral.disconnect(self)
        # ... FTP
        self.ftp.close()
        # ... Telnet
        self.tn.close()
```

```
        # Remove temporary file due to ftp
        os.remove('image.bmp')
        print("Disconnected from Telnet, FTP and cleaned up.")
```

10.4.2 trigger.py

```
"""
Classes to trigger the system.

Provided classes:
- MQTT-Trigger: triggering over MQTT broker
- ContinuousTrigger: triggering with a fixed provided cycle time
"""

# Import python in-built libraries
import json
import sys
import time

# Import libraries that had been installed with pip install
import paho.mqtt.client as mqtt

class MqttTrigger:
    """
    This class provides an MQTT client instance to receive
    trigger from the MQTT broker. Each instance is automatically
    connected to the set MQTT broker. As soon as a a trigger is
    received, the callback function _on_message is executed.
    The method disconnect() disconnects from MQTT broker.

    Args of constructor:
        cam[Cognex/GenICam]:    A configurated camera ready to
                                get an image. get_image() function
                                must be provided by object
        interface[string]:      Camera interface that is used
        acquisition_delay[float]:
                                Delay between trigger and
                                acquisition
        mqtt_host[string]:      Hostname or IP address of the MQTT

                                broker
        mqtt_port[int]:         Network mqtt_port of the server
                                mqtt_host to connect to
        mqtt_topic[string]:     Topic on MQTT Broker where trigger

                                signal to save an image is send to
```

```
                                (e.g. "test/trigger/")
        only for Cognex cam:
        (opt.) job_id[int]:     ID of the job that should be used
                                on cognex camera, integer from 0
                                to 999, default: None
        (opt.) conventional_image_processing[string]:
                                Choose if conventional image pro-
                                cessing is used and want to get
                                the result, use True or False,
                                default: False

    Returns of constructor:
        A connected instance of MqttTrigger
    """

    def __init__(self,cam,interface,acquisition_delay,mqtt_host,mq
tt_port,mqtt_topic,job_id=None,conventional_image_processing=False
) -> None:
        """
        Connect MQTT client.

        Args:
            see class description

        Returns:
            see class description
        """

        self.cam = cam
        self.interface = interface
        self.acquisition_delay = acquisition_delay
        self.mqtt_host = mqtt_host
        self.mqtt_port = mqtt_port
        self.mqtt_topic = mqtt_topic
        self.job_id = job_id
        self.conventional_image_processing = conventional_image_pr
ocessing



        # Connect to the Broker
        self.client = mqtt.Client()
        self.client.connect(self.mqtt_host, self.mqtt_port)
        # Start the loop to be always able to receive messages
        #   from broker
        self.client.loop_start()
        # Subscribe to the given mqtt_topic
        self.client.subscribe(self.mqtt_topic)
        # Call the _on_message when message is received from
```

```
        #   broker
        self.client.on_message = self._on_message


    # Is called always when a new message is received
    def _on_message(self,client,userdata,msg) -> None:
        """
        Callback function for MQTT on_message.

        Gets a new image with the cam object.

        Args:
            client:         client instance for this callback
            userdata:       private user data as set in Client()
                            or user_data_set()
            message:        an instance of MQTTMessage. This is a
                            class with members topic, payload,
                            qos, retain.

        Returns:
            None
        """
        # If no acquisition delay skip the following
        if self.acquisition_delay > 0.0:
            # Get timestamp of time  when trigger was received.
            #   Measured in ms since epoch. Epoch is defined as
            #   January 1, 1970, 00:00:00 (UTC)
            timestamp_ms = int(round(time.time() * 1000))

        # Deserialize Json
        message = json.loads(msg.payload)
        print("Image acquisition trigger received")

        # If no acquisition delay skip the following
        if self.acquisition_delay > 0.0:
            # Check if timestamp in ms is provided in message.
            #   Use this timestamp instead.
            if 'timestamp_ms' in message:
                timestamp_ms = int(message['timestamp_ms'])
            time_to_get_image = timestamp_ms + round(self.acquisit
ion_delay * 1000)

        # If not cognex used, skip the Cognex specific stuff
        if self.interface != "Cognex":
            # If no acquisition delay skip the following
            if self.acquisition_delay > 0.0:
```

```
                if time_to_get_image < round(time.time() * 1000):
                    sys.exit("Environment Error: ACQUISITION_DELAY
 to short ||| Set acquisition delay is shorter than the processing
 time.")
                while time_to_get_image > round(time.time() * 1000
):# Get an image
                    # Avoid CPU overloading
                    time.sleep(0.1)
            print("Get an image.")
            self.cam.get_image()
            # Stop function at this point
            return

        # Continuing with Cognex specific stuff
        # Check if keyword was in message
        if 'job_id' in message:
            new_job_id = int(message['job_id'])
            # Check if already a job was set
            if self.job_id is not None:
                # Only assign and set job if it is different to
                #   the actually used job
                if self.job_id is not new_job_id:
                    self.cam.set_job(new_job_id)
            else:
                # If there was not set a job yet, assign the
                #   given job
                self.cam.set_job(new_job_id)
            # Used only to update the current job variable in
            #   this class. Setting the job_id happens in
            #   self.cam.set_job(new_job_id)
            self.job_id = new_job_id

        # Check if keyword was in message
        if 'conventional_image_processing' in message:
            conventional_image_processing = message['conventional_
image_processing']

        # Change string into boolean
        if self.conventional_image_processing == 'True':
            self.conventional_image_processing = True
        elif self.conventional_image_processing == 'False':
            self.conventional_image_processing = False


        # If no acquisition delay skip the following
        if self.acquisition_delay > 0.0:
            if time_to_get_image < round(time.time() * 1000):
```

```
                sys.exit("Environment Error: ACQUISITION_DELAY to
short ||| Set acquisition delay is shorter than the processing tim
e.")
            while time_to_get_image > round(time.time() * 1000):#
Get an image
                # Avoid CPU overloading
                time.sleep(0.1)

        # Get an image
        print("Get an image.")
        self.cam.get_image(self.conventional_image_processing)


    def disconnect(self) -> None:
        """
        Disconnects from MQTT broker.

        Args:
            None

        Returns:
            None
        """
        self.client.loop_stop()
        self.client.disconnect()

class ContinuousTrigger:
    """
    This class provides a continuous trigger that acquires images
    with a fixed cycle time. All functionalities are provided
    in the constructor. No methods. Object already stays in a
    while-loop so that the process never jumps out of this
    instance.

    Args of constructor:
        cam[Cognex/GenICam]:    A configurated camera ready to
                                get an image. get_image() function
                                must be provided by object
        interface[string]:      Camera interface that is used
        cycle_time[float]:      Time between each image acqui-
                                sition in seconds
        only for Cognex cam:
        (opt.) conventional_image_processing[string]:
                                Choose if conventional image pro-
                                cessing is used and want to get
                                the result, use True or False,
                                default: False
```

```
    Returns of constructor:
        Continuous triggering instance in which the process will
        stay forever.
    """

    def __init__(self,cam,interface,cycle_time,conventional_image_
processing=False) -> None:
        """
        While-loop with time measuring to have always the same
        cycle time.

        Args:
            see class description

        Returns:
            see class description
        """
        self.cam = cam
        self.interface = interface
        self.cycle_time = cycle_time
        self.conventional_image_processing = conventional_image_pr
ocessing

        # Start the loop to take an image according to cycle time
        while True:
            # Get actual time and save it as start time
            timer_start = time.time()

            # Change string into boolean, only if cognex
            if self.interface == "Cognex":
                if self.conventional_image_processing == 'True':
                    self.conventional_image_processing = True
                elif self.conventional_image_processing == 'False'
:
                    self.conventional_image_processing = False

                # Get an image
                self.cam.get_image(self.conventional_image_process
ing)

            self.cam.get_image()

            # Get actual time and substract the start time from
            #   it to get the time which the current loop needed
            #   to run the code
            loop_time = time.time() - timer_start
            # If the processing time is longer than the cycle
```

```
            #   time throw error
            if loop_time > self.cycle_time:
                sys.exit("Environment Error: CYCLE_TIME to short |
|| Set cycle time is shorter than the processing time for each ima
ge.")
            else:
                # Sleep for difference of cycle time minus loop
                #   time to have a constant cycle time
                delay = self.cycle_time - loop_time
                print("Delay to reach constant cycle time.")
                time.sleep(delay)
```

10.4.3 saving.py

```
"""
Classes to save images

Provided classes:
- LocalSaver
"""

# Import python in-built libraries
import time
import base64
import json
import os

# Import libraries that had been installed with pip install
import paho.mqtt.client as mqtt
import cv2
import numpy as np

class LocalSaver:
    """
    This class is for saving images to a local repository.
    Therefor, every instance of this class is connected to the
    MQTT broker. Every time the broker publishes a new image
    to this instance the _save_image() callback function is
    called to save the image under the set path. The path is
    fixed here because it is for running in a docker container.
    The files are saved in the container and then shared with a
    Docker volume with the host system. To change the volume
    repository on the host system you have to change the path in
    the docker-compose file for the volume.
    It is possible to set a filename. The timestamp of the image
    acquisition is always added to the given filename. Only
    use jpg, png or bmp as file type.
```

```
    Args of constructor:
        file_name[string]:      Enter the name under which the
                                file should be saved in directory
        file_type[string]:      Set file type (possible values:
                                "jpg", "png", "bmp")
        mqtt_host[string]:      Hostname or IP address of the MQTT

                                broker
        mqtt_port[int]:         Network mqtt_port of the server
                                mqtt_host to connect to
        mqtt_topic[string]:     Topic on MQTT Broker where images
                                are sent to
                                (e.g. "test/image/")

    Returns of constructor:
        An instance of LocalSaver to save images from MQTT broker
        in the Docker volume.
    """

    def __init__(self,file_name,file_type,mqtt_host,mqtt_port,mqtt
_topic) -> None:
        """
        Connect to MQTT broker and wait for images to save them.

        Args:
            see class description

        Returns:
            see class description
        """

        self.file_name = file_name
        self.file_type = file_type
        self.host = mqtt_host
        self.port = mqtt_port
        self.topic= mqtt_topic

        # Set static file path
        self.local_data_dir = '/app/ia-
qualitybox/machine_vision/train_images'
        # Connect to the Broker, default port for MQTT 1883
        self.client = mqtt.Client()
        self.client.connect(self.host, self.port)
        # Start the loop to be always able to receive messages
        #   from broker
        self.client.loop_start()
        # Subscribe to the given topic
```

```
        self.client.subscribe(self.topic)
        # Call the _save_image when new message is sent from
        #   broker
        self.client.on_message = self._save_image
        print("Saver active. Waiting for first image")

    # Is called always when a new message is received
    def _save_image(self,client,userdata,msg) -> None:
        """
        Callback function for MQTT on_message.

        The timestamp of the image acquisition is used for unique
        file_names for every image. This guarantees that no file
        is overwriten because of the same file name.
        Depending on the array, the image is saved with one byte
        (monochromatic) or three bytes (colored: BGR/RGB) per
        pixel.

        Args:
            client:         client instance for this callback
            userdata:       private user data as set in Client()
                            or user_data_set()
            message:        an instance of MQTTMessage. This is a
                            class with members topic, payload,
                            qos, retain.

        Returns:
            None
        """
        print("Message received")
        # Deserialize Json
        received_message = json.loads(msg.payload)
        ## Get data out of received message
        # Get timestamp_ms in ms since epoch out of message
        timestamp_ms = received_message["timestamp_ms"]
        # Everything about the image to reconstruct
        image = received_message["image"]
        # Image data
        # Use encode() to convert a string to bytes
        image_bytes = image["image_bytes"].encode()
        # Shape to reconstruct numpy array
        image_height = image["image_height"]
        image_width = image["image_width"]
        image_channels = image["image_channels"]

        # Decode byte array to get numpy array
```

```
        decoded = base64.decodebytes(image_bytes)
        # Shape numpy array
        image_data = np.ndarray(buffer=decoded,
                        dtype=np.uint8,
                        shape=(image_height, image_width, image_ch
annels))

        # Generate complete file path with file name, timestamp_ms
        #   file type
        file_path = os.path.join(
            self.local_data_dir,
            self.file_name + timestamp_ms + "." + self.file_type)

        # Save file under set path
        cv2.imwrite(file_path, image_data)
        print("Image saved under " + file_path)

    def disconnect(self) -> None:
        """
        Stops loop and disconnects from MQTT broker.

        Args:
            None

        Returns:
            None
        """
        self.client.loop_stop()
        self.client.disconnect()
```

10.4.4 inference.py

```
"""
Classes to communicate with TF Serving hosts via REST and gRPC
protocols.

The module provides the RestInferenceClient and
GrpcInferenceClient classes. Both of them are children of
BaseInferenceClient. They define the abstract method
request_method() in order to adapt the functionality to the
required protocol.
"""

# Import python in-built libraries
from abc import ABC, abstractmethod
import base64
import json
import sys
```

```
# Import libraries that had been installed with pip install
import numpy as np
import requests
import cv2
import paho.mqtt.client as mqtt

from tensorflow import make_tensor_proto as make_tensor_proto
from tensorflow_serving.apis import predict_pb2
from tensorflow_serving.apis import prediction_service_pb2_grpc
import grpc

class BaseInferenceClient(ABC):
    """
    Abstract base clase for the different TF Serving protocols.
    This class defines only the basic constructor, the method
    _publish_mqtt() to publish the results to the MQTT broker,
    the method disconnect() and the abstract method
    request_method(). Children must define request_method().

    Args of constructor:
        server_address[string]:     Serving host's address
        server_port[int]:           Serving host's port
        mqtt_host[string]:          Hostname or IP address of the
                                    MQTT broker
        mqtt_port[int]:             Network port of the server
                                    host to connect to
        mqtt_topic_input[string]:   Topic on MQTT Broker where
                                    images for inference are sent
                                    to (e.g. "test/images/")
        mqtt_topic_output[string]:  Topic on MQTT Broker where
                                    results of inference are sent
                                    to (e.g. "test/results/")

    Returns of constructor:
        See inheritors

    """

    def __init__(self,server_address,server_port,mqtt_host,mqtt_po
rt,mqtt_topic_input,mqtt_topic_output) -> None:
        """
        Constructor configures the object with the inference
        serving and MQTT host settings.

        Args:
```

```
            see class description

        Returns:
            see class description
        """

        super().__init__()
        self.server_address = server_address
        self.server_port = server_port
        self.mqtt_host = mqtt_host
        self.mqtt_port = mqtt_port
        self.mqtt_topic_input = mqtt_topic_input
        self.mqtt_topic_output = mqtt_topic_output

        # Connect to the Broker, default port for MQTT 1883
        self.client = mqtt.Client()
        self.client.connect(self.mqtt_host, self.mqtt_port)
        # Start the loop to be always able to receive messages
        #   from broker
        self.client.loop_start()
        # Subscribe to the given input topic
        self.client.subscribe(self.mqtt_topic_input)
        # Call the _request_method when new message is sent
        #   from broker
        self.client.on_message = self._request_method


    @abstractmethod
    def _request_method(self,client, userdata, msg) -> None:
        """
        Must be defined by children.

        Callback function for MQTT on_message.

        Sends an image to inference server to request the label
        and inference results. Afterwards, _publish_mqtt is used
        to send everything back to MQTT broker.

        Args:
            client:         client instance for this callback
            userdata:       private user data as set in Client()
                            or user_data_set()
            message:        an instance of MQTTMessage. This is a
                            class with members topic, payload,
                            qos, retain.

        Returns:
```

```
            None
        """
        pass

    def _publish_mqtt(self,prepared_dictionary) -> None:
        """
        Sends the image, scores and resulting class to the MQTT
        broker. Therefore, image is first converted from a numpy
        array into a byte array and from a byte array into a
        string. Scores is converted into a string array.
        The topic to which it is published on Broker is
        userdefined.

        The MQTT message contains in one json:
            - serial number of the camera
            - timestamp of the acquisition time in ms since epoch
            - image information:
                - string containing the image data (image_bytes)
                - image height, width and channels
            - array of strings containing the classification
                labels
            - array with the probability scores of each label
            - string with the name of the resulting class
            - overall quality (0 for quality not ok and 1 for
                quality ok)

        Dictionary format:
            {
            'serial_number': serial number,
            'timestamp_ms': timestamp_ms,
            'image':
                    {'image_bytes': encoded_image,
                    'image_height': image.shape[0],
                    'image_width': image.shape[1],
                    'image_channels':image.shape[2]},
            'labels': labels,
            'scores': scores,
            'result_class': result_class,
            'overall_quality': overall quality,
            }

        Args:
            prepared_dictionary[dict]:  Already prepared
                                        dictionary which contains
                                        the above information to
                                        send it to the MQTT
```

```
                                        broker

        Returns:
            None

        """
        # Get json formatted string, convert python object
        #   into json object
        message = json.dumps(prepared_dictionary)
        # Publish the message
        self.client.publish(self.mqtt_topic_output, message)

    def disconnect(self) -> None:
        """
        Stops loop and disconnects from MQTT broker.

        Args:
            None

        Returns:
            None
        """
        self.client.loop_stop()
        self.client.disconnect()

class RestInferenceClient(BaseInferenceClient):
    """
    Sends REST requests to a TF Serving host.

    Defines the host to which the requests will be sent and the
    mqtt broker to which the results will be sent.

    Args of constructor:
        server_address[string]:     TF Serving host's address
        server_port[int]:           TF Serving host's port
        mqtt_host[string]:          Hostname or IP address of the
                                    MQTT broker
        mqtt_port[int]:             Network port of the server
                                    host to connect to
        mqtt_topic_input[string]:   Topic on MQTT Broker where
                                    images for inference are sent
                                    to (e.g. "test/images/")
        mqtt_topic_output[string]:  Topic on MQTT Broker where
                                    results of inference are sent
                                    to (e.g. "test/results/")

    Returns:
```

```
        A configured instance of RestInferenceClient.
    """

    def __init__(self,server_address,server_port,mqtt_host,mqtt_po
rt,mqtt_topic_input,mqtt_topic_output) -> None:
        """
        Define the settings for the Rest client and connect to
        MQTT broker based on abstract base class.

        Args:
            see class description

        Returns:
            see class description
        """
        super().__init__(server_address=server_address,
                         server_port=server_port,
                         mqtt_host=mqtt_host,
                         mqtt_port=mqtt_port,
                         mqtt_topic_input=mqtt_topic_input,
                         mqtt_topic_output=mqtt_topic_output)

    def _request_method(self,client,userdata,msg) -> None:
        """
        Callback function for MQTT on_message.

        Sends an image to inference server to request the label
        and inference results. Afterwards, _publish_mqtt is used
        to send everything back to MQTT broker.

        Args:
            client:         client instance for this callback
            userdata:       private user data as set in Client()
                            or user_data_set()
            message:        an instance of MQTTMessage. This is a
                            class with members topic, payload,
                            qos, retain.

        Returns:
            None
        """
        print("Message received")
        # Deserialize Json
        received_message = json.loads(msg.payload)
        ## Get data out of received message
        # Everything about the image to reconstruct
```

```
        image = received_message["image"]
        # Image data
        # Use encode() to convert a string to bytes
        image_bytes = image["image_bytes"].encode()
        # Shape to reconstruct numpy array
        image_height = image["image_height"]
        image_width = image["image_width"]
        image_channels = image["image_channels"]

        # Decode byte array to get numpy array
        decoded = base64.decodebytes(image_bytes)
        # Shape numpy array
        image_data = np.ndarray(buffer=decoded,
                        dtype=np.uint8,
                        shape=(image_height, image_width, image_ch
annels))

        # Encode
        retval, buffer = cv2.imencode('.jpg', image_data)
        encoded_image = base64.b64encode(buffer).decode('utf-8')

        # Prepare request's payload
        instances = {
            'instances': [
                    {'image_bytes': {'b64': str(encoded_image)},
                    'key': 'ilqi_image'}
            ]
        }

        print("Send image to inference.")
        # Send
        url = 'http://{}:{}/v1/models/model:predict'.format(self.s
erver_address, self.server_port)
        response = requests.post(url, data=json.dumps(instances))
        print("Inference succesful.")
        # Parse response
        labels = response.json()['predictions'][0]['labels']
        scores = response.json()['predictions'][0]['scores']
        highest_index = np.argmax(scores)
        result_class = labels[highest_index]

        print("Scores: " + str(scores))
        print("Resulting class according to inference: " + result_
class)

        # TODO: Assessment of the overall quality is missing.
        #   Therefore the information, which labels stand for
        #   good and which for bad, must also be used for REST
```

```
        #   as with gRPC. Or the assessment of the overall
        #   quality has to take place elsewhere. (see gRPC)

        # Add labels, scores and resulting class to retrieved
        #   message dictionary
        received_message['labels'] = str(labels)
        received_message['scores'] = str(scores)
        received_message['result_class'] = result_class

        # Call publish function to send the data to a MQTT broker
        self._publish_mqtt(received_message)


class GrpcInferenceClient(BaseInferenceClient):
    """
    Sends gRPC requests to a TF Serving host.

    Defines the host to which the requests will be sent and the
    mqtt broker to which the results will be sent.
    Additionally the model's name, classification labels, serving
    signature name, input tensor name and input tensor shape must
    be given.

    Args of constructor:
        server_address[string]:         Serving host's address
        server_port[int]:               Serving host's port
        mqtt_host[string]:              Hostname or IP address of
                                        the MQTT broker
        mqtt_port[int]:                 Network port of the server
                                        host to connect to
        mqtt_topic_input[string]:       Topic on MQTT Broker where

                                        images for inference are
                                        sent to
                                        (e.g. "test/images/")
        mqtt_topic_output[string]:      Topic on MQTT Broker where

                                        results of inference are
                                        sent to
                                        (e.g. "test/results/")
        model[string]:                  Name of the model
        labels[string array]:           Array of strings
                                        containing the
                                        classification labels
        quality_labels[string array]:   Array of strings
                                        containing the quality
                                        definition of each label
```

```
        signature_name[string]:         Name of the serving
                                        signature
        input_tensor_name[string]:      Name of the input tensor
        input_tensor_shape[int array]:  Array of ints that define
                                        the input tensor's shape

    Returns:
        A configured instance of GrpcInferenceClient.
    """

    def __init__(self,server_address,server_port,mqtt_host,mqtt_po
rt,mqtt_topic_input,mqtt_topic_output,model,labels,quality_labels,
signature_name,input_tensor_name,input_tensor_shape) -> None:
        """
        Define the settings for the gRPC client and connect to
        MQTT broker based on abstract base class.

        Args:
            see class description

        Returns:
            see class description
        """

        super().__init__(server_address=server_address,
                         server_port=server_port,
                         mqtt_host=mqtt_host,
                         mqtt_port=mqtt_port,
                         mqtt_topic_input=mqtt_topic_input,
                         mqtt_topic_output=mqtt_topic_output)

        self.model = model
        self.labels = labels
        self.quality_labels = quality_labels
        self.signature_name = signature_name
        self.input_tensor_name = input_tensor_name
        self.input_tensor_shape = input_tensor_shape

    def _request_method(self,client,userdata,msg) -> None:
        """
        Callback function for MQTT on_message.

        Sends an image to inference server via gRPC request
        to request the label and inference results. Afterwards,
        _publish_mqtt is used to send everything back to MQTT
        broker.

        Args:
```

```
            client:         client instance for this callback
            userdata:       private user data as set in Client()
                            or user_data_set()
            message:        an instance of MQTTMessage. This is a
                            class with members topic, payload,
                            qos, retain.

        Returns:
            None
        """
        print("Message received")
        # Deserialize Json
        received_message = json.loads(msg.payload)
        ## Get data out of received message
        # Everything about the image to reconstruct
        image = received_message["image"]
        # Image data
        # Use encode() to convert a string to bytes
        image_bytes = image["image_bytes"].encode()
        # Shape to reconstruct numpy array
        image_height = image["image_height"]
        image_width = image["image_width"]
        image_channels = image["image_channels"]

        # Decode byte array to get numpy array
        decoded = base64.decodebytes(image_bytes)
        # Shape numpy array
        image = np.ndarray(buffer=decoded,
                           dtype=np.uint8,
                           shape=(image_height, image_width, image_ch
annels))

        # Preprocess image to fit to input_tensor_shape
        image = cv2.resize(image, (self.input_tensor_shape[1], sel
f.input_tensor_shape[2]))

        # Create a grpc insecure channel
        channel = grpc.insecure_channel('{host}:{port}'.format(hos
t=self.server_address, port=self.server_port))
        stub = prediction_service_pb2_grpc.PredictionServiceStub(c
hannel)

        # Call classification model to make prediction on the
        #   image
        request = predict_pb2.PredictRequest()
        request.model_spec.name = self.model
        request.model_spec.signature_name = self.signature_name
```

```
        request.inputs[self.input_tensor_name].CopyFrom(make_tenso
r_proto(image, dtype=np.float32, shape=self.input_tensor_shape))

        # Using 10.0 to block the thread until the response
        #   arrives. Use stub.Predict(request, 5.0) for
        #   asynchronous calls
        prediction = stub.Predict(request, 10.0)

        # Parse and print the response
        scores = prediction.outputs['outputs'].float_val
        highest_index = np.argmax(scores)
        result_class = self.labels[highest_index]
        if self.quality_labels[highest_index] == 'Good':
            overall_quality = 1
        elif self.quality_labels[highest_index] == 'Bad':
            overall_quality = 0
        else:
            sys.exit("Environment Error: One ore more of QUALITY_L
ABEL_1-
5 is not allowed ||| Make sure to only use the words 'Bad' and 'Go
od' without quotation marks.")

        print("Scores: " + str(scores))
        print("Resulting class according to inference: " + result_
class)
        print("Overall quality: " + str(overall_quality))

        # Add labels, scores, resulting class and overall_quality
        #   to retrieved message dictionary
        received_message['labels'] = str(labels)
        received_message['scores'] = str(scores)
        received_message['result_class'] = result_class
        received_message['overall_quality'] = str(overall_quality)

        # Call publish function to send the data to a MQTT broker
        self._publish_mqtt(received_message)
```

10.5 Symbole Prozessflussdiagramme

Für die Prozessflussdiagramme wurden die nachfolgend beschriebenen und in Abb. 10.5 dargestellten Symbole verwendet: [Myl98; ISO5807]

- Flusslinie (engl. Flowline): Zeigt den Prozessblauf, in dem sie von einem Symbol ausgeht und auf ein anderes zeigt.
- Start/Ende (engl. Terminator): Symbolisiert den Beginn oder das Ende eines Programms oder Teilprozesses

- Tätigkeit (engl. Process): Repräsentiert einen Satz von Funktionen, die Daten ändern, umformen, speichern usw.
- Entscheidung (engl. Decision): Steht für eine Fallunterscheidung, um zwischen verschiedenen Pfaden je nach Bedingung zu wählen.
- Unterprogramm (engl. Predefined Process): Stellt einen Prozess dar, der an einer anderen Stelle außerhalb des aktuellen Prozesses ausdefiniert ist.
- Status (nicht ISO konform): Zeigt den aktuellen Status eines Objekts.

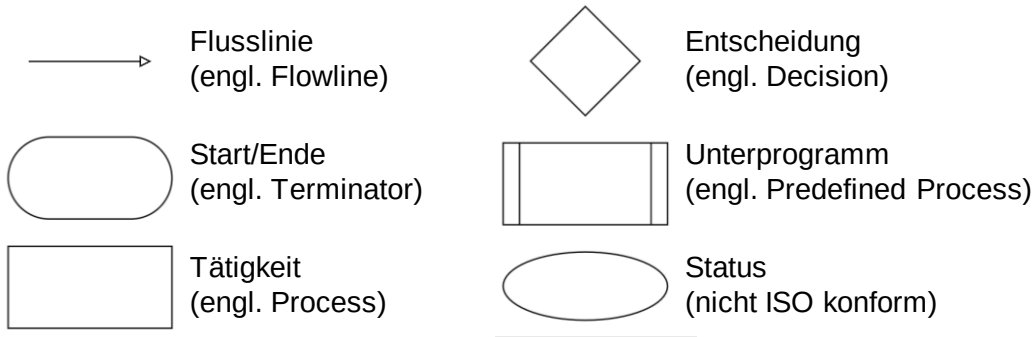


Abb. 10.5: Verwendete Symbole in den Flussdiagrammen in Anlehnung an ISO 5807 (i. A. a. [ISO5807])

10.6 Microservice: image_acquisition_cognex

10.6.1 Prozessflussdiagramm

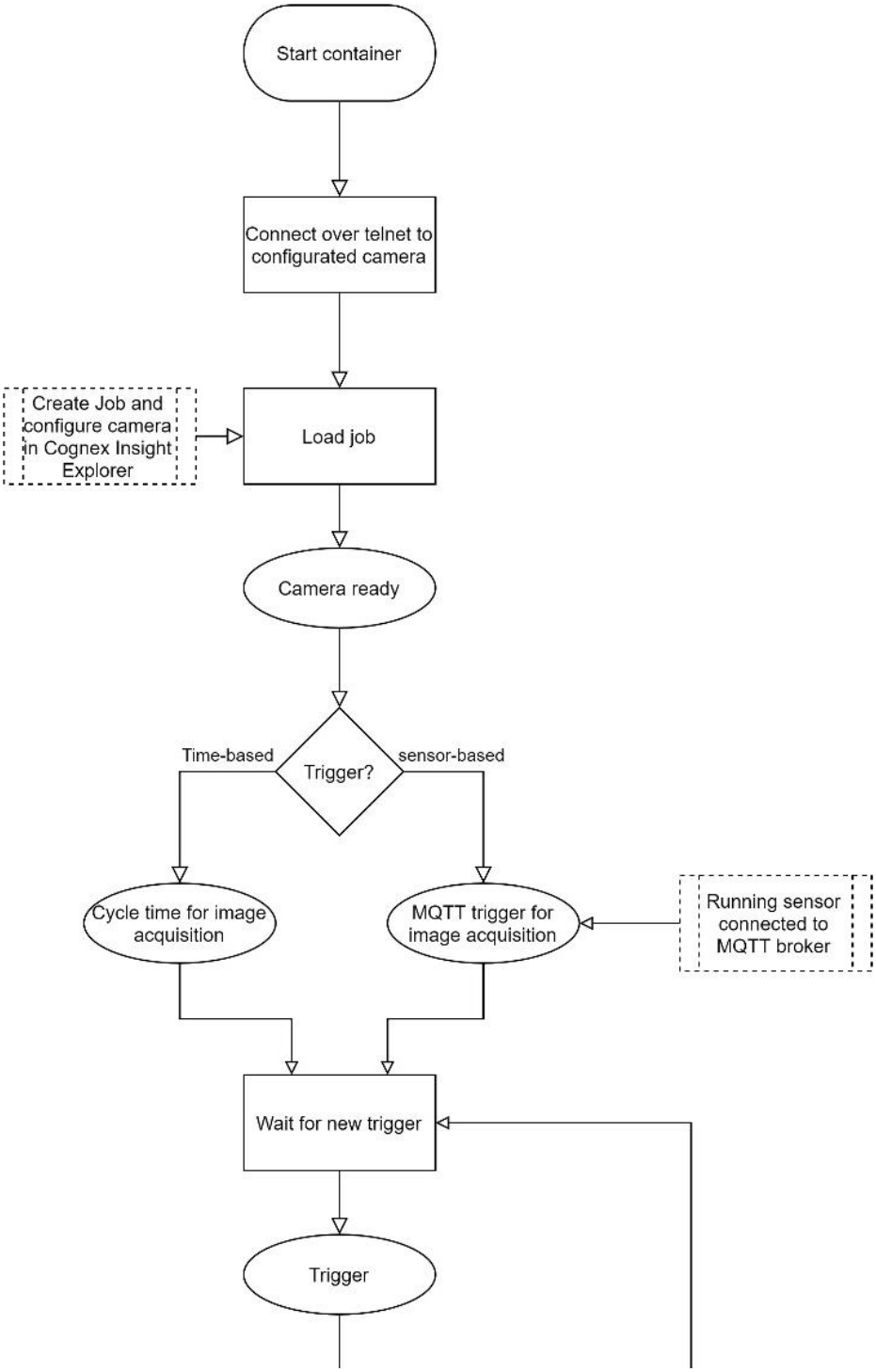


Abb. 10.6: Prozessflussdiagramm Microservice image_acquisition_cognex (Teil 1 von 2)

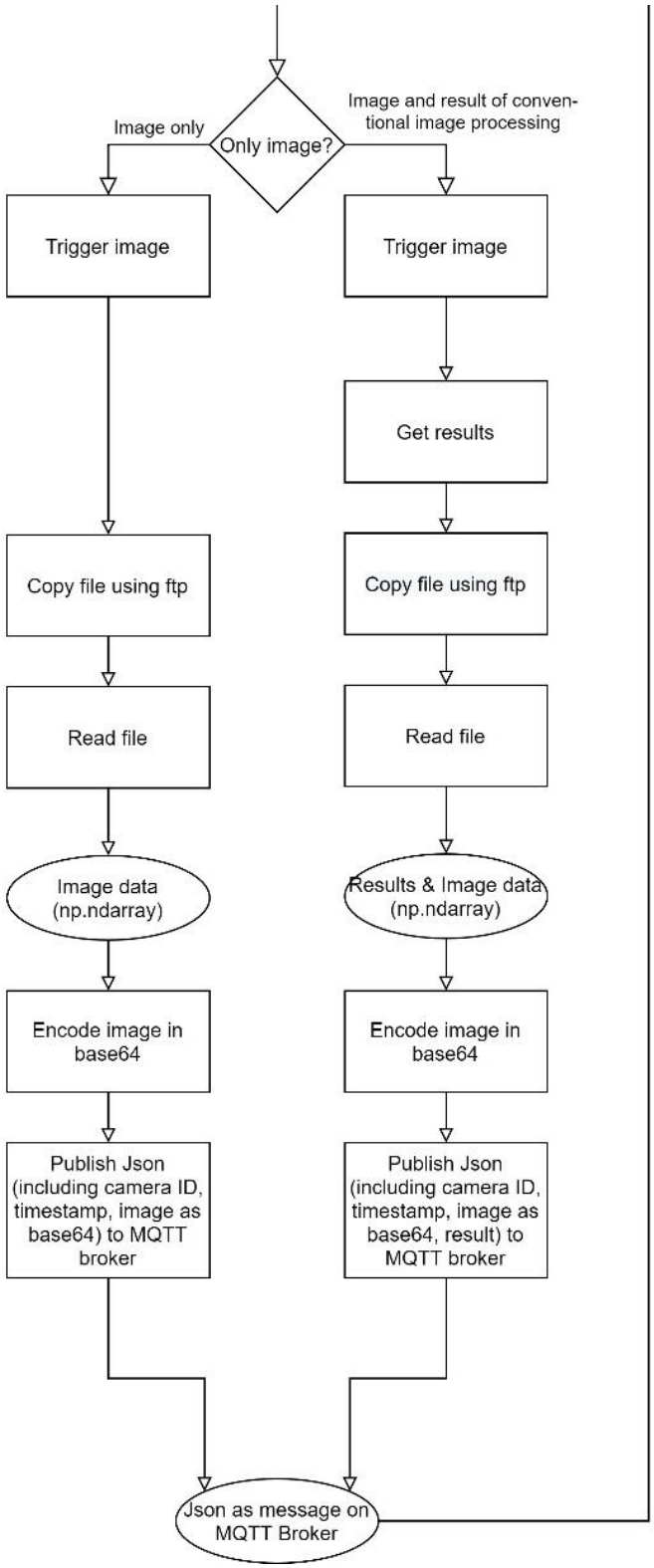


Abb. 10.7: Prozessflussdiagramm Microservice image_acquisition_cognex (Teil 2 von 2)

10.6.2 main.py

```
"""
Main script that is executed insight the docker container.
It imports the required libraries and self-written modules
as well as the environment variables. The defintion of the
environment variables can be found in the env-file.

Depending on the settings of the environment variables,
objects of classes are intantiated which provides all
necessary connections and functionalities. To find out
more about the classes take a look into the imported
modules. If there is not a while-loop already provided by
the instance, a while-loop is used to keep everything
running until the user stops the container.
"""

# Import python in-built libraries
import time
import os
import sys

# Import self-written modules
from cameras import Cognex
from trigger import MqttTrigger,ContinuousTrigger

### LOAD OVERALL ENVIROMENT SETTINGS
## MQTT SETTINGS
MQTT_HOST = os.environ.get('MQTT_HOST')
MQTT_PORT = int(os.environ.get('MQTT_PORT', 1883))
MQTT_TOPIC_TRIGGER = os.environ.get('MQTT_TOPIC_TRIGGER', 'quality
Inspection/trigger')
MQTT_TOPIC_IMAGE = os.environ.get('MQTT_TOPIC_IMAGE', 'qualityInsp
ection/image')

## TRIGGER & PROCESS SETTINGS
TRIGGER = os.environ.get('TRIGGER')
ACQUISITION_DELAY = float(os.environ.get('ACQUISITION_DELAY', 2.0)
)
CYCLE_TIME = float(os.environ.get('CYCLE_TIME', 10.0))

## CAMERA SETTINGS
CAMERA_INTERFACE = os.environ.get('CAMERA_INTERFACE')
SERIAL_NUMBER = os.environ.get('SERIAL_NUMBER','')

# Cognex settings
IP_ADDRESS_COGNEX = os.environ.get('IP_ADDRESS_COGNEX')
JOB_ID = os.environ.get('JOB_ID', 'default')
```

```
if JOB_ID == 'default':
    JOB_ID = None
else:
    JOB_ID = int(JOB_ID)
CONVENTIONAL_IMAGE_PROCESSING = os.environ.get('CONVENTIONAL_IMAGE
_PROCESSING', 'False')

### End of loading settings ###


if __name__ == "__main__":
    # Check selected camera interface
    if CAMERA_INTERFACE == "GenICam":
        # Stop system, not possible to run with this setting
        sys.exit("Environment Error: INTERFACE not allowed ||| You
 cannot use this docker container with GenICam. Use the sepcific o
ne for GenICam cameras.")
    elif CAMERA_INTERFACE == "Cognex":
        # Get cam object, pass required arguments
        cam = Cognex(MQTT_HOST,MQTT_PORT,MQTT_TOPIC_IMAGE,SERIAL_N
UMBER,IP_ADDRESS_COGNEX,JOB_ID)
    else:
        # Stop system, not possible to run with this setting
        sys.exit("Environment Error: INTERFACE not supported ||| M
ake sure to set a value that is allowed according to the specified
 possible values for this environment variable and make sure the s
pelling is correct.")

    # Check trigger type and use appropriate instance of the
    #   trigger classes
    if TRIGGER == "Continuous":
        # Never jumps out of the processes of the instance
        ContinuousTrigger(cam,CAMERA_INTERFACE,CYCLE_TIME,CONVENTI
ONAL_IMAGE_PROCESSING)
    elif TRIGGER == "MQTT":
        # Starts an asynchroneous process for working with
        #   the received mqtt data
        trigger = MqttTrigger(cam,CAMERA_INTERFACE,ACQUISITION_DEL
AY,MQTT_HOST,MQTT_PORT,MQTT_TOPIC_TRIGGER,JOB_ID,CONVENTIONAL_IMAG
E_PROCESSING)

        # Run forever to stay connected
        while True:
            # Avoid overloading the CPU
            time.sleep(10)
            print("Still running.")
    else:
        # Stop system, not possible to run with this setting
```

```
        sys.exit("Environment Error: TRIGGER not supported ||| Mak
e sure to set a value that is allowed according to the specified p
ossible values for this environment variable and make sure the spe
lling is correct.")
```

10.6.3 requirements.txt

opencv-python==4.2.0.34

harvesters==1.2.6

paho-mqtt==1.5.0

10.6.4 Dockerfile

```
# Try to keep the number docker layers low
# Only the RUN ADD and COPY commands generate layers
# Build up from python:3.7 image
FROM python:3.7

# Fix possible dependency problems and install dependencies,
#   first layer because it changes less often. Requiered for
#   openCV
RUN apt-get update -y && \
    apt-get install -y --no-install-recommends \
    libglib2.0-dev \
    libsm6 \
    libxext6 \
    libxrender1

# Install python libraries
COPY ./src/requirements.txt /app/ia-
qualitybox/image_acquisition_cognex/
RUN pip install -r /app/ia-
qualitybox/image_acquisition_cognex/requirements.txt

# Copy the current directory contents into the container at /app
COPY ./src/cameras.py \
     ./src/trigger.py \
     ./src/main.py \
     /app/ia-qualitybox/image_acquisition_cognex/

# Run when the container launches
CMD [ "python", "-u", "/app/ia-
qualitybox/image_acquisition_cognex/main.py" ]
```

10.7 Microservice: image_acquisition_genicam

10.7.1 Prozessflussdiagramm

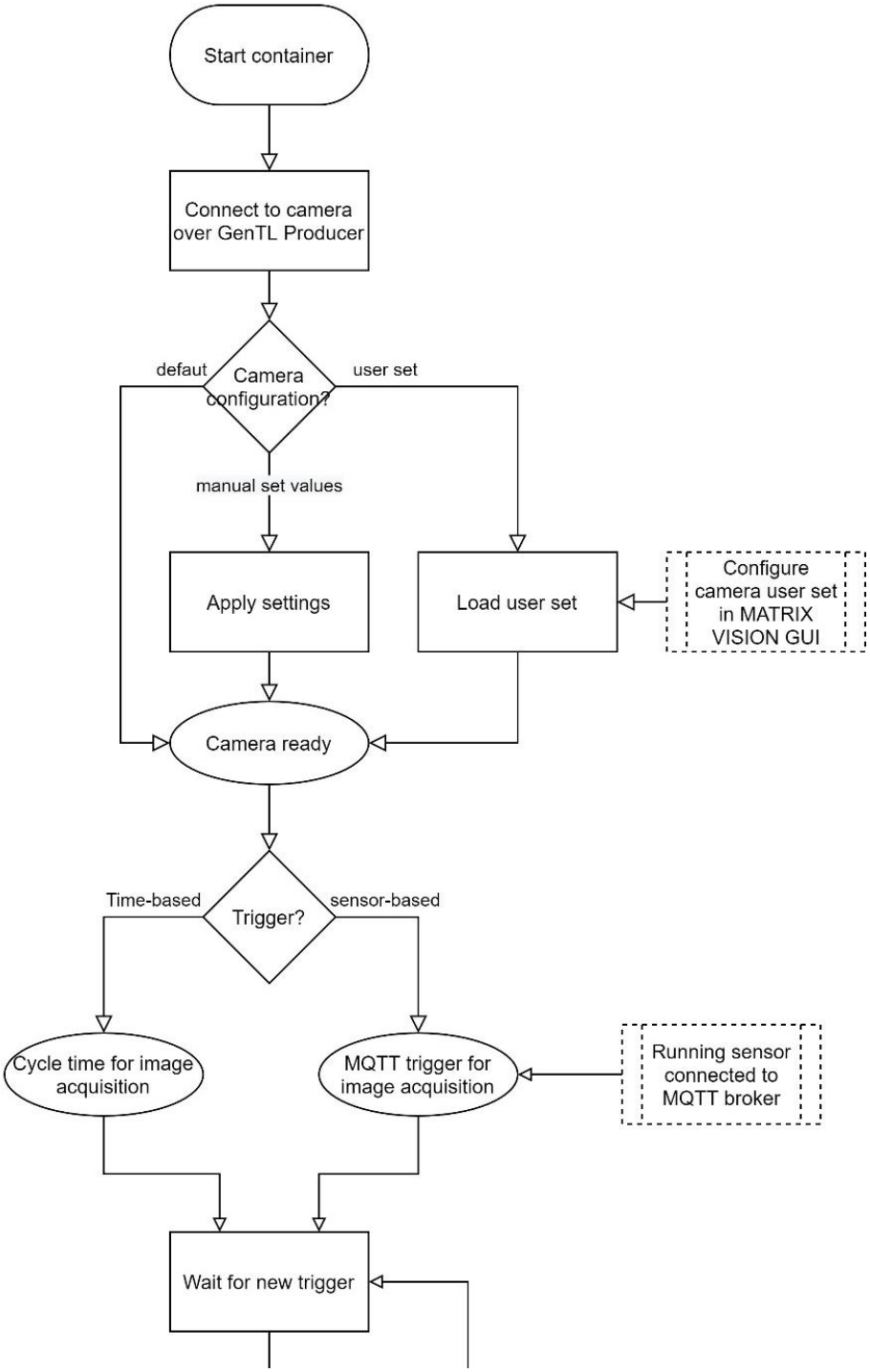


Abb. 10.8: Prozessflussdiagramm Microservice image_acquisition_genicam (Teil 1 von 2)

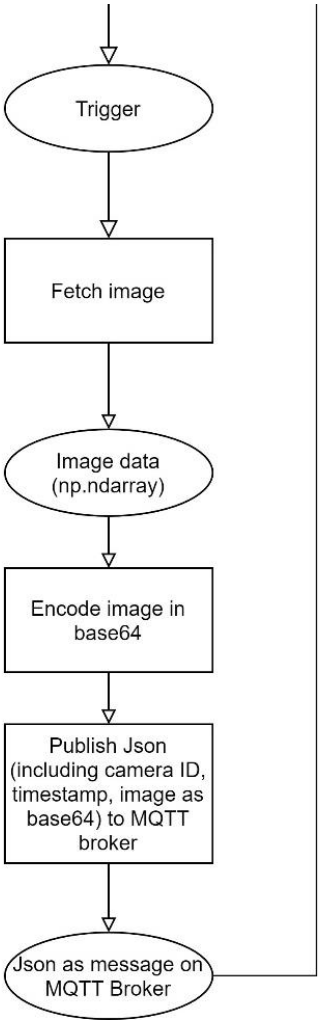


Abb. 10.9: Prozessflussdiagramm Microservice image_acquisition_genicam (Teil 2 von 2)

10.7.2 main.py

```
"""
Main script that is executed insight the docker container.
It imports the required libraries and self-written modules
as well as the environment variables. The defintion of the
environment variables can be found in the env-file.

Depending on the settings of the environment variables,
objects of classes are intantiated which provides all
necessary connections and functionalities. To find out
more about the classes take a look into the imported
modules. If there is not a while-loop already provided by
the instance, a while-loop is used to keep everything
running until the user stops the container.
"""
```

```
# Import python in-built libraries
import time
import os
import sys

# Import self-written modules
from cameras import GenICam
from trigger import MqttTrigger,ContinuousTrigger

### LOAD OVERALL SETTINGS
## MQTT SETTINGS
MQTT_HOST = os.environ.get('MQTT_HOST')
MQTT_PORT = int(os.environ.get('MQTT_PORT', 1883))
MQTT_TOPIC_TRIGGER = os.environ.get('MQTT_TOPIC_TRIGGER', 'quality
Inspection/trigger')
MQTT_TOPIC_IMAGE = os.environ.get('MQTT_TOPIC_IMAGE', 'qualityInsp
ection/image')

## TRIGGER & PROCESS SETTINGS
TRIGGER = os.environ.get('TRIGGER')
ACQUISITION_DELAY = float(os.environ.get('ACQUISITION_DELAY', 2.0)
)
CYCLE_TIME = float(os.environ.get('CYCLE_TIME', 10.0))

## CAMERA SETTINGS
CAMERA_INTERFACE = os.environ.get('CAMERA_INTERFACE')
SERIAL_NUMBER = os.environ.get('SERIAL_NUMBER','')

# GenICam settings
DEFAULT_GENTL_PRODUCER_PATH = os.environ.get('DEFAULT_GENTL_PRODUC
ER_PATH', '/opt/mvIMPACT_Acquire/lib/x86_64/mvGenTLProducer.cti')
USER_SET_SELECTOR = os.environ.get('USER_SET_SELECTOR', 'Default')
IMAGE_WIDTH = int(os.environ.get('IMAGE_WIDTH', 800))
IMAGE_HEIGHT = int(os.environ.get('IMAGE_HEIGHT', 800))
PIXEL_FORMAT = os.environ.get('PIXEL_FORMAT', 'Mono8')
IMAGE_CHANNELS = os.environ.get('IMAGE_CHANNELS', 'None')
EXPOSURE_TIME = os.environ.get('EXPOSURE_TIME', 'None')
EXPORSURE_AUTO = os.environ.get('EXPORSURE_AUTO', 'Off')
GAIN_AUTO = os.environ.get('GAIN_AUTO', 'Off')
BALANCE_WHITE_AUTO = os.environ.get('BALANCE_WHITE_AUTO', 'Off')

if IMAGE_CHANNELS != 'None':
    IMAGE_CHANNELS = int(IMAGE_CHANNELS)
if EXPOSURE_TIME != 'None':
    EXPOSURE_TIME = float(EXPOSURE_TIME)

### End of loading settings ###
```

```
if __name__ == "__main__":
    # Check selected camera interface
    if CAMERA_INTERFACE == "Cognex":
        # Stop system, not possible to run with this setting
        sys.exit("Environment Error: INTERFACE not allowed ||| You
 cannot use this docker container with Cognex. Use the sepcific on
e for Cognex cameras.")
    elif CAMERA_INTERFACE == "GenICam":
        cam = GenICam(MQTT_HOST,MQTT_PORT,MQTT_TOPIC_IMAGE,SERIAL_
NUMBER,DEFAULT_GENTL_PRODUCER_PATH,image_width=IMAGE_WIDTH,image_h
eight=IMAGE_HEIGHT,pixel_format=PIXEL_FORMAT)
    else:
        # Stop system, not possible to run with this setting
        sys.exit("Environment Error: INTERFACE not supported ||| M
ake sure to set a value that is allowed according to the specified
 possible values for this environment variable and make sure the s
pelling is correct.")

    # Check trigger type and use appropriate instance of the
    #   trigger classes
    if TRIGGER == "Continuous":
        # Never jumps out of the processes of the instance
        ContinuousTrigger(cam,CAMERA_INTERFACE,CYCLE_TIME)
    elif TRIGGER == "MQTT":
        # Starts an asynchroneous process for working with
        #   the received mqtt data
        trigger = MqttTrigger(cam,CAMERA_INTERFACE,ACQUISITION_DEL
AY,MQTT_HOST,MQTT_PORT,MQTT_TOPIC_TRIGGER)

        # Run forever to stay connected
        while True:
            # Avoid overloading the CPU
            time.sleep(10)
            print("Still running.")
    else:
        # Stop system, not possible to run with this setting
        sys.exit("Environment Error: TRIGGER not supported ||| Mak
e sure to set a value that is allowed according to the specified p
ossible values for this environment variable and make sure the spe
lling is correct.")
```

10.7.3 requirements.txt

opencv-python==4.2.0.34

harvesters==1.2.6

paho-mqtt==1.5.0

10.7.4 Dockerfile

```
# Try to keep the number docker layers low
# Only the RUN ADD and COPY commands generate layers
# Build up from python:3.7 image
FROM python:3.7

# Fix possible dependency problems and install dependencies,
#   first layer because it changes less often. Requiered for
#   openCV and MATRIX Vision Impact Acquire SDK
RUN apt-get update -y && \
    apt-get install -y --no-install-recommends \
    dialog apt-utils \
    apt-utils \
    build-essential \
    cmake \
    git \
    curl \
    ca-certificates \
    zlib1g-dev \
    libgtk2.0-0 \
    libglib2.0-dev \
    libsm6 \
    libxext6 \
    libxrender1 \
    libxrender-dev \
    libpng-dev &&\
    rm -rf /var/lib/apt/lists/*

# Install genTL producer
WORKDIR /app/ia-qualitybox/gen_tl_producer/
ADD http://static.matrix-
vision.com/mvIMPACT_Acquire/2.37.1/install_mvGenTL_Acquire.sh /app
/ia-qualitybox/gen_tl_producer/
ADD http://static.matrix-
vision.com/mvIMPACT_Acquire/2.37.1/mvGenTL_Acquire-x86_64_ABI2-
2.37.1.tgz /app/ia-qualitybox/gen_tl_producer/
RUN chmod ugo+x install_mvGenTL_Acquire.sh
RUN ./install_mvGenTL_Acquire.sh

# Install python libraries
COPY ./src/requirements.txt /app/ia-
qualitybox/image_acquisition_genicam/
RUN pip install -r /app/ia-
qualitybox/image_acquisition_genicam/requirements.txt

# Copy the current directory contents into the container at /app
COPY ./src/cameras.py \
```

```
    ./src/trigger.py \
    ./src/main.py \
    /app/ia-qualitybox/image_acquisition_genicam/

# Run when the container launches
CMD [ "python", "-u", "/app/ia-
qualitybox/image_acquisition_genicam/main.py" ]
```

10.8 Microservice: local_saving

10.8.1 Prozessflussdiagramm

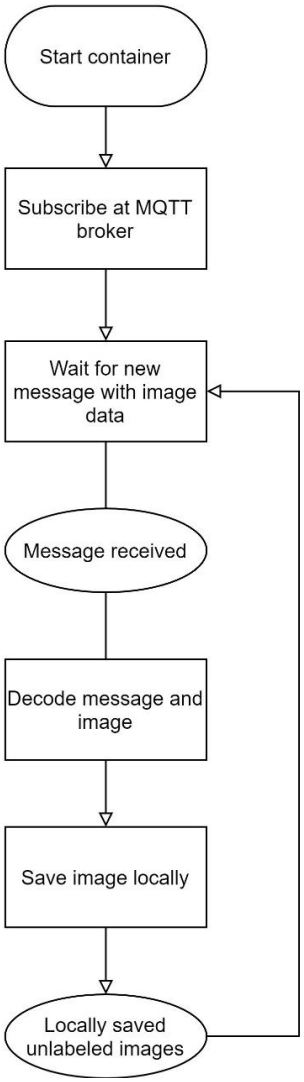


Abb. 10.10: Prozessflussdiagramm Microservice local_saving

10.8.2 main.py

```
"""
Main script that is executed insight the docker container.
It imports the required libraries and self-written modules
as well as the environment variables. The defintion of the
environment variables can be found in the env-file.

Depending on the settings of the environment variables,
objects of classes are intantiated which provides all
necessary connections and functionalities. To find out
more about the classes take a look into the imported
modules. A while-loop is used to keep everythin running
until the user stops the container.
"""

# Import python in-built libraries
import time
import os

# Import self-written modules
from saving import LocalSaver

### LOAD OVERALL ENVIROMENT SETTINGS
## MQTT SETTINGS
MQTT_HOST = os.environ.get('MQTT_HOST')
MQTT_PORT = int(os.environ.get('MQTT_PORT', 1883))
MQTT_TOPIC_IMAGE = os.environ.get('MQTT_TOPIC_IMAGE', 'qualityInsp
ection/image')

## CAPTURE TRAINING IMAGES (Variables that are shared
#   throughout the capture training image package):
FILE_NAME = os.environ.get('FILE_NAME', '')
FILE_TYPE = os.environ.get('FILE_TYPE', 'png')

### End of loading settings ###

if __name__ == "__main__":
    # Starts an asynchroneous process for working with the
    #   received mqtt data
    imageSaver = LocalSaver(FILE_NAME,FILE_TYPE,MQTT_HOST,MQTT_POR
T,MQTT_TOPIC_IMAGE)

    # Run forever to stay connected
    while True:
        # Avoid overloading the CPU
        time.sleep(10)
```

```
        print("Still running.")
```

10.8.3 requirements.txt

opencv-python==4.2.0.34

paho-mqtt==1.5.0

numpy==1.18.4

10.8.4 Dockerfile

```
# Try to keep the number docker layers low
# Only the RUN ADD and COPY commands generate layers
# Build up from python:3.7 image
FROM python:3.7

# Fix possible dependency problems and install dependencies,
#   first layer because it changes less often. Requiered for
#   openCV
RUN apt-get update -y && \
    apt-get install -y --no-install-recommends \
    libglib2.0-dev \
    libsm6 \
    libxext6 \
    libxrender1

# Install python libraries
COPY ./src/requirements.txt /app/ia-qualitybox/local_saving/
RUN pip install -r /app/ia-
qualitybox/local_saving/requirements.txt

# Copy the current directory contents into the container at /app
COPY ./src/saving.py \
     ./src/main.py \
     /app/ia-qualitybox/local_saving/

# Run when the container launches
CMD [ "python", "-u", "/app/ia-qualitybox/local_saving/main.py" ]
```

10.9 Microservice: inference_client

10.9.1 Prozessflussdiagramm

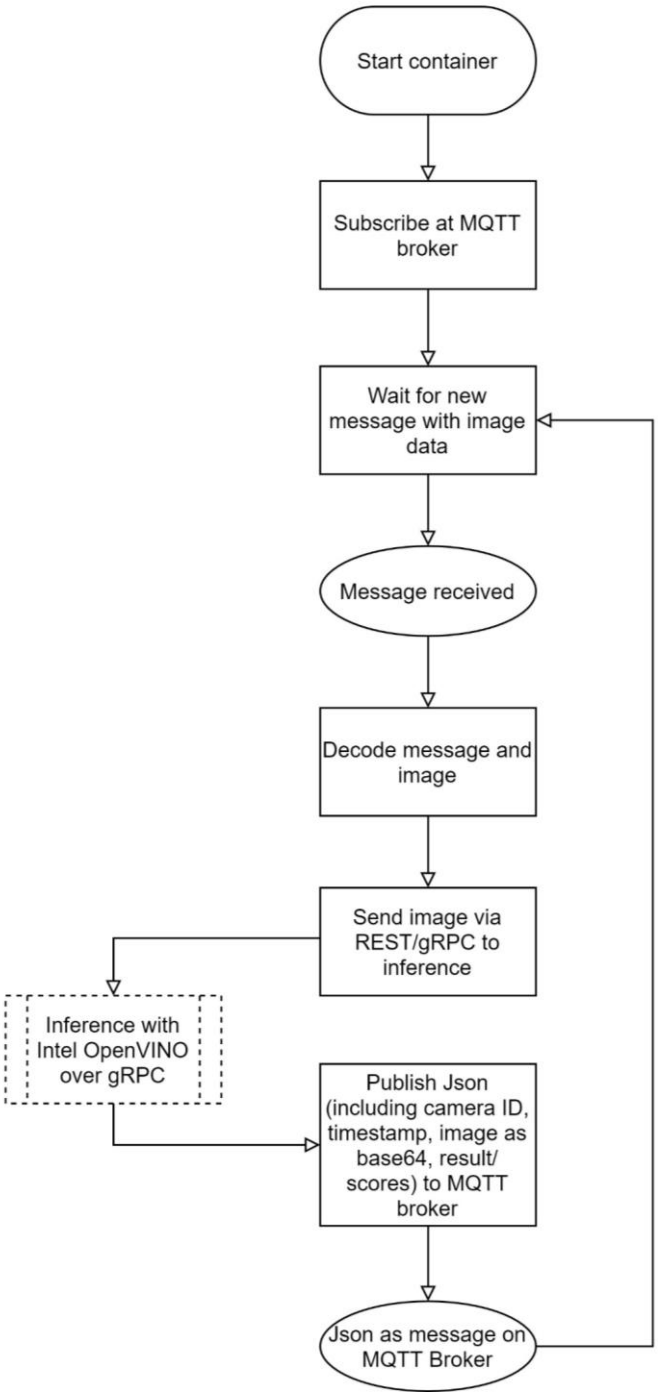


Abb. 10.11: Prozessflussdiagramm Microservice inference_client

10.9.2 main.py

```
"""
Main script that is executed insight the docker container.
It imports the required libraries and self-written modules
as well as the environment variables. The defintion of the
environment variables can be found in the env-file.

Depending on the settings of the environment variables,
objects of classes are intantiated which provides all
necessary connections and functionalities. To find out
more about the classes take a look into the imported
modules. A while-loop is used to keep everythin running
until the user stops the container.
"""

# Import python in-built libraries
import os
import logging
import time
import sys

# Import self-written modules
from inference import RestInferenceClient,GrpcInferenceClient

### LOAD OVERALL ENVIRONMENT SETTINGS
## MQTT SETTINGS
MQTT_HOST = os.environ.get('MQTT_HOST')
MQTT_PORT = int(os.environ.get('MQTT_PORT', 1883))
MQTT_TOPIC_IMAGE = os.environ.get('MQTT_TOPIC_IMAGE', 'qualityInsp
ection/image')
MQTT_TOPIC_RESULTS = os.environ.get('MQTT_TOPIC_RESULTS', 'quality
Inspection/results')

## INFERENCE (Variables that are shared throughout the
#   inference package):
SERVING_INTERFACE = os.environ.get('SERVING_INTERFACE')
SERVING_HOST = os.environ.get('SERVING_HOST')
SERVING_PORT = int(os.environ.get('SERVING_PORT'))

# gRPC specfic variables
MODEL = os.environ.get('MODEL', 'model')
LABEL_1 = os.environ.get('LABEL_1','')
LABEL_2 = os.environ.get('LABEL_2','')
LABEL_3 = os.environ.get('LABEL_3','')
LABEL_4 = os.environ.get('LABEL_4','')
LABEL_5 = os.environ.get('LABEL_5','')
QUALITY_LABEL_1 = os.environ.get('QUALITY_LABEL_1','')
```

```
QUALITY_LABEL_2 = os.environ.get('QUALITY_LABEL_2','')
QUALITY_LABEL_3 = os.environ.get('QUALITY_LABEL_3','')
QUALITY_LABEL_4 = os.environ.get('QUALITY_LABEL_4','')
QUALITY_LABEL_5 = os.environ.get('QUALITY_LABEL_5','')
SIGNATURE_NAME = os.environ.get('SIGNATURE_NAME', 'serving_default
')
INPUT_TENSOR_NAME = os.environ.get('INPUT_TENSOR_NAME', 'inputs')
INPUT_TENSOR_SHAPE_1ST_DIMENSION = int(os.environ.get('INPUT_TENSO
R_SHAPE_1ST_DIMENSION'))
INPUT_TENSOR_SHAPE_2ND_DIMENSION = int(os.environ.get('INPUT_TENSO
R_SHAPE_2ND_DIMENSION'))
INPUT_TENSOR_SHAPE_3RD_DIMENSION = int(os.environ.get('INPUT_TENSO
R_SHAPE_3RD_DIMENSION'))
INPUT_TENSOR_SHAPE_4TH_DIMENSION = int(os.environ.get('INPUT_TENSO
R_SHAPE_4TH_DIMENSION'))

# Get one array which contains all labels
labels_all = [LABEL_1,LABEL_2,LABEL_3,LABEL_4,LABEL_5]
# Use only none empty labels
labels = []
for i in labels_all:
    if i != '':
        labels.append(i)
# Get one array which contains all quality statements for labels
quality_labels_all = [QUALITY_LABEL_1,QUALITY_LABEL_2,QUALITY_LABE
L_3,QUALITY_LABEL_4,QUALITY_LABEL_5]
# Use only none empty quality statements
quality_labels = []
for i in quality_labels_all:
    if i != '':
        quality_labels.append(i)

# Check user settings
if len(labels) != len(quality_labels):
    sys.exit("Environment Error: QUALITY_LABEL_1-
5 do not fit to LABEL_1-
5 ||| Make sure to define the quality for each label that you are
using.")

# Get one array for input tensor shape
input_tensor_shape = [INPUT_TENSOR_SHAPE_1ST_DIMENSION,INPUT_TENSO
R_SHAPE_2ND_DIMENSION,INPUT_TENSOR_SHAPE_3RD_DIMENSION,INPUT_TENSO
R_SHAPE_4TH_DIMENSION]

### End of loading settings ###

if __name__ == '__main__':
    # Setup serving client according to the needed interface
```

```
    # Starts an asynchroneous process for working with mqtt
    if SERVING_INTERFACE == 'REST':
        serving_client = RestInferenceClient(SERVING_HOST,
                                    SERVING_PORT,
                                    MQTT_HOST,
                                    MQTT_PORT,
                                    MQTT_TOPIC_IMAGE,
                                    MQTT_TOPIC_RESULTS)

    elif SERVING_INTERFACE == 'gRPC':
        serving_client = GrpcInferenceClient(SERVING_HOST,
                                    SERVING_PORT,
                                    MQTT_HOST,
                                    MQTT_PORT,
                                    MQTT_TOPIC_IMAGE,
                                    MQTT_TOPIC_RESULTS,
                                    MODEL,
                                    labels,
                                    quality_labels,
                                    SIGNATURE_NAME,
                                    INPUT_TENSOR_NAME,
                                    input_tensor_shape)

    else:
        # Stop system, not possible to run with this setting
        sys.exit("Enviroment Error: SERVING_INTERFACE not supporte
d ||| Make sure to set a value that is allowed according to the sp
ecified possible values for this environment variable and make sur
e the spelling is correct.")

    # Run forever to stay connected
    while True:
        # Avoid overloading the CPU
        time.sleep(10)
        print("Still running.")
```

10.9.3 requirements.txt

opencv-python==4.2.0.34

paho-mqtt==1.5.0

numpy==1.18.4

requests==2.21.0

grpcio==1.28.1

tensorflow-serving-api==1.13.0

10.9.4 Dockerfile

```
# Try to keep the number docker layers low
# Only the RUN ADD and COPY commands generate layers
# Build up from python:3.7 image
FROM python:3.7

# Fix possible dependency problems and install dependencies,
#   first layer because it changes less often.Requiered for openCV
RUN apt-get update -y && \
    apt-get install -y --no-install-recommends \
    libglib2.0-dev \
    libsm6 \
    libxext6 \
    libxrender1

# Install python libraries
COPY ./src/requirements.txt /app/ia-qualitybox/inference_client/
RUN pip install -r /app/ia-
qualitybox/inference_client/requirements.txt

# Copy the current directory contents into the container at /app
COPY ./src/inference.py \
     ./src/main.py \
     /app/ia-qualitybox/inference_client/

# Run when the container launches
CMD [ "python", "-u", "/app/ia-
qualitybox/inference_client/main.py" ]
```

10.10 Cognex Kamera

10.10.1 Verzeichnisstruktur

```
cognex
|   docker-compose.yml
|   qualityInsp.env
|
+---image_acquisition_cognex
|   |   Dockerfile
|   |
|   \---src
|           cameras.py
|           main.py
|           requirements.txt
```

```
|           trigger.py
|
+---inference_client
|   |   Dockerfile
|   |
|   \---src
|           inference.py
|           main.py
|           requirements.txt
|
\---local_saving
    |   Dockerfile
    |
    \---src
            main.py
            requirements.txt
            saving.py
```

10.10.2 Cognex: docker-compose.yml

```
version: '3'

services:
  image_acquisition_cognex:
    container_name: microservice_image_acquisition_cognex
    network_mode: host #important to connect to camera, otherwise
in network of container
    build:
      context: ./image_acquisition_cognex/
      dockerfile: ./Dockerfile
    env_file:
      - ./qualityInsp.env
    restart: unless-stopped

  local_saving:
    container_name: microservice_local_saving
    network_mode: host
    build:
      context: ./local_saving/
      dockerfile: ./Dockerfile
    env_file:
      - ./qualityInsp.env
    volumes: # Change path here for other directory for saving
      - ./images/Train:/app/ia-
qualitybox/machine_vision/train_images
    restart: unless-stopped

  inference_client:
```

```
    container_name: microservice_inference_client
    network_mode: host
    build:
      context: ./inference_client/
      dockerfile: ./Dockerfile
    env_file:
      - ./qualityInsp.env
    restart: unless-stopped
```

10.11 GenICam Kamera

10.11.1 Verzeichnisstruktur

```
genicam
|   docker-compose.yml
|   qualityInsp.env
|
+---image_acquisition_genicam
|   |   Dockerfile
|   |
|   \---src
|           cameras.py
|           main.py
|           requirements.txt
|           trigger.py
|
+---inference_client
|   |   Dockerfile
|   |
|   \---src
|           inference.py
|           main.py
|           requirements.txt
|
\---local_saving
    |   Dockerfile
    |
    \---src
            main.py
            requirements.txt
            saving.py
```

10.11.2 GenICam: docker-compose.yml

```
version: '3'

services:
  image_acquisition_genicam:
    container_name: microservice_image_acquisition_genicam
    network_mode: host #important to connect to camera, otherwise
in network of container
    build:
      context: ./image_acquisition_genicam/
      dockerfile: ./Dockerfile
    env_file:
      - ./qualityInsp.env
    restart: unless-stopped

  local_saving:
    container_name: microservice_local_saving
    network_mode: host
    build:
      context: ./local_saving/
      dockerfile: ./Dockerfile
    env_file:
      - ./qualityInsp.env
    volumes: # Change path here for other directory for saving
      - ./images/Train:/app/ia-
qualitybox/machine_vision/train_images
    restart: unless-stopped

  inference_client:
    container_name: microservice_inference_client
    network_mode: host
    build:
      context: ./inference_client/
      dockerfile: ./Dockerfile
    env_file:
      - ./qualityInsp.env
    restart: unless-stopped
```

10.12 Umgebungsvariablen Implementierung Druckkontrolle (qualityInsp.env) für GenICam

```
### OVERALL SETTINGS

## MQTT SETTINGS
# Set the hostname or IP address of the MQTT broker
# Possible values: IP-address without using quotation marks
```

```
# No default
MQTT_HOST=172.16.37.2
# DCC broker: 192.168.1.149

# Set the MQTT port, reserved ports for MQTT are 1883 or
#   8883 over SSL
# Possible values: four digit integer
# Default: 1883
MQTT_PORT=8883

# Set the topics to which trigger, images and results are sent
# Possible values: Any path using slash layer spearation and
#                  without using quotation marks
# Defaults: qualityInspection/trigger
#           qualityInspection/image
#           qualityInspection/results
MQTT_TOPIC_TRIGGER=ia/ia/DCCAachen/Demonstrator/detectedAnomaly
MQTT_TOPIC_IMAGE=qualityInspection/image
MQTT_TOPIC_RESULTS=qualityInspection/results

### IMAGE ACQUISITION
# Choose between a continous trigger with a fixed cycle time
#   between each acquisition or a trigger over MQTT
# Possible values: Continuous, MQTT
# No default
TRIGGER=Continuous

# IF "MQTT" set the time between the trigger and image
#   acquisition. If a timestamp in ms since epoch is
#   provided in MQTT trigger message, this timestamp
#   is used. Epoch is defined as January 1, 1970, 00:00:00
#   (UTC). If no timestamp is provided in trigger message,
#   a timestamp is generated as soon as message is
#   received at image acquirer. This acqusition delay can
#   be used to avoid problems caused by varying latency.
#   Use 0.0 for stationary inspection processes to do it
#   as fast as possible without any delay.
# Possible values: Floats in seconds
# Default: 2.0
ACQUISITION_DELAY=1.0

# If "Continuous" set cycle time of your process
# If the cycle time is less than the processing time for each
#   image, an error is thrown
# Possible values: Floats in seconds
# Default: 10.0
```

```
CYCLE_TIME=10.0

## CAMERA SETTINGS
# Set camera interface
# Possible values: GenICam, Cognex
# No default
CAMERA_INTERFACE=GenICam

# Set serial number of camera
# Possible values: String without quotation marks
# Default:    (empty)
SERIAL_NUMBER=

# GENICAM
# If you use the GenTL producer of Matrix Vision as installed
#   in docker-compose you do not ave to change the path.
#   If you change the GenTL producer in the docker-compose file
#   change the path accordingly here.
# Possible values: String without quotation marks
# Default: /opt/mvIMPACT_Acquire/lib/x86_64/mvGenTLProducer.cti
DEFAULT_GENTL_PRODUCER_PATH=/opt/mvIMPACT_Acquire/lib/x86_64/mvGen
TLProducer.cti

# GenICam feature settings:
# There are two ways to configure your camera:
#   1. You use wxPropView of Matrix Vision to configure or any
#       other SDK of the manufacturer of your choice
#      -> save your settings at the end of your configuration in
#            an user set
#   2. You can enter the most important settings directly here
#       manually.
#
# Recommendation: Use 1.
# If your camere does not support user sets, use 2.
#
# ATTENTION:
# If you use 1., the settings of 2. here are ignored. It is only
#   either 1. or 2. possible, not both at the same time.

# 1.
# Load User Set. If Default is set, no user set is loaded and
#   settings in 2. are used.
# Possible values: Default, UserSet1, UserSet2, UserSet3,
#           UserSet4, UserSet5; (number of user sets is camera
#           dependent.)
# Default: Default
USER_SET_SELECTOR=Default
```

```
# 2.
# Determine with width and height in pixels the region of
#   interest (ROI). ROI will be always centered in camera sensor.
#   A value higher than maximum resolution of the camera will
#   set maximum values instead of the values entered here. To
#   find out the highest value, search for the resolution in the
#   specifications of the camera.
#   Specifications are available in the manual or on the website
#   where you bought the camera.
# Possible values: Integer
# Defaults: 800
IMAGE_WIDTH=800
IMAGE_HEIGHT=800
# This Programm allows you to take pictures in monochrome pixel
#   formats (use: "Mono8") and RGB/BRG color pixel formats (use:
#   "RGB8Packed" or "BGR8Packed")
#   If you only have a camera with only one image sensor, you
#   can only take monochrome images. Colored images are not
#   supported for these by this program.
# Possible values: Mono8, RGB8Packed, BGR8Packed
# Default: Mono8
PIXEL_FORMAT=BGR8Packed
# Number of channels (bytes per pixel) that are used in the
#   array (third dimension of the image data array). You do not
#   have to set this value. If None, the best number of channels
#   for your set PIXEL_FORMAT will be used
# Possible Values: None, 1, 3
# Default: None
IMAGE_CHANNELS=None
# Set exporsure time manually
# Possible values: Float or None
# Default: None
EXPOSURE_TIME=None
# Use these functions to let the camera automatically adjust
#   the exporsure time, gain and white balance.
#   Your settings will only be executed if the camera supports
#   this. You do not have to check if the camera supports this.
# Possible values are:
#   - Off:  No automatic adjustment
#   - Once: Adjusted once (Attention: This can lead to different
#                                image quality)
#   - Continuous: Continuous adjustment (Attention: This could
#                                have a big impact on the frame
#                                rate of your camera)
# Defaults: Off
EXPORSURE_AUTO=Off
```

```
GAIN_AUTO=Off
BALANCE_WHITE_AUTO=Off

### LOCAL SAVING
# Set base name of the files (timestamp is always used/added
#   by program). You do not have to set a file name. You can
#   let it empty.
# Possible values: String without quotation marks
# Default:    (empty)
FILE_NAME=

# Set image file type
# Possible values: jpg, png, bmp
# Default: png
FILE_TYPE=png

### CALL DL INFERENCE PLATFORM:
## Overall settings
# Choose between REST and gRPC to connect to serving platform
# Possible values: REST, gRPC
# No default
SERVING_INTERFACE=gRPC

# Set inference serving host's address
# Possible values: IP-address without using quotation marks
# No default
SERVING_HOST=192.168.1.51
# Set inference serving host's port
# Possible values: four digit integer
# No default (usually use 8505 for REST and gRPC 8506)
SERVING_PORT=8506
# DCC: REST_HOST=192.168.1.51
# DCC: REST_PORT=8505
# DCC: GRPC_HOST=192.168.1.51
# DCC: GRPC_PORT=8506

### Everything following this point is only for gRPC interface
# Name of the model
# Possible values: String without quotation marks
# Default: model
MODEL=model

# Set the labels which are used in inference here. Make sure to
#   use the same order that was used to train the model.
# Simply leave unused labels empty.
# Possible values: String without quotation marks
# Defaults:    (empty)
```

```
LABEL_1=QNOK
LABEL_2=QOK
LABEL_3=
LABEL_4=
LABEL_5=
# Define which labels represents a good and which a bad quality.
# Simply leave unused labels empty.
# Possible values: Good, Bad
# Defaults:    (empty)
QUALITY_LABEL_1=Bad
QUALITY_LABEL_2=Good
QUALITY_LABEL_3=
QUALITY_LABEL_4=
QUALITY_LABEL_5=
# Name of the serving signature
# Possible values: String without quotation marks
# Default: serving_default
SIGNATURE_NAME=serving_default
# Name of the input tensor
# Possible values: String without quotation marks
# Default: inputs
INPUT_TENSOR_NAME=inputs
# Array of ints that define the input tensor's shape
# The first dimension is the batch size, the second the image
#   height, the third the image width and the fourth the image
#   channels.
# Possible values: Integer
# No defaults
INPUT_TENSOR_SHAPE_1ST_DIMENSION=1
INPUT_TENSOR_SHAPE_2ND_DIMENSION=224
INPUT_TENSOR_SHAPE_3RD_DIMENSION=224
INPUT_TENSOR_SHAPE_4TH_DIMENSION=3
```

10.13 Umgebungsvariablen Implementierung Kontrolle des Nähprozesses (qualityInsp.env) für Cognex

```
### OVERALL SETTINGS

## MQTT SETTINGS
# Set the hostname or IP address of the MQTT broker
# Possible values: IP-address without using quotation marks
# No default
MQTT_HOST=172.16.37.2
# DCC broker: 192.168.1.149
```

```
# Set the MQTT port, reserved ports for MQTT are 1883 or
#   8883 over SSL
# Possible values: four digit integer
# Default: 1883
MQTT_PORT=8883

# Set the topics to which trigger, images and results are sent
# Possible values: Any path using slash layer spearation and
#                  without using quotation marks
# Defaults: qualityInspection/trigger
#           qualityInspection/image
#           qualityInspection/results
MQTT_TOPIC_TRIGGER=ia/ia/DCCAachen/Demonstrator/detectedAnomaly
MQTT_TOPIC_IMAGE=qualityInspection/image
MQTT_TOPIC_RESULTS=qualityInspection/results

### IMAGE ACQUISITION
# Choose between a continous trigger with a fixed cycle time
#   between each acquisition or a trigger over MQTT
# Possible values: Continuous, MQTT
# No default
TRIGGER=MQTT

# IF "MQTT" set the time between the trigger and image
#   acquisition. If a timestamp in ms since epoch is
#   provided in MQTT trigger message, this timestamp
#   is used. Epoch is defined as January 1, 1970, 00:00:00
#   (UTC). If no timestamp is provided in trigger message,
#   a timestamp is generated as soon as message is
#   received at image acquirer. This acqusition delay can
#   be used to avoid problems caused by varying latency.
#   Use 0.0 for stationary inspection processes to do it
#   as fast as possible without any delay.
# Possible values: Floats in seconds
# Default: 2.0
ACQUISITION_DELAY=1.0

# If "Continuous" set cycle time of your process
# If the cycle time is less than the processing time for each
#   image, an error is thrown
# Possible values: Floats in seconds
# Default: 10.0
CYCLE_TIME=10.0

## CAMERA SETTINGS
# Set camera interface
# Possible values: GenICam, Cognex
```

```
# No default
CAMERA_INTERFACE=Cognex

# Set serial number of camera
# Possible values: String without quotation marks
# Default:    (empty)
SERIAL_NUMBER=

# COGNEX
# Get IP adress from insight explorer
# Possible values: IP-address without using quotation marks
# No default
IP_ADDRESS_COGNEX=172.16.37.149

# To use the job ID number feature, the job to be loaded must
#   be saved with a numerical prefix of 0 to 999. Enter the prefix
#   of the job here. Otherwise the job which is set in the
#   In-Sight Explorer will be used (default job).
# Possible Values: Integer from 0 to 999, default
# Default: default
JOB_ID=1

# Choose if conventional image processing is used and if you
#   want to get the results of this
# Possible Values: True, False
# Default: False
CONVENTIONAL_IMAGE_PROCESSING=True

### LOCAL SAVING
# Set base name of the files (timestamp is always used/added
#   by program). You do not have to set a file name. You can
#   let it empty.
# Possible values: String without quotation marks
# Default:    (empty)
FILE_NAME=

# Set image file type
# Possible values: jpg, png, bmp
# Default: png
FILE_TYPE=png

### CALL DL INFERENCE PLATFORM:
## Overall settings
# Choose between REST and gRPC to connect to serving platform
# Possible values: REST, gRPC
# No default
```

```
SERVING_INTERFACE=gRPC

# Set inference serving host's address
# Possible values: IP-address without using quotation marks
# No default
SERVING_HOST=192.168.1.51
# Set inference serving host's port
# Possible values: four digit integer
# No default (usually use 8505 for REST and gRPC 8506)
SERVING_PORT=8506
# DCC: REST_HOST=192.168.1.51
# DCC: REST_PORT=8505
# DCC: GRPC_HOST=192.168.1.51
# DCC: GRPC_PORT=8506

### Everything following this point is only for gRPC interface
# Name of the model
# Possible values: String without quotation marks
# Default: model
MODEL=model

# Set the labels which are used in inference here. Make sure to
#   use the same order that was used to train the model.
# Simply leave unused labels empty.
# Possible values: String without quotation marks
# Defaults:    (empty)
LABEL_1=QNOK
LABEL_2=QOK
LABEL_3=
LABEL_4=
LABEL_5=
# Define which labels represents a good and which a bad quality.
# Simply leave unused labels empty.
# Possible values: Good, Bad
# Defaults:    (empty)
QUALITY_LABEL_1=Bad
QUALITY_LABEL_2=Good
QUALITY_LABEL_3=
QUALITY_LABEL_4=
QUALITY_LABEL_5=
# Name of the serving signature
# Possible values: String without quotation marks
# Default: serving_default
SIGNATURE_NAME=serving_default
# Name of the input tensor
# Possible values: String without quotation marks
# Default: inputs
```

```
INPUT_TENSOR_NAME=inputs
# Array of ints that define the input tensor's shape
# The first dimension is the batch size, the second the image
#   height, the third the image width and the fourth the image
#   channels.
# Possible values: Integer
# No defaults
INPUT_TENSOR_SHAPE_1ST_DIMENSION=1
INPUT_TENSOR_SHAPE_2ND_DIMENSION=224
INPUT_TENSOR_SHAPE_3RD_DIMENSION=224
INPUT_TENSOR_SHAPE_4TH_DIMENSION=3
```

10.14 LICENSE.txt

Harvesters

License: Apache License, Version 2.0

https://github.com/genicam/harvesters

OpenCV (Open Source Computer Vision Library)

License: 3-clause BSD License

https://github.com/opencv/opencv

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

* Neither the names of the copyright holders nor the names of the contributors may be used to endorse or promote products derived from this software without specific prior written permission.

This software is provided by the copyright holders and contributors "as is" and any express or implied warranties, including, but not limited to, the implied

warranties of merchantability and fitness for a particular purpose are disclaimed.
In no event shall copyright holders or contributors be liable for any direct,
indirect, incidental, special, exemplary, or consequential damages
(including, but not limited to, procurement of substitute goods or services;
loss of use, data, or profits; or business interruption) however caused
and on any theory of liability, whether in contract, strict liability,
or tort (including negligence or otherwise) arising in any way out of
the use of this software, even if advised of the possibility of such damage.

NumPy
License: 3-clause BSD License
https://github.com/numpy/numpy

Copyright (c) 2005-2020, NumPy Developers.
All rights reserved.

Redistribution and use in source and binary forms, with or without modification,
are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice,
  this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice,
  this list of conditions and the following disclaimer in the documentation
  and/or other materials provided with the distribution.

* Neither the names of the copyright holders nor the names of the contributors
  may be used to endorse or promote products derived from this software

without specific prior written permission.

This software is provided by the copyright holders and contributors "as is" and any express or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall copyright holders or contributors be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.

Requests
License: Apache License, Version 2.0
https://github.com/psf/requests

Copyright (c) 2013 Kenneth Reitz

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

See the License for the specific language governing permissions and
limitations under the License.

---

grpcio

License: Apache License, Version 2.0

https://github.com/grpc/grpc.io

Copyright (c) 2020 gRPC Authors

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.

TensorFlow Serving

License: Apache License, Version 2.0

https://github.com/tensorflow/serving

10.15 MATRIX VISION GmbH - Special Software License Agreement for mvIMPACT Acquire SDK camera driver software.docx

MATRIX VISION GmbH - Special Software License Agreement for mvIMPACT Acquire SDK camera driver software

This Software License Agreement (Agreement) is a special legal Agreement between you and MATRIX VISION GmbH (MV) for the MATRIX VISION software product mvIMPACT Acquire SDK, which includes computer software and may include associated media, printed materials or online or electronic documentation. In addition to the provisions herein, the General Terms and Conditions (“Allgemeine Geschäftsbedingungen”) of MV, current status: November 2015, are applicable. In case of a conflict, the Agreement herein shall prevail. The license grant, is only valid, if you accept this Agreement and the General Terms and Conditions (“Allgemeine Geschäftsbedingungen”) of MV, current status: November 2015, and in particular do not refer to or use General Conditions that in any way are in conflict with or provide additional conditions not provided herein. You may contact MV to obtain the current version (<http://www.matrix-vision.com/de-agb.html> but URL is subject to change).

1. DEFINITIONS

“Software” means all of the contents of the files provided by MV to you under the product name mvIMPACT Acquire SDK.

“Documentation” means the User's Guides, User's Manuals and Reference Manuals, if any, accompanying delivery of the Software. Documentation may be delivered in printed and/or electronic and/or online forms.

2. LICENSE GRANT

The licensed Software and Documentation shall at all times remain the property of MV and/or its licensors, and you, as licensee, shall have no right, title, or interest in the Software, except as expressly set forth in this Agreement.

a) Use with MV Hardware

MV grants to you a nonexclusive license to install and use the Software as provided herein on the applicable hardware.

b) Use with Third Party Hardware as a courtesy

MV grants to you free of charge, as a courtesy a nonexclusive, revocable license to install and use the Software as provided herein with third party hardware, i.e.

hardware not manufactured or delivered to you by MV under the following conditions:

- Any of your contractual rights related to products acquired by you from MV, in particular but not limited to delivery, defects, support, updates, documentation, transfer shall not apply.
- MV has neither designed nor tested the Software for its use with Third Party Hardware and, therefore, you install and use the Software at your own risk.
- The provisions related to Donations (sec. 516 et seq. of the German Civil Code) shall apply instead of the provisions regarding defects and liability of the the General Terms and Conditions (“Allgemeine Geschäftsbedingungen”) of MV and this agreement.

2.3. COMPLIANCE WITH LICENSE

You agree that upon request from MV or MV's authorized representative, you will within thirty (30) days fully document and certify that use of any and all MV Software at the time of the request is in conformity with your valid license from MV.

2.4. DOCUMENTATION

You may make and use a reasonable number of copies of any Documentation, provided that such copies shall be used only for internal purposes and are not to be republished or distributed (either in hard copy or electronic form) beyond your premises.

2.7 SYSTEM ENVIRONMENT

You are responsible for providing the system environment in accordance with the system requirements for Software.

2.8 SUBLICENCES

You may sublicense the Software only upon prior written consent by MV. In case such consent is given by MV to you, you are obliged to sublicense the Software with a license whose terms and conditions are at least as restrictive as the terms in this Agreement.

3. DESCRIPTION OF OTHER RIGHTS AND LIMITATIONS

3.1. RENTAL

You may not rent, lease or lend the Software.

3.2. SOFTWARE TRANSFER

Except as may be expressly prohibited herein, you may transfer all your rights to use the Software to a third party provided that (1) you also transfer all your obligations out of this Agreement, the Software and all other software bundled with the

Software, including all copies, updates and prior versions, to such third party; (2) you retain no copies, including backups and copies stored on a computer; and (3) the receiving party accepts the terms and conditions upon which you legally obtained a license to the Software.

3.3. BACKUP COPY

You may make a reasonable number of copies of the Software to backup devices such as hard disks, optical media, or tape and a reasonable number of physical DVD media backup copies of the Software solely to replace the original copy provided to you if the original copy is damaged or destroyed. All rights not specifically granted to you herein are retained by MV.

3.4. LIMITATION ON REVERSE ENGINEERING, DECOMPILATION, DISASSEMBLY AND MODIFICATION

You may not reverse engineer, decompile, disassemble or otherwise attempt to discover the source code of the Software, except to the extent you may be expressly and mandatorily permitted under applicable law, it is essential to do so in order to achieve interoperability of the Software with another software program, and you have first requested MV to provide the information necessary to achieve such interoperability and MV has not made such information available.

Any information supplied by MV or obtained by you, as permitted hereunder, may only be used by you for the purpose described herein and may not be disclosed to any third party.

You may not modify, adapt, or otherwise alter the Software except as expressly permitted herein. In particular, you may not attempt to remove the license protection mechanism from the Software.

3.5. TRADEMARKS

This Agreement does not grant you any rights in connection with any trademarks or service marks of MV or of MV´s suppliers. You may not use the name, trademarks, or service marks of MV or of MV´s suppliers in any advertising, promotional literature or any other material, whether in written, electronic, or other form, distributed to any third party, except as expressly permitted by MV.

3.6. UPGRADES

If the Software is an upgrade to a previous version of the Software, any obligation MV may have to support the previous version of the Software may be ended within one (1) year upon availability of the upgrade.

3.7. LICENSE FOR THIRD PARTY SOFTWARE

MV has been granted licenses to distribute certain third party software. As a condition of those licenses, MV is required to distribute the software subject to specific terms and conditions, which may be different from or additional to those contained herein for MV's Software. You understand and agree that acceptance of this Agreement also confirms your acceptance of the applicable provisions for use, including the restrictions on use, of such third party software. You may contact MV to obtain the current applicable provisions. Any breach of the applicable provisions of any third party's license terms shall also be considered a material breach of this Agreement.

3.8. TERMINATION

Without prejudice to any other rights, MV may terminate this Agreement for good cause if you fail to comply with the terms and conditions of this Agreement. In such event, you must destroy all copies of the Software and all of its components.

4. INTELLECTUAL PROPERTY RIGHTS

The Software is protected by copyright and other intellectual property laws and treaties. All title and intellectual property rights in and to the Software, including, but not limited to, any digital images or example programs, incorporated into the Software, the Documentation and any copies of the Software are owned by MV or its suppliers. The Software is licensed, not sold.

5. RIGHTS IN CASE OF DEFECTS

In addition to the provisions regarding the rights in case of defects of the General Terms and Conditions (“Allgemeine Geschäftsbedingungen”) of MV the following provisions shall be applicable:

5.1 Defects must be notified in writing and, unless it is an unreasonable burden, with a comprehensible description of the error symptoms, as far as possible evidenced by written recordings, hard copies or other documents demonstrating the defects. The notification of the defect should enable the reproduction of the error.

5.2 Subsequent performance may also take place through the delivery or installation of a new program version or a reasonable work-around of the Software. If the defect does not or not substantially impair the functionality, then MV is entitled, to remedy the defect by delivering a new version or an update within a reasonable time as part of its version, update and upgrade planning.

9. HAZARDOUS USES

The Software is not designed and/or intended for use in connection with any application requiring fail-safe performance such as the operation of nuclear power facilities, air traffic control or navigation systems, weapon control systems, life support systems, or any other system whose failure could lead to death, personal injury, or severe physical or environmental damage. You agree that MV will have no

responsibility of any nature, and you are solely responsible for any expense, loss, injury or damage incurred as a result of such use of the Software.

10. ENTIRE AGREEMENT

This Agreement contains the entire, final and exclusive understanding between MV and you, and may not be modified or amended except by written instrument, executed by authorized representatives of MV and you. Amendments or additions to this Agreement must be made in writing to be effective. This shall also apply to amendments of this written form requirement.

11. APPLICABLE LAW

This Agreement is governed by the laws of Germany except for the UN Sales Convention (United Nations Convention on Contracts for the International Sale of Goods dated 11.4.1980).

12. LEGAL VENUE

The courts in Stuttgart/Germany shall have exclusive jurisdiction over all disputes under and in connection with this Agreement, provided that you are a merchant within the meaning of the German Commercial Code or if upon the commencement of legal proceedings, you have no place of business or ordinary residence in the Federal Republic of Germany.

13. SEVERABILITY

If any provisions of this license is held to be illegal, invalid or enforceable for any reason, then such provision will be enforced to the maximum extent permissible and the remainder of the provisions of this license will remain in full force and effect.

Status: November 2015

10.16 Installation und Nutzung von Docker-Compose zur Ausführung der Microservices

Sofern noch nicht auf dem Rechner vorhanden, muss Docker zunächst installiert werden. Für Linux können hierzu die folgenden Befehle in die Kommandozeile eingegeben werden:

```
apt-get install -y python3 python3-pip libffi-dev python3-
dev libssl-dev
pip3 install setuptools docker-compose
docker-compose –version
```

Die Installation ist erfolgreich, wenn die Versionsnummer von Docker-Compose zurückgegeben wird. Anschließend kann die entsprechende Docker-Compose Datei (s. Anhang 10.10.2 für Cognex, 10.11.2 für GenICam) mit der zugehörigen Verzeichnisstruktur (s. Anhang 10.10.1 für Cognex, 10.11.1 für GenICam) und den darin aufgeführten Skripten ausgeführt werden. Durch Eingabe des Befehls `docker-compose up --build` in der Kommandozeile des Arbeitsverzeichnisses der Docker-Compose Datei werden die Container gebaut und ausgeführt.

Sofern nach der Installation Änderungen an den Umgebungsvariablen durchgeführt werden, müssen die Container neu gestartet werden. Dazu werden die aktuellen Container zunächst mit `docker-compose down` in der Kommandozeile beendet und anschließend wieder mit den neuen Umgebungsvariablen mittels `docker-compose up` gestartet. Die Befehle werden stets im Arbeitsverzeichnis der docker-compose.yml Datei ausgeführt.

11 Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig angefertigt habe. Es wurden nur die in der Arbeit ausdrücklich benannten Quellen und Hilfsmittel benutzt. Wörtlich oder sinngemäß übernommenes Gedankengut habe ich als solches kenntlich gemacht.

Ort, Datum

Unterschrift