Bosch Engineering

BOSCH

HOCHSCHULE HEILBRONN

**Electrical Systems Engineering (SPO 01)**

# End-to-End MLOps: Ganzheitliche Realisierung von Machine Learning Operations am praktischen Beispiel

Matthias Benedikt Fassian*

30. Januar 2025

betreut von Johannes Vogel (BEG/ECP1)
eingereicht bei Prof. Dr. Nicolaj Stache (HHN)

*210563, mfassian@stud.hs-heilbronn.de

# Danksagung

An dieser Stelle möchte ich allen danken, die zum Gelingen dieser Thesis beigetragen haben.

Mein besonderer Dank gilt dem Unternehmen Bosch Engineering, das mir durch die Bereitstellung der notwendigen Rahmenbedingungen und die fachliche Unterstützung die Durchführung dieser Arbeit ermöglicht hat. Ebenso möchte ich meinem Betreuer Johannes Vogel für seine wertvolle Unterstützung, die konstruktiven Hinweise und die fachliche Begleitung während des gesamten Projekts danken.

Ein herzlicher Dank gilt auch meinen Korrekturlesern Christina, Matthias und Ralf für ihre wertvollen Anmerkungen und ihre sorgfältige Durchsicht meines Manuskripts. Ihre Unterstützung hat wesentlich zur Qualität dieser Arbeit beigetragen.

Nicht zuletzt möchte ich mich bei meiner Verlobten Anja bedanken. Ihre Unterstützung, ihr Verständnis und ihre motivierenden Worte haben mir stets neuen Antrieb gegeben und zum erfolgreichen Abschluss dieses Projekts beigetragen.

Diese Arbeit ist das Ergebnis einer intensiven und spannenden Zeit, die ohne die Unterstützung aller Beteiligten nicht möglich gewesen wäre.

# Abstrakt

Der Einsatz von Machine Learning (ML) gilt als Schlüsseltechnologie zur Transformation von Geschäftsprozessen, Dienstleistungen und Produkten. Dennoch scheitern zahlreiche ML-Projekte in Unternehmen daran, entwickelte Modelle erfolgreich in die Produktionsumgebung zu überführen und langfristig zu betreiben. Die Ursachen liegen in nicht unmittelbar sichtbaren Herausforderungen von ML-Anwendungen, die sich erst im späteren Projektverlauf zeigen, jedoch von Beginn an berücksichtigt werden müssen. Besonders entscheidend sind dabei die Anforderungen an Qualität, Verfügbarkeit, Sicherheit und Nachvollziehbarkeit, die im Vergleich zu klassischen Softwareprojekten deutlich komplexer sind und - wenn nicht erfüllt - beim Betrieb der Modelle sehr hohe Kosten verursachen. In diesem Zusammenhang spricht man deshalb auch von „technischen Schulden“, die aufgrund von Vernachlässigung der besonderen Anforderungen während der Entwicklung entstehen und langfristig zum Scheitern des Projektes führen können. Dies verdeutlicht die Notwendigkeit einer systematischen Herangehensweise bei der Entwicklung von ML-Projekten sowie der effizienten Verwaltung des gesamten Modelllebenszyklus im Betrieb.

Die vorliegende Arbeit untersucht die Konzepte von Machine Learning Operations (MLOps) als Lösung für diese Herausforderungen. MLOps erweitert etablierte DevOps-Praktiken um Methoden und Werkzeuge für die Entwicklung, Bereitstellung, Überwachung und kontinuierliche Optimierung von ML-Modellen. Ziel der Arbeit war die Entwicklung eines universellen Systems, das die Prinzipien und Funktionalitäten von MLOps vereint und praktisch anwendet. Dafür wurden bewährte Ansätze und Komponenten identifiziert, bewertet und gemeinsam als flexible Multi-Container-Anwendung implementiert.

Das entwickelte System wurde am praktischen Beispiel der Abstandsmessung mit einer Rückfahrkamera unter Laborbedingungen evaluiert. Die Ergebnisse zeigen, dass ein ML-Modell effizient bereitgestellt sowie automatisiert trainiert, überwacht und optimiert werden kann. Gleichzeitig erfüllt das entwickelte System zentrale Anforderungen an Nachvollziehbarkeit, Reproduzierbarkeit und Erklärbarkeit.

Die Arbeit leistet einen wichtigen Beitrag zum Verständnis der komplexen Thematik und stellt mit der entwickelten Softwarelösung eine solide Grundlage für die praktische Anwendung von MLOps bereit. Die Ergebnisse sind die Basis für den langfristig erfolgreichen Einsatz von Machine Learning in Projekten und Produkten eines Unternehmens.

# Inhaltsverzeichnis

# Abkürzungsverzeichnis

**AIOps** Artificial Intelligence for IT Operations

**API** Programmierschnittstelle

**AWS** Amazon Web Services

**CD** Continuous Deployment

**CI** Continuous Integration

**DevOps** Development and Operations

**GPIO** General Purpose Input/Output

**HTTP** Hypertext Transfer Protocol

**I2C** Inter-Integrated Circuit

**IDE** Integrierte Entwicklungsumgebung

**IoT** Internet of Things

**JSON** JavaScript Object Notation

**KI** Künstliche Intelligenz

**ML** Machine Learning

**MLOps** Machine Learning Operations

**PWM** Pulsweitenmodulation

**SQL** Strukturierte Abfragesprache

**UI** Benutzeroberfläche

**VSCode** Visual Studio Code

# 1. Einleitung

## 1.1. Motivation

Machine Learning (ML) gilt als Schlüsseltechnologie für die Zukunft unserer Wirtschaft. Mit Blick auf den globalen Wettbewerb setzen immer mehr Unternehmen zur Transformation ihrer Prozesse, Dienstleistungen und Produkte auf ML oder planen in naher Zukunft, in diese Technologie zu investieren [1]. Allein für das Jahr 2024 wurde eine Verdopplung der Investitionen zum Vorjahr erwartet [2]. Eine Studie der Unternehmensberatung Gartner zeigt neben dem großen Interesse an Entwicklung und Einsatz von ML jedoch auch, dass ein großer Teil dieser Projekte aufgrund von falschen Erwartungen und Prioritäten scheitert [3]. Eine Ursache dafür liegt in der Überführung des ML-Modells von der Entwicklung zur Produktionsumgebung, welche im Vergleich zu herkömmlichen Softwareprojekten deutlich komplexer ist [4]. Auch bei erfolgreich in die Produktionsumgebung überführten ML-Projekten ist der operative Betrieb herausfordernd, da Leistungsfähigkeit, Sicherheit und Zuverlässigkeit der ML-Anwendung kontinuierlich gewährleistet werden müssen. Zurückzuführen sind diese oft unerwarteten Probleme auf den *hidden technical debt in machine lerarning systems*, der nicht unmittelbar erkennbare technische Schulden beschreibt, die während der Softwareentwicklung und insbesondere bei ML-Projekten entstehen. Sie resultieren aus kurzfristigen Entscheidungen wie dem Einsatz nicht skalierbarer Module oder der Vernachlässigung von Best Practices in der Softwareentwicklung. Diese Nachlässigkeiten manifestieren sich später in Form von erhöhtem Wartungsaufwand, unerwarteten Ausfallzeiten und Hürden bei der Weiterentwicklung, was die langfristige Leistungsfähigkeit und Zuverlässigkeit der Systeme beeinträchtigt [5].

Vor diesem Hintergrund gewinnt das Themenfeld Machine Learning Operations (MLOps) zunehmend an Bedeutung. MLOps stellt eine Erweiterung der Development and Operations (DevOps)-Praktiken dar, die speziell auf die Bedürfnisse von ML-Projekten zugeschnitten sind. Als oberstes Ziel verfolgt MLOps, den gesamten Lebenszyklus von ML-Modellen - von der Entwicklung über die Bereitstellung bis hin zum Betrieb - effizient zu gestalten, indem unterstützende Strukturen und Prozesse implementiert werden. Dadurch können technische Schulden frühzeitig erkannt und vermieden werden, was sowohl die Robustheit, Skalierbarkeit und Sicherheit als auch die Nachvollziehbarkeit, Erklärbarkeit und Anpassungsfähigkeit von ML-Systemen langfristig gewährleistet [6, S. 1–3].

MLOps führt eine systematische und im Projekt standardisierte Herangehensweise ein, um die Komplexität des Betriebs und der Weiterentwicklung von ML-Modellen in der Produktionsumgebung zu bewältigen und Prozesse zu optimieren. So wird sichergestellt, dass Modelle versioniert, getestet, überwacht sowie kontinuierlich angepasst werden. Damit kann MLOps nicht

nur die erfolgreiche Überführung von ML-Projekten in Produktionsumgebungen sicherstellen, sondern ist auch der Schlüssel für den langfristig erfolgreichen Einsatz von ML in Produkten und Dienstleistungen des Unternehmens [7, Kapitel 1].

## 1.2. Zielsetzung

Das Ziel dieser Arbeit ist die Entwicklung eines Systems zur Realisierung von MLOps im Kontext eines ML-Projekts. Bewährte Prinzipien und Komponenten sollen dafür identifiziert, aufgeschlüsselt und als Bestandteil einer Softwarelösung realisiert werden. Anhand eines Praxisbeispiels soll die Umsetzung von MLOps in einem realen Anwendungsfall gezeigt und bewertet werden. Die Arbeit soll die benötigten Grundlagen bündeln und die Vorteile der konsequenten Umsetzung von MLOps aufzeigen. Die Softwarelösung soll dabei als universelle Grundlage für die erleichterte Anwendung der MLOps-Methoden in weiteren Projekten dienen und so zum langfristig erfolgreichen Einsatz von ML in Produkten und Dienstleistungen des Unternehmens beitragen.

## 1.3. Einführung in das praktische Beispiel

Das Praxisbeispiel dieser Arbeit widmet sich der Implementierung eines Kamera-basierten Systems zur Abstandsmessung am Fahrzeug und greift damit ein aktuelles Thema aus der Automobilindustrie auf [8][9]. In modernen Fahrzeugen werden als Komponenten der akustischen und optischen Einparkhilfe sowohl Ultraschallsensoren als auch Kameras verbaut [10]. Beide Sensorsysteme nehmen dabei Messwerte auf, aus denen der Abstand zu einem Hindernis ermittelt werden kann. Konkret kann ein ML-Modell anhand des Bildes der Rückfahrkamera den Abstand zu einem Hindernis schätzen und so die Daten des Ultraschallsensors um Redundanz erweitern oder vollständig ersetzen. Dadurch kann letztlich auf die Ultraschallsensoren verzichtet werden, was den Fertigungsaufwand reduziert, Bauteilkosten senkt und Reparaturkosten verringert [11].

Diese Anwendung bietet neben ihrer praktischen Relevanz eine solide Grundlage für die Erprobung von MLOps, da sie auf einer einfach zu verstehenden und gleichzeitig vielseitigen Anwendung im Bildverarbeitungsbereich basiert. Die vorhandene Redundanz beider Sensorsysteme – Ultraschallsensor und Kamera – erlaubt durch den verlässlichen Referenzwert die Generierung eines umfangreichen, beschrifteten Datensatzes zur Modellentwicklung und kontinuierlichen Validierung der Modellqualität. Für die Entwicklung wird ein vereinfachtes Fahrzeugmodell genutzt, sodass die Komplexität der Implementierung in ein reales Fahrzeug entfällt und Veränderungen an der Umgebung gut simuliert werden können. Damit können die Komponenten der MLOps-Lösung unter realitätsnahen Bedingungen eingesetzt und gleichzeitig isoliert getestet werden.

# 2. Stand der Technik

Die Prinzipien und Methoden von MLOps basieren auf den bewährten Praktiken der Softwareentwicklung und des IT-Betriebs, die durch steigende Anforderungen über die Zeit entstanden sind. Dabei haben sich verschiedenste Tools und Frameworks etabliert, die als Hilfsmittel zur Realisierung der benötigten Komponenten dienen. Für die erfolgreiche Nutzung dieser Hilfsmittel ist das Verständnis der zugrundeliegenden Ideen und Technologien notwendig, das nachfolgend vermittelt werden soll.

## 2.1. Klassische Softwareentwicklung und DevOps

### 2.1.1. Lokale Entwicklung

Die Arbeit am Quellcode eines Softwareprodukts erfolgt häufig in einem Code-Editor oder einer Integrierte Entwicklungsumgebung (IDE). Neben dem Editor unterstützen verschiedene Tools die Entwicklung, welche sowohl über die Kommandozeile universell zugänglich als auch auf Wunsch durch eine grafische Benutzeroberfläche intuitiv bedienbar sind. Dazu zählen je nach Anwendungsfall beispielsweise eine einfache Dateiverwaltung, Code-Highlighting, Autovervollständigung, die Integration von Compilern, Debugging-Tools und viele weitere Funktionen [12]. Eine der am verbreitetsten Entwicklungsumgebungen ist die von Microsoft entwickelte Visual Studio Code (VSCode), die plattformübergreifend verfügbar und durch eine Vielzahl an Erweiterungen universell einsetzbar ist [6, S. 79]. Für die Entwicklung von ML-Modellen mit Python haben sich Jupyter Notebooks etabliert, die das Teilen des Quellcodes in Code- und multimediale Markdown-Blöcke erlauben und interaktive Steuerungselemente bereitstellen. Dadurch kann der Code schrittweise ausgeführt, dokumentiert und die Ausgabe visualisiert werden [13].

### 2.1.2. Versionskontrolle

An der Entwicklung eines Softwareprodukts sind verschiedene Akteure beteiligt, die in unterschiedlichen Phasen des Entwicklungsprozesses zusammenarbeiten. Diese Zusammenarbeit erfordert eine klare Standardisierung und Strukturierung von Daten und Prozessen. Als zentrale Komponente für die Kollaboration wird ein Versionskontrollsystem eingesetzt, das den Quellcode verwaltet, die Dokumentation von Änderungen ermöglicht und Konflikte bei der Zusammenführung vermeidet. Durch die Zentralisierung des Dateisystems als Repository können Entwickler gleichzeitig an verschiedenen Teilen des Codes arbeiten und ihre Änderungen in das Dateisystem einpflegen [14]. Abbildung 2.1 zeigt den standardisierten Ablauf bei Codeänderungen im Versionskontrollsystem *git*, dass sich als Standard etabliert hat und von mehr als 96 Prozent der Entwickler weltweit genutzt wird [15].

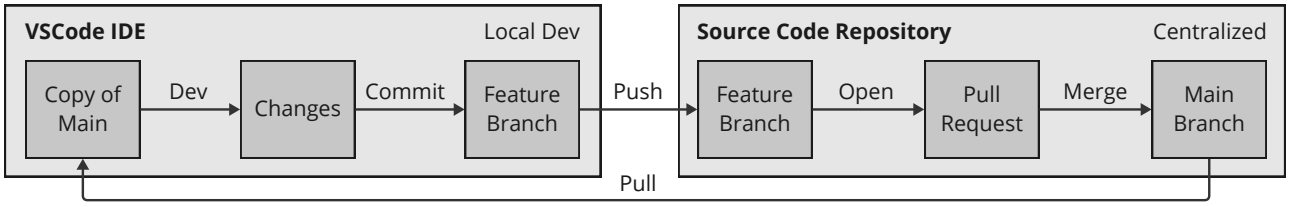



Abbildung 2.1.: Ablauf bei Codeänderungen im Versionskontrollsystem git

Die Bearbeitung des Codes erfolgt üblicherweise in einer lokalen IDE, die eine Kopie des Repositories enthält. Durchgeführte Änderungen werden mit Kommentar in einen lokalen Branch eingecheckt, der per Push in das zentrale Dateisystem geladen wird. Dort wird ein Pull-Request erstellt, die alle Änderungen dokumentiert, überprüft und nach Zustimmung durch andere Entwickler in den Main-Branch überführt. Das Synchronisieren der Branches mit einem zentralen Dateisystem, auch Repository genannt, ist für die Kollaboration unerlässlich. Plattformen wie GitHub, GitLab oder Bitbucket bieten das Hosting von Repositories sowie weiteren Funktionen zur Zusammenarbeit als digitale Dienstleistung an.

### 2.1.3. Standardisierung

Zur Verbesserung der Codequalität und langfristigen Wartbarkeit ist es entscheidend, Code und Prozesse zu standardisieren. Dies beginnt bei der Strukturierung des Codes in aufgabenspezifische Module, der Benennung von Variablen und Funktionen nach festgelegten Naming Conventions, der ausführlichen Dokumentation von Änderungen und der Einhaltung von Formatierungsrichtlinien. So sollten zum Beispiel die Variablennamen möglichst aussagekräftig sein und Funktionen mit Docstrings umfänglich beschrieben werden [6, S. 82–84]. Die Einhaltung dieser Vorgaben kann durch einen Linter automatisiert überprüft werden. Dieser analysiert den Code schon während der Entwicklung und weist sofort auf Fehler hin. Für Python-Projekte hat sich das Tool *Flake8* etabliert, das als Erweiterung in VSCode verfügbar ist und individuell für das Projekt konfiguriert werden kann. Eine Verletzung der Vorgaben wird dann direkt im Quellcode, ähnlich wie ein Syntaxfehler, angezeigt [16].

Zur Standardisierung von Softwarekomponenten (Units) hat sich das Test Driven Development aus der agilen Softwareentwicklung etabliert. Dabei werden verschiedene Tests parallel zur Implementierung des eigentlichen Codes geschrieben, indem iterativ zunächst eine bestimmte Anforderung im Test definiert und anschließend im Code umgesetzt wird. Dadurch sind die Funktionalitäten einzelner Units klar definiert, wodurch sie unabhängig voneinander entwickelt und getestet werden können [17]. Das Ausführen der implemtierten Tests erfolgt automatisiert mit Frameworks wie *Pytest* oder *Unittest*. Änderungen dürfen erst dann in den Main-Branch übernommen werden, wenn Linting und Tests erfolgreich abgeschlossen wurden, sodass nur qualitativer und funktionaler Code in die Produktionsumgebung gelangt [18, S. 19].

Im Quellcode verwendete Pakete und Abhängigkeiten müssen ebenfalls dokumentiert werden. Dafür eignet sich der Paketmanager *pip*, der die Paketinstallation verwaltet und automatisiert. Im einfachsten Fall werden die benötigten Pakete dafür mit Angabe der Version in der Datei `requirements.txt` gespeichert. Um Konflikte mit anderen Paketen zu vermeiden, kann die Entwicklung in einem virtuellen Environment erfolgen, das die Abhängigkeiten isoliert und die Kompatibilität sicherstellt [19].

Diese Maßnahmen zur Standardisierung machen den Quellcode lesbar, nachvollziehbar sowie anpassbar und stellen die fehlerfreie Ausführung sicher. Sie sind die Grundlage für die Automatisierung von Prozessen, die im Rahmen von MLOps eine zentrale Rolle spielt.

### 2.1.4. DevOps und Automatisierung

Development and Operations (DevOps) ist ein Ansatz, der Entwicklung und Betrieb von Softwareprodukten zusammenführt. Traditionell wurden beide Bereiche organisatorisch und funktional isoliert betrachtet, was zu einer Reihe von Problemen führt, die Effizienz und Qualität beeinträchtigen. So können zum Beispiel Fehler in der Entwicklung erst spät im Betrieb erkannt werden, was dann hohe Kosten und Verzögerungen verursacht. Durch die geschlossene Betrachtung können diese Probleme vermieden und eine höhere Effizienz erreicht werden. DevOps basiert auf den Konzepten der Zusammenarbeit und Standardisierung und erweitert diese um Tools zur Automatisierung und zum Betrieb [20].

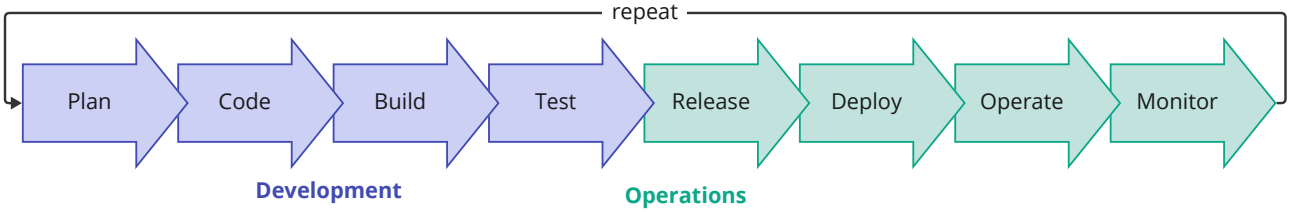



Abbildung 2.2.: DevOps Workflow als geschlossener Kreislauf

Zur Automatisierung von Prozessen werden Pipelines eingesetzt, die nach einem Commit, also nach Übertrag eines neuen Softwarestands in das Versionskontrollsystem, oder nach dem Zusammenführen von Änderungen mit dem Main-Branch als automatisierter Prozess ausgeführt werden. Im DevOps Kontext spricht man dabei von Continuous Integration (CI) und Continuous Deployment (CD). CI-Pipelines überprüfen die Einhaltung der Standardisierungsvorgaben, führen Unittests aus und starten den Build-Prozess. CD-Pipelines übernehmen nachfolgend die Konfiguration der Infrastruktur, die Bereitstellung des Codes in die Test- oder Produktionsumgebung und führen Integrationstests durch [21]. Die Definition der Pipelines erfolgt mittels einer Konfigurationsdatei, die von CI/CD-Tools wie *Github Actions* interpretiert und auf dedizierten Ressourcen ausgeführt wird.

Für die erfolgreiche Realisierung von DevOps ist auch die Überwachung der Prozesse und der Infrastruktur entscheidend. Dafür werden Monitoring-Tools eingesetzt, die den Status der Pipelines und der Infrastruktur überwachen und Stakeholder bei Fehlern oder Ausfällen benachrichtigen. Die Überwachung erfolgt in Echtzeit und kann in Dashboards visualisiert werden, die viele Dienstleister direkt bereitstellen. Zur zentralen Überwachung und Kollaboration eignen sich individuelle Dashboards, die zum Beispiel mit dem Tool Grafana konfiguriert und verwaltet werden können. Dieses kommuniziert über Adapter mit den verschiedenen Diensten und stellt die Daten visuell aufbereitet in Echtzeit dar [22].

Die DevOps Praktiken und Tools umfassen noch weitere Themen wie die Konfigurationsverwaltung, das Ressourcenmanagement oder die Sicherheit. Mit DevOps können Änderungen iterativ und kontinuierlich in die Produktionsumgebung überführt und überwacht werden, was die Entwicklungszyklen verkürzt und die Zuverlässigkeit des Systems nachhaltig erhöht.

## 2.2. Entwicklung von ML-Modellen in Python

Machine Learning ist ein Teilgebiet der Künstlichen Intelligenz (KI), das sich mit der Entwicklung von Algorithmen und Modellen beschäftigt, die aus Daten lernen können, indem sie Muster in diesen erkennen. Dadurch sind Computer in der Lage, Probleme zu lösen, die Wissen über die reale Welt erfordern, und können Entscheidungen treffen, die subjektiv erscheinen [23, S. 2–3]. Die Basis hierfür sind Netze aus Neuronen, siehe Abbildung 2.3, die aus einer Input-Schicht, beliebig vielen versteckten Schichten (Layer) und einer Output-Schicht bestehen. Jedes Neuron wendet eine Aktivierungsfunktion an, die den Input-Wert transformiert und festlegt, ob und wie stark das Signal an die nächste Schicht weitergegeben wird. Die Neuronen benachbarter Schichten sind miteinander verbunden und übertragen Informationen. Dabei werden jedem Neuron ein Gewicht und ein Bias zugeordnet, die während des Trainings angepasst werden, um ein bestimmtes Verhalten zu approximieren [24, Kapitel 1].

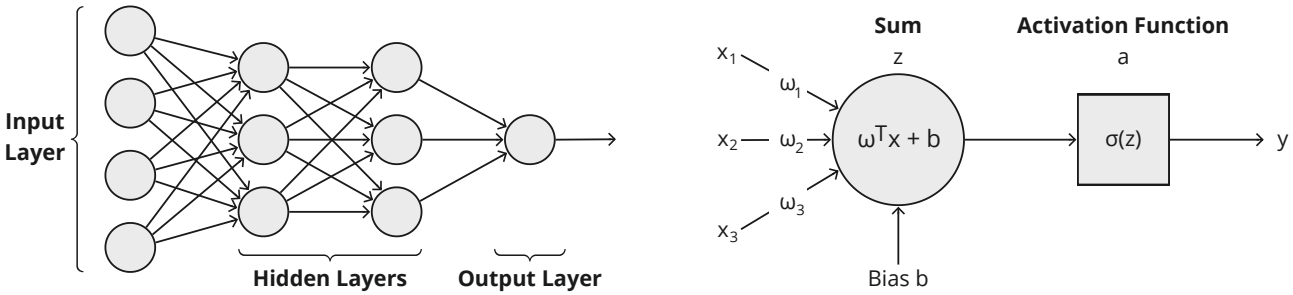



Abbildung 2.3.: Neuronales Netz (links) und künstliches Neuron (rechts) [24]

Mithilfe vieler Schichten von Neuronen ist das neuronale Netz in der Lage, durch Kombination von einfachen Funktionen auch komplexe Zusammenhänge wiederzugeben. Dafür lernt das

Modell nicht nur die Auswirkung von Merkmalen (Features) auf das Ergebnis, sondern auch die Features selbst. Man spricht von einem „Deep Neural Network“ oder „Deep Learning“ [23].

Als Basis für die Entwicklung von ML-Modellen hat sich die Programmiersprache Python etabliert, die sich durch eine einfache Syntax, gute Lesbarkeit und die Verfügbarkeit von umfangreichen Bibliotheken und Frameworks auszeichnet [25]. Neben Bibliotheken zur Datenverarbeitung wie *numpy*, *pandas* oder *opencv* sind mit dem Ziel der erleichterten Entwicklung von ML-Modellen mit der Zeit verschiedene Frameworks entstanden. Am verbreitetsten sind *TensorFlow*, veröffentlicht 2015 von Google, und *PyTorch*, entwickelt 2016 von Meta [26]. Beide Frameworks sind quelloffen, kostenfrei und werden von einer großen Entwickler-Community unterstützt. Während *scikit-learn* auf klassische ML-Modelle spezialisiert ist und zahlreiche Algorithmen sowie Werkzeuge bereitstellt [27], kommt für das praktische Beispiel *TensorFlow* in Kombination mit ausgewählten Algorithmen aus *scikit-learn* zum Einsatz. Die Programmstruktur lässt sich in universelle Schritte unterteilen, die in Abbildung 2.4 dargestellt sind.

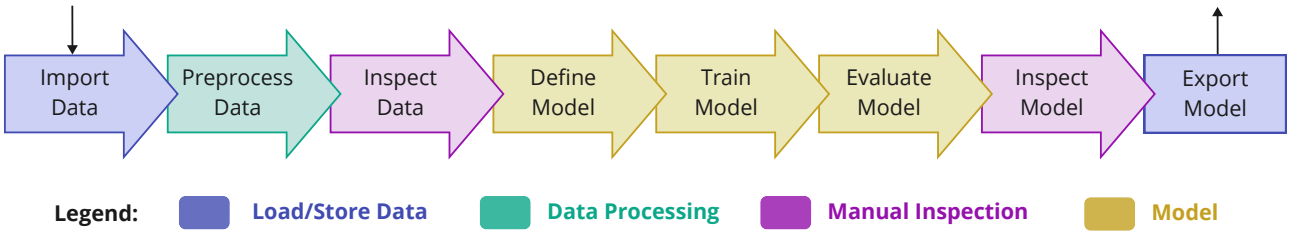



Abbildung 2.4.: Schritte bei der Entwicklung von ML-Modellen in Python

Zuerst werden die Trainingsdaten aus einer Datenquelle geladen und in ein geeignetes Format gebracht. Anschließend folgt die Datenaufbereitung, bei der die Daten normalisiert, in Trainings- und Testdaten aufgeteilt und bei Bedarf anschließend manuell inspiziert werden. Sind die Trainings- und Testdaten vollständig sowie ausreichend qualitativ, wird auf Basis der gewählten Architektur ein Modell definiert, das die Schichten, die Anzahl der Neuronen und die Aktivierungsfunktionen umfasst. Das Modell wird dann trainiert, indem die Gewichte und Biases anhand der Trainingsdaten iterativ angepasst werden. Die Evaluation und Validierung des Modells mit den Testdaten liefert verschiedene Metriken zur Bewertung der Modellqualität. Die Ergebnisse sowie die Trainingshistorie werden manuell ausgewertet und dokumentiert. Abschließend wird das trainierte und evaluierte Modell als Datei exportiert.

Der beschriebene sequentielle Prozess wird als *Model training pipeline* bezeichnet. Die Genauigkeit des hieraus resultierenden Modells wird durch die Trainingsdaten, die Modellarchitektur und weitere Hyperparameter beeinflusst, deren Optimierung experimentell erfolgt. Für gute Ergebnisse müssen die Trainingsdaten möglichst repräsentativ sein, präzise Labels enthalten und durch eine sorgfältige Datenaufbereitung optimal für das Modell nutzbar gemacht werden.

Die Modellarchitektur muss so gewählt werden, dass sie die Komplexität des Problems abbilden kann, jedoch gleichzeitig ressourcenschonend ist und nicht zu Überanpassung neigt. Hyperparameter können die Modellarchitektur (Schichten, Neuronen, Dropout), die Trainingsparameter (Batchgröße, Lernrate, Epochen) oder die Optimierungsfunktion betreffen. Die Trainingspipeline wird mit variierenden Parametern iterativ durchlaufen, während die Ergebnisse dokumentiert werden, sodass anschließend das beste Modell ausgewählt werden kann [28, S. 113–121].

## 2.3. Herausforderungen beim Einsatz von ML

### 2.3.1. Technische Schulden in ML-Systemen

Immer mehr Unternehmen arbeiten an der Integration von KI in ihre Produkte und Dienstleistungen, um dadurch innovativer, effizienter und nachhaltiger zu werden [1]. Diese Entwicklung bringt viele Chancen mit sich, birgt jedoch auch eine Reihe von Herausforderungen, die langfristig oft zum Scheitern der Projekte führen [3]. Die Entwicklung von ML-Modellen unterscheidet sich in vielen Aspekten von der klassischen Softwareentwicklung. Deshalb ist es wichtig, die spezifischen Anforderungen von ML-Projekten zu verstehen und den Entwicklungsprozess von Beginn an auf diese auszurichten.

Viele ML-Projekte scheitern nicht an der Entwicklung der Modelle selbst, sondern an deren langfristigem und wirtschaftlich erfolgreichem Einsatz. Während die Entwicklung von ML-Lösungen in kleinen Teams und die Bereitstellung dieser in einer Testumgebung oft ein vielversprechendes Proof-of-Concept darstellt, sind die Herausforderungen des Betriebes in einer Produktionsumgebung deutlich komplexer. Ähnlich wie beim DevOps-Konzept erfordern Entwicklung und Betrieb von ML-Modellen eine systematische und standardisierte Herangehensweise, die die Wartung und Weiterentwicklung der Komponenten ermöglicht und die Zuverlässigkeit und Skalierbarkeit des Systems gewährleistet [29]. Im Vergleich zu herkömmlicher Software können bei der Entwicklung von ML-Modellen - etwa durch die eingeschränkte Nachvollziehbarkeit von Änderungen in Code, Daten und Modellen auf die Ergebnisse - weitere Schulden entstehen, die sich erst während des Betriebs zeigen. Dieser „Hidden Technical Debt in Machine Learning Systems“ [5] führt zu einem unerwartet hohen Aufwand beim tatsächlichen Einsatz der Modelle und beeinträchtigt die langfristige Leistungsfähigkeit und Zuverlässigkeit des Systems. „Hidden Technical Debt“ bezieht sich auf technologische Schulden, die nicht unmittelbar sichtbar sind, jedoch durch komplexe Wechselwirkungen innerhalb von ML-Systemen entstehen. Dazu gehören unter anderem Abhängigkeiten zwischen Daten und Modellen, die bei Änderungen zu unerwartetem Verhalten führen können, oder die unzureichende Dokumentation von Experimenten, welche die Nachvollziehbarkeit von Entscheidungen erschwert. Die Identifikation und Vermeidung dieser Schulden ist entscheidend für den langfristigen Erfolg beim produktiven Einsatz von Machine Learning in Unternehmen [7, Kapitel 1].

### 2.3.2. Veränderungen in Daten und Modellen

Herkömmliche Softwareprodukte können nach erfolgreichem Test und Deployment über einen langen Zeitraum unverändert in der Produktionsumgebung betrieben werden und erfüllen ihre Aufgabe dabei mit gleichbleibender Qualität. Die Funktion eines bereitgestellten ML-Modells hängt in Bezug auf das ursprüngliche Projektziel hingegen von vielen Faktoren ab, die sich mit der Zeit ändern können. Dabei verschlechtern sich Leistung und Qualität des Modells langfristig, sodass es die Anforderungen gegebenenfalls nicht mehr erfüllt. Dieses Phänomen wird als Drift bezeichnet und kann verschiedene Ursachen haben [7, Kapitel 5]. Ändern sich die statistischen Eigenschaften eines Datensatzes mit der Zeit, spricht man von **Data Drift**. Durch die Verschiebung der unterliegenden Datenverteilung entstehen Diskrepanzen zwischen den Trainingsdaten und den realen Daten, was sich in einer Verschlechterung der Modellleistung äußert. Betrifft die Veränderung der Daten die Merkmale, spricht man von **Covariate Shift**. Bei Bilddaten können zum Beispiel Veränderungen von Helligkeit, Auflösung oder Farbraum eine Änderung der statistischen Eigenschaften der Bilddaten hervorrufen. Neben den Merkmalen können auch die Labels der Daten betroffen sein, was als **Label Shift** bezeichnet wird. Dies tritt auf, wenn sich die Häufigkeit bestimmter Labels in der Realität von ihrer Verteilung in den Trainingsdaten unterscheidet. Das können Labels sein, die am Randbereich liegen oder in den Trainingsdaten allgemein unterrepräsentiert sind. Zuletzt kann sich auch die Beziehung zwischen den Merkmalen und den Labels ändern, was als **Concept Drift** bezeichnet wird. Dabei können neue oder veränderte Umweltgegebenheiten, die in den Trainingsdaten nicht vorkamen, Merkmale aufweisen, die vom Modell nicht erkannt oder falsch interpretiert werden. Am Beispiel der Rückfahrkamera im Auto kann eine Änderung der Perspektive, etwa durch eine leicht geöffnete Kofferraumklappe, die Hindernisse im Bild verzerren und so die Beziehung zwischen Bilddaten und Labels verändern [6, S. 171–174][7, Kapitel 5].

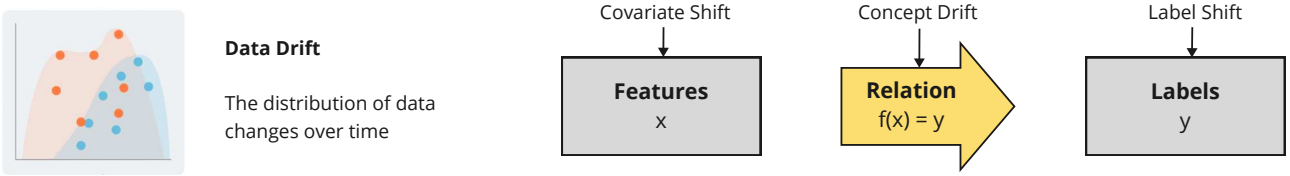



Abbildung 2.5.: Drift beim Einsatz von ML-Modellen [7]

Drift ist ein allgegenwärtiges Problem beim Betrieb von ML-Modellen über einen längeren Zeitraum oder in anderen Umgebungen. Um die Leistungsfähigkeit und Zuverlässigkeit des Systems zu erhalten, müssen ML-Modelle kontinuierlich überwacht und angepasst werden. Der dadurch entstehende Aufwand wird oft vernachlässigt und verursacht durch offene technische Schulden unerwartet hohe Kosten.

## 2.4. MLOps Kernkonzepte und Tools

### 2.4.1. Prinzipien und Komponenten im Überblick

Machine Learning Operations (MLOps) umfasst Tools und Prinzipien zur Unterstützung des gesamten Lebenszyklus von ML-Modellen. Dabei werden bewährte Praktiken aus der Softwareentwicklung und des IT-Betriebs auf ML-Projekte übertragen und um systematische Methoden für Datenverarbeitung, Training, Bereitstellung und Überwachung erweitert. Prozesse und Abläufe werden dokumentiert, standardisiert und weitgehend automatisiert. So können technische Schulden vermieden und ML-Systeme langfristig erfolgreich betrieben werden [7, Kapitel 1.04].

Für die praktische Realisierung dieser Anforderungen wurden sehr viele Tools und Frameworks entwickelt, die spezifische Funktionalitäten bereitstellen und in Kombination eingesetzt werden können. Aufgrund der großen Menge an Optionen mit eigenen Stärken und Schwächen stellt die Auswahl der richtigen Tools eine Herausforderung dar [30]. Dazu können die Konzepte von MLOps unterschiedlich interpretiert und angewendet werden, wodurch viele verschiedene Ansätze - jeweils mit bestimmten Schwerpunkten - in der Literatur zu finden sind. Somit gibt es auch nicht nur eine einzige und universelle MLOps-Lösung, die als universeller Baustein für jedes Projekt geeignet ist.

Aus den verschiedenen Ansätzen kristallisieren sich jedoch **Prinzipien** heraus, die von einer MLOps-Lösung unterstützt werden sollten [29]:

- **Versionierung:** Versionierung aller Daten und Komponenten des ML-Projekts
- **Reproduzierbarkeit:** Experimente müssen dokumentiert und reproduzierbar sein
- **Workflow Orchestrierung:** Zentrale Koordination und Visualisierung von Prozessen
- **Kollaboration:** Zusammenarbeit der Stakeholder an Daten, Modellen und Code
- **CI/CD Automation:** Automatisierung von Integrations- und Bereitstellungsprozessen
- **Metadata Logging:** Zwingende Speicherung von Metadaten bei jedem Ablauf
- **Continous Monitoring:** Überwachung von bereitgestellten Modellen und Infrastruktur
- **Feedback Loops:** Automatisierte Ereignisse, die auf Überwachungsergebnisse reagieren
- **Kontinuierliches Training:** Kontinuierliche Anpassung von Modellen an neue Daten

Zur Umsetzung dieser Methoden im ML-Projekt werden einige technische **Komponenten** benötigt, die bestimmte Funktionalitäten bereitstellen [7][29]:

- **Source Code Repository:** Zentrale Verwaltung und Dokumentation von Quellcode
- **Experiment Tracking:** Zentrale Dokumentation von Trainingsdurchläufen
- **Model Registry:** Zentrale Verwaltung und Versionierung von trainierten Modellen
- **Feature Store:** Zentrale Verwaltung und Versionierung von Datensätzen
- **Metadata Store:** Zentrale Ablage von Metadaten
- **CI/CD Tools:** Tools zur Implementierung von CI/CD-Pipelines
- **Workflow Orchestration Tool:** Tool für Prozessmanagement und -visualisierung
- **Inference Pipeline:** Modellbetrieb, Sammeln und Evaluieren von Daten in Echtzeit
- **Model Training Pipeline:** Kontinuierliche Anpassung von Modellen
- **Monitoring Tool:** Aufbereitung und grafische Darstellung der Metadaten

Die benötigten Funktionalitäten müssen nicht von Grund auf neu entwickelt werden, sondern lassen sich durch die Kombination bestehender Tools und Frameworks erreichen. Einige Bestandteile wie das Source Code Repository oder die CI/CD Tools können dabei aus dem DevOps-Kontext übernommen werden. Im Folgenden wird eine Auswahl weiterer Tools vorgestellt, die zur Realisierung der eigenen MLOps-Lösung an dem ausgewählten praktischen Beispiel beitragen. Diese wurden anhand konkreter Anforderungen an Funktionalität und Kompatibilität ausgewählt. Das Zusammenspiel der Komponenten wird in Kapitel 3 ausführlich beschrieben.

### 2.4.2. Experiment Tracking und Modellverwaltung mit MLFlow

Während des Trainings von ML-Modellen fallen große Datenmengen an, die mit dem Ziel der Reproduzierbarkeit und Nachvollziehbarkeit dokumentiert werden müssen. Durch Analyse der gesammelten Informationen kann die Auswirkung gewählter Daten und Parameter auf den Trainingserfolg festgestellt und die Modellqualität verbessert werden. Das Tool *MLFlow* bietet hierfür eine zentrale Plattform, die Experimente aufzeichnet, Ergebnisse visualisiert und Modelle versioniert. Die Kommunikation erfolgt über eine Programmierschnittstelle (API) und wahlweise über eine grafische Benutzeroberfläche (UI). Für die Integration in den Modellcode wird ein Python-Paket genutzt, das mit nur wenigen Codezeilen die Verbindung mit der

MLFlow-API herstellt und Daten wie Trainingsparameter, Historie und Artefakte übermittelt. Für die Ablage der Daten greift MLFlow auf einen Bucket und eine SQL[1]-Datenbank zurück. Im Bucket werden die Artefakte (Modelle, Metriken oder Visualisierungen) gespeichert, während die relationale Datenbank die Metadaten und Historie der Experimente enthält. Jeder Trainingsdurchlauf mit bestimmten Parametern wird dabei als Run abgelegt. Mehrere Runs des Trainings eines bestimmten Modells werden als Experiment zusammengefasst, sodass alle Ergebnisse in der UI gebündelt visualisiert und damit vergleichbar sind. Anschließend werden die Modelle mit den besten Evaluationsergebnissen ausgewählt und im Modellregister gespeichert. Dieses versioniert die Modelle und ermöglicht deren Nutzung in der Inference-Pipeline, also der Umgebung, in der das Modell zur Laufzeit genutzt wird, um neue Daten zu analysieren und Ergebnisse zu liefern. Das Laden eines bestimmten Modells aus dem Modellregister erfolgt dort über die API und ist durch das Python-Paket sehr einfach möglich. MLFlow ist quelloffen, kostenfrei und wird von einer breiten Entwickler-Community getragen [28, S. 125–128].

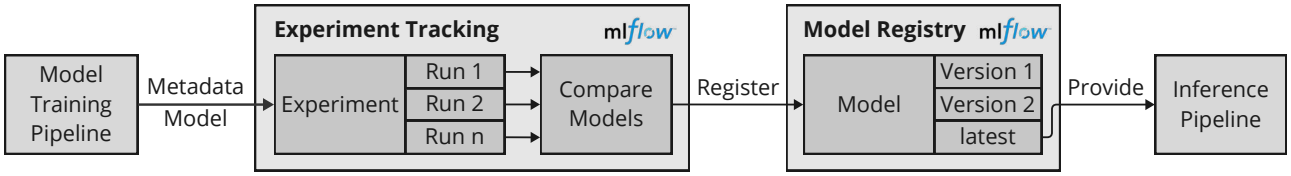



Abbildung 2.6.: Anwendung von Experiment Tracking und Model Registry

### 2.4.3. Überwachen des Systems mit InfluxDB und Grafana

Entscheidend für jedes MLOps-System ist das Monitoring der bereitgestellten Modelle und der Infrastruktur, damit sich Drift oder Ausfälle frühzeitig erkennen und beheben lassen. Dafür müssen möglichst viele aussagekräftige Metriken kontinuierlich aufgezeichnet und ausgewertet werden. Für das Speichern von Daten in Abhängigkeit der Zeit eignen sich Zeitreihendatenbanken wie *InfluxDB*, die Daten in Zeitreihen speichern, um effiziente Abfragen, Filter und Transformationen zu ermöglichen. Die Kommunikation mit der Datenbank erfolgt über eine API und wahlweise über die enthaltene UI. InfluxDB ist horizontal skalierbar, quelloffen und bietet eine umfangreiche Integration mit anderen Tools. Um Daten während des Betriebs zu sammeln, kann in der Inference-Pipeline das zugehörige Python-Paket genutzt werden. Mit dem Plugin *telegraf* können auch Metriken der Infrastruktur aufgezeichnet werden [31].

Neben dem Speichern relevanter Zeitreihendaten ist auch die aussagekräftige Visualisierung entscheidend. Das Open-Source-Tool *Grafana* bietet hierfür eine umfangreiche Plattform, mit der Daten aus InfluxDB und anderen Quellen über einen Adapter abgerufen und in Dashboards grafisch darstellt werden können. Ein Dashboard enthält verschiedene Panels für die Visualisie-

[1]SQL ist eine Sprache zur Verwaltung und Abfrage von Daten in relationalen Datenbanken

rung bestimmter Metriken. Diese können individuell konfiguriert werden und bieten eine große Auswahl an Darstellungsformen, darunter Diagramme, Tabellen, Text und Heatmaps [32].

Zusätzlich zur Visualisierung in Dashboards können in Grafana auch Alerts konfiguriert werden, die eine bestimmte Metrik überwachen und auslösen, wenn ein, durch oberen und unteren Grenzwert definierter, Bereich für eine bestimmte Zeit verlassen wird. Die beim Auslösen eines Alerts ausgeführten Aktionen sind individuell konfigurierbar, sodass beispielsweise eine E-Mail an einen Stakeholder gesendet oder eine API per HTTP-Anfrage aufgerufen werden kann. Dies ermöglicht das automatische Neutraining von Modellen, wenn deren Performance durch Drift beeinträchtigt ist.

### 2.4.4. Orchestrierung von Daten und Abläufen mit Dagster

Während der Entwicklung und Bereitstellung von ML-Modellen spielt das Verarbeiten von Daten eine zentrale Rolle. Um technische Schulden zu vermeiden, müssen alle verwendeten und generierten Daten sinnvoll organisiert werden. Dazu gehören die Standardisierung von Abläufen, die Visualisierung des Datenflusses, das geordnete Speichern der Daten und die weitgehende Automatisierung von Prozessen. Ein typischer Anwendungsfall im ML-Kontext ist das Modelltraining, bei dem Daten aus einer Quelle geladen, vorverarbeitet sowie in Trainings- und Testdaten aufgeteilt werden. Anschließend wird das Modell definiert, trainiert, evaluiert und in einem Modellregister abgelegt. Solche Pipelines bestehen aus festgelegten Prozessschritten, die aufeinander aufbauen und voneinander abhängen.

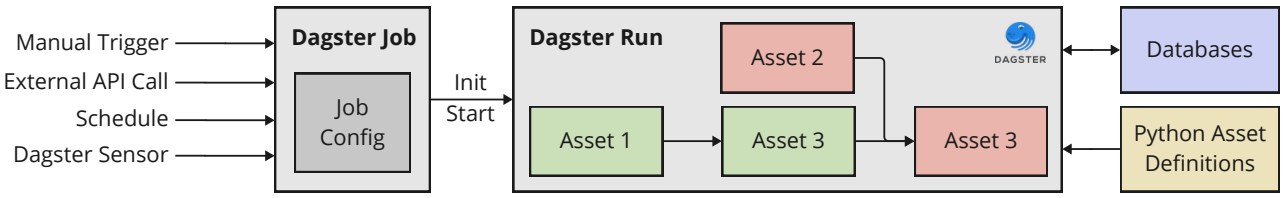



Abbildung 2.7.: Orchestrierung von Prozessen mit Dagster

Um diese Anforderungen zu erfüllen, können Orchestrierungs-Tools wie *Dagster* eingesetzt werden. Dagster ermöglicht die Definition der einzelnen Prozessschritte als sogenannte Software-defined Assets. Jedes Asset repräsentiert ein bestimmtes Datenobjekt, das anhand seines Codes und seiner Konfiguration generiert und persistent gespeichert wird. Durch die Definition der benötigten Eingangsdaten ergeben sich Abhängigkeiten zwischen den Assets, wodurch ein Asset-Graph entsteht. Dieser Graph wird in der Dagster-UI mit dem aktuellen Zustand der einzelnen Assets visualisiert. Die grafische Darstellung erleichtert das Verständnis komplexer Prozesse und ermöglicht die Überwachung des Datenflusses in Echtzeit. Die Ausführung bestimmter Assets wird als Job bezeichnet. Jobs können manuell über die UI, von externen Quel-

len über die API, nach Zeitplan oder durch Dagster-Sensoren ausgelöst werden. Sensoren sind Python-Funktionen, die bestimmte Bedingungen überwachen und bei deren Erfüllung die Ausführung des Jobs auslösen. Jede Ausführung eines Jobs wird als Run protokolliert. Ein Run enthält wichtige Metadaten wie Start- und Endzeit, Status sowie erzeugte Artefakte, wodurch die Nachvollziehbarkeit der Abläufe sichergestellt wird. Neben diesen grundlegenden Funktionen bietet Dagster erweiterte Lösungen, um spezifischen Anforderungen gerecht zu werden. Die Konfiguration aller Dagster-Komponenten erfolgt in Python, sodass bestehender modularer Modellcode nahtlos in Assets integriert und direkt in der Pipeline verwendet werden kann. Dies ermöglicht die enge Verzahnung von Modellcode und automatisierten Abläufen [33][34].

### 2.4.5. Bereitstellen von individuellen Diensten mit Flask

Nicht jede benötigte Funktionalität kann mit den bestehenden Tools vollständig realisiert werden. Deshalb muss auch die Möglichkeit zur Entwicklung eigener Komponenten und deren Integration in das Gesamtsystem bestehen. Als universelle und flexible Schnittstelle bietet sich die Verwendung einer RESTful[2] API an, mit der Daten über das Netzwerk ausgetauscht werden können. Der API-Endpunkt wird durch einen Service bereitgestellt, der HTTP-Anfragen von Clients entgegennimmt. Diese werden an definierte Routen gesendet, die jeweils spezifische Anfragen verarbeiten und eine Antwort zurückgeben. Antworten enthalten beliebige Daten in standardisiertem Format und mindestens einen Response-Code, der dem Client Auskunft über den Erfolg oder Misserfolg der Anfrage gibt [6, S. 178–181].

Um bestehenden Python-Code bei Anfragen an die API auszuführen, wird das Framework *Flask* eingesetzt. Mit Flask können die benötigten Routen direkt in Python definiert und die Anfragen an die entsprechenden Funktionen weitergeleitet werden. Die Antwort an den Client wird dann über den Rückgabewert der jeweiligen Funktion, zum Beispiel als JavaScript Object Notation (JSON)-Objekt, gesendet. Der Python-Code kann dann als Flask-App ausgeführt und als API über einen beliebigen Port bereitstellt werden [35].

Durch den korrekten Einsatz der Tools MLFlow, Dagster und InfluxDB sowie individuellen Flask-Apps werden anfallende Daten strukturiert, dokumentiert und organisiert. Dadurch kann auf den Einsatz eines zusätzlichen Versionskontrollsystems zur Verwaltung der Datensätze verzichtet werden. Der konkrete Einsatz dieser Tools im Gesamtsystem zum Erreichen der Anforderungen an die Nachvollziehbarkeit wird in Kapitel 3 detailliert beschrieben.

[2] API folgt den Representational State Transfer Architekturbeschränkungen und Vorgaben

### 2.4.6. Methoden zur automatisierten Datenannotation

Die Qualität eines ML-Modells hängt maßgeblich von der Qualität der Trainingsdaten ab. Die Annotation von Daten, also das Hinzufügen von Labels, ist ein notwendiger Schritt, der häufig manuell mit hohem Aufwand durchgeführt wird [36]. Im Rahmen einer ML-Anwendung fallen ständig große Datenmengen an, sodass alternative Methoden zur Annotation erforderlich sind.

Ein vielversprechender Ansatz ist das **Echtzeitlabelling**, bei dem die Nutzer selbst - direkt während der Datenerfassung - Labels hinzufügen. Dies kann durch Smartphone-Apps erfolgen, die kontinuierlich Sensordaten aufzeichnen, während der Benutzer seine aktuelle Aktivität selbst meldet. Diese Methode ermöglicht eine schnelle und effiziente Kennzeichnung von Daten, hängt jedoch qualitativ stark von der Zuverlässigkeit der Nutzer ab. [37]

**Active Learning** kann helfen, den Bedarf an manueller Datenannotation zu reduzieren, indem das Modell die zu beschriftenden Datenpunkte sortiert und gezielt die Daten auswählt, von denen es am meisten lernen kann. Als Kriterium dient die Unsicherheit des Modells bei der Klassifizierung des Datenpunktes. Je unsicherer die Vorhersage, desto wahrscheinlicher ist ein Fehler. Das Labeln von Datenpunkten mit hoher Unsicherheit hat deshalb den größten positiven Einfluss auf die Modellgenauigkeit. Dadurch müssen nur kleine Mengen unbeschrifteter Daten gelabelt werden, was den Aufwand verringert, jedoch nicht gänzlich vermeidet [38].

Im praktischen Beispiel dieser Arbeit findet das Annotieren der Bilddaten durch einen Laser-Abstandssensor statt. Dies ist nur möglich, da das Modell in der konkreten Anwendung die Funktion des Sensors abbilden soll. Voraussetzung ist dabei, dass die eingesetzte **Referenzsensorik** exakt die für das Label benötigten Informationen liefert und die zeitliche und räumliche Auflösung der Daten mit den Anforderungen des Modells übereinstimmt.

### 2.4.7. Systematisches Deployment von Modellen

Im Betrieb einer ML-Anwendung ist es essenziell, die in der produktiven Umgebung bereitgestellten Modelle kontinuierlich zu überwachen und bei Bedarf anzupassen. Nach dem Training eines neuen Modells muss dieses in die Produktionsumgebung überführt und dafür das bisherige Modell ausgetauscht werden. Hierfür kommen verschiedene Deployment-Strategien infrage, die systematisch festlegen, wie der Wechsel zwischen den Modellen erfolgt.

Im einfachsten Fall, dem **Direct Deployment**, wird das bisherige Modell direkt durch das neue Modell ersetzt. Diese Strategie ist leicht umzusetzen, birgt jedoch immer das Risiko, dass sich das neue Modell im Produktivsystem unerwartet schlechter verhält als das bisherige. Um dieses Risiko zu vermeiden, kann die **Shadow Deployment**-Strategie eingesetzt werden. Dabei wird das neue Modell parallel zum bisherigen Modell in der Produktionsumgebung betrieben, wobei

die Ausgaben des neuen Modells nicht an den Endnutzer weitergegeben, sondern ausschließlich zur Analyse der Performance gespeichert werden. Das Deployment des neuen Modells erfolgt erst dann, wenn die einwandfreie Funktion anhand der gesammelten Daten bestätigt wurde. Sind vereinzelte Fehler des Modells tolerierbar, kann das **Canary Deployment** eine sinnvolle Strategie sein. Dabei wird das neue Modell nur für einen kleinen Teil der Nutzer bereitgestellt, bevor es schrittweise auf alle Nutzer ausgeweitet wird. Dies ermöglicht die Überprüfung der Funktion anhand realer Daten und dem Feedback der Nutzer. Je kleiner die Gruppe der Nutzer des neuen Modells ist, desto länger dauert es, bis eine ausreichend große Anzahl an belastbaren Daten gesammelt wurde. Die **A/B-Testing** Strategie funktioniert ähnlich, jedoch wird das neue Modell direkt für die Hälfte der Nutzer bereitgestellt. Die Daten beider Modelle werden gesammelt und statistisch ausgewertet, um die Performance der Modelle in der Produktivumgebung zu vergleichen. Da eine Fehlfunktion des neuen Modells nicht auszuschließen ist, sind sowohl das Canary Deployment als auch das A/B-Testing nicht für alle Anwendungen geeignet. Zuletzt kann die **Multi-armed bandit** Strategie eingesetzt werden, die eingehende Anfragen dynamisch an die verschiedenen Modelle verteilt. Dabei wird die Performance der Modelle kontinuierlich überwacht und die Verteilung der Anfragen entsprechend angepasst. Dies führt dazu, dass über einen längeren Zeitraum hinweg das beste Modell bevorzugt wird. Auch hier können Fehler des neuen Modells zu unerwünschten Ergebnissen führen, jedoch wird der mittlere Fehler durch die dynamische Anpassung minimiert.

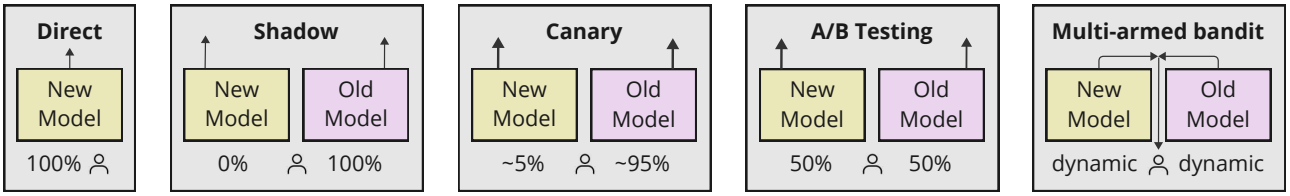



Abbildung 2.8.: Überblick über mögliche Deploymentstrategien

Die Wahl einer Deployment-Strategie hängt von den konkreten Anforderungen der Anwendung ab. Je kritischer die Anwendung, desto vorsichtiger sollte das Deployment erfolgen. Die Wahl der Strategie muss sorgfältig abgewogen und dokumentiert werden, sodass die Anforderungen an die produktive ML-Anwendung jederzeit erfüllt sind [7, Kapitel 5].

## 2.5. Infrastruktur

Die erfolgreiche Implementierung von MLOps hängt nicht nur von den Prinzipien und Tools ab, sondern erfordert auch eine robuste Infrastruktur. Diese bildet die Grundlage für die effiziente Verarbeitung von Daten, das Modelltraining und deren zuverlässige Bereitstellung in produktiven Umgebungen. Einzelne Komponenten des MLOps-Gesamtsystems werden dabei voneinander unabhängig als Microservice bereitgestellt. Microservices sind eigenständige Anwendungen, die über definierte Schnittstellen kommunizieren. Durch diese Modularität können einzelne Dienste unabhängig entwickelt, getestet und bereitgestellt werden, was sowohl die Entwicklungszeit verkürzt als auch die Skalierbarkeit und Wartbarkeit des Gesamtsystems verbessert [6, S. 179–181][39].

### 2.5.1. Containerisierung und Docker

Die Bereitstellung von Microservices in einer skalierbaren und portablen Weise erfordert geeignete Technologien, um Entwicklungs- und Produktionsumgebungen zu standardisieren. Hierfür hat sich die Containerisierung etabliert, die es ermöglicht, Anwendungen und ihre Abhängigkeiten jeweils in einen isolierten Container zu verpacken. Diese Container enthalten alle notwendigen Bibliotheken, Pakete und Konfigurationen, sodass der enthaltene Microservice zuverlässig auf jeder beliebigen Infrastruktur ausgeführt werden kann [6, S. 103–107]. Als Werkzeug zur Entwicklung von Containern und einfachen Containersystemen hat sich *Docker* etabliert. Docker ist ein quelloffenes Tool, das die Virtualisierung auf Betriebssystemebene bereitstellt und dadurch die Ausführung und Verwaltung containerisierter Anwendungen ermöglicht. Dafür wird auf dem Host-System die Docker-Engine ausgeführt, die Container erstellen, starten, stoppen und verwalten kann. Die Definition eines Docker-Containers erfolgt in einem `Dockerfile`, das alle notwendigen Schritte zur Erstellung und zum Start des Containers beschreibt. Die Container können dann kompiliert, als Docker-Image gespeichert, im Docker-Registry bereitgestellt und gestartet werden. Das Speichern persistenter Daten erfolgt in Laufwerken (Volumes), die sich getrennt vom Container auf dem Host-System befinden [6, S. 103–107][40].

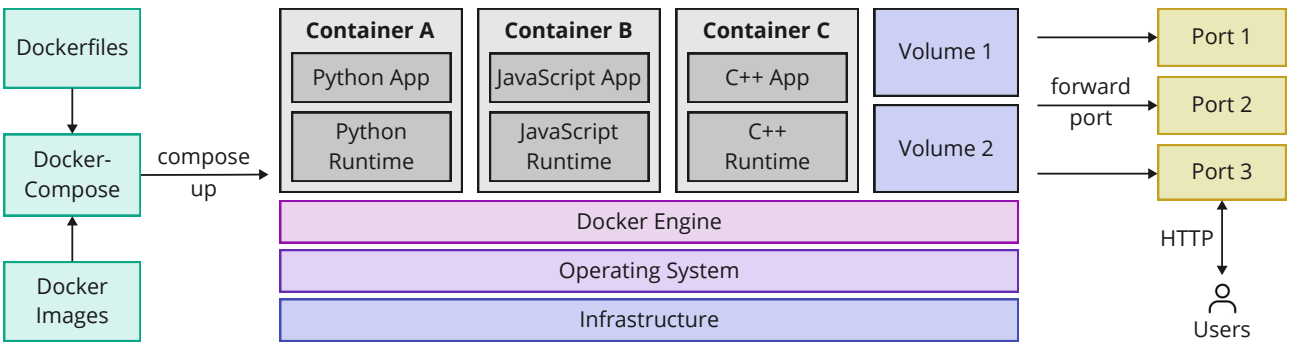



Abbildung 2.9.: Containerisierung von Anwendungen mit Docker

Ein Gesamtsystem (siehe Abbildung 2.9) besteht aus verschiedenen Microservices, die in Containern verpackt sind und über definierte Schnittstellen miteinander kommunizieren. Zur Definition und Ausführung einer solchen Multi-Container-Anwendung wird das Tool *Docker Compose* eingesetzt. Dafür werden die benötigten Container, Abhängigkeiten, Schnittstellen und Volumes in der Datei `docker-compose.yml` definiert. Die Container werden gemeinsam gestartet, gestoppt und verwaltet, sodass auch die Bereitstellung umfangreicher Systeme mit vielen Containern und Abhängigkeiten möglich ist [41].

Bei hohen Anforderungen an Skalierbarkeit und Verfügbarkeit können die Container in einem Orchestrierungs-Tool wie *Kubernetes* verwaltet werden. Kubernetes ist ein quelloffenes Tool, das die Verwaltung von Containern in einem Cluster ermöglicht. Dabei werden die Container auf verschiedene Knoten verteilt, überwacht und bei Ausfällen automatisch neu gestartet. Kubernetes stellt so sicher, dass die Anwendung jederzeit verfügbar ist und die Ressourcen effizient genutzt werden [42].

### 2.5.2. Cloud Computing

Während der Entwicklung kann das Gesamtsystem lokal auf einem Entwicklungsrechner oder in einer Testumgebung bereitgestellt werden. Für den produktiven Einsatz ist jedoch eine skalierbare und zuverlässige Infrastruktur notwendig, die früher von den Unternehmen selbst betrieben wurde. Dafür müssen physische Server und weitere Infrastruktur bereitgestellt werden, welche die hohen Anforderungen an Verfügbarkeit, Skalierbarkeit und Sicherheit erfüllen. Der Aufbau eigener Infrastruktur ist zeit- und kostenintensiv, da Beschaffung, Installation und Wartung erforderlich sind und freie Kapazitäten vorgehalten werden müssen. Cloud-Computing bietet hierfür eine kostengünstige und flexible Lösung, indem die benötigten Ressourcen von einem Dienstleister verwaltet und nach Verbrauch abgerechnet werden. Abbildung 2.10 veranschaulicht diesen Unterschied: Selbst wenn die Anwendung im Moment nur zwei Server beansprucht, müssen für den Fall von Lastspitzen alle sechs Server durchgehend selbst betrieben werden. In der Cloud hingegen werden nur die zwei tatsächlich genutzten Server abgerechnet, obwohl weitere Kapazitäten bereitstehen. Cloud-Ressourcen sind sofort verfügbar, jederzeit skalierbar und erfüllen höchste Sicherheitsstandards. Je nach gewünschtem Grad der Kontrolle kann zwischen verschiedenen Dienstleistungsmodellen gewählt werden. **Infrastructure as a Service** stellt die grundlegende Infrastruktur wie Rechenleistung, Speicher und Netzwerk in Form einer virtuellen Maschine bereit und ermöglicht die individuelle Installation und Konfiguration von Betriebssystem und Anwendungen. Bei **Platform as a Service** wird eine Plattform wie Kubernetes direkt bereitgestellt, ohne dass die zugrunde liegende Infrastruktur sichtbar ist. Dies stellt eine große Zeitersparnis dar, da Konfiguration und Wartung der Infrastruktur entfallen. Vollständige Anwendungen können als **Software as a Service** genutzt werden [43].

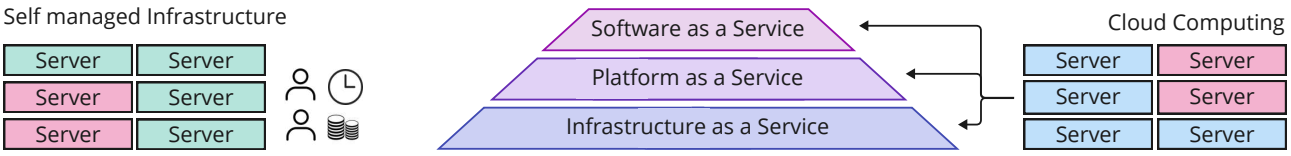



Abbildung 2.10.: Traditionelle Infrastruktur vs. Cloud Computing, Schichtenmodell

Wurde das gesamte System als Multi-Container-Anwendung entwickelt, kann es unabhängig von der Infrastruktur ausgeführt und so nach Belieben lokal oder über eine *Platform as a Service*-Dienstleistung cloudbasiert bereitgestellt werden.

## 2.6. MLOps-Lösungen führender Anbieter

Der Cloud Computing Markt wird von wenigen großen Anbietern dominiert, die eigene Komplettpakete für die Entwicklung und Bereitstellung von ML-Modellen anbieten. Diese Machine Learning Plattformen umfassen Tools und Dienstleistungen zur Datenverarbeitung, Modellentwicklung, Bereitstellung und Überwachung. Die Anbieter unterscheiden sich dabei in der Art und Weise, wie die einzelnen Komponenten umgesetzt sind und welche zusätzlichen Dienste angeboten werden. Die bekanntesten Anbieter sind *Amazon Web Services (AWS)* mit *Amazon Sagemaker*, *Microsoft Azure* mit *Azure Machine Learning* und *Google Cloud* mit *Google AI Platform*. Alle Plattformen beinhalten Tools und Dienste für die Datenverarbeitung, Modellentwicklung, Bereitstellung und Überwachung. Die Grundstruktur aller Lösungen ist sehr ähnlich, jedoch gibt es Unterschiede in der Umsetzung und den angebotenen Funktionen.

**Amazon SageMaker**, Teil des AWS-Ökosystems, hebt sich durch seine enge Integration mit anderen AWS-Diensten hervor. Es bietet die umfangreiche Entwicklungsumgebung *SageMaker Studio*, eine Vielzahl integrierter Algorithmen und Funktionen wie den *Sagemaker Model Monitor* zur kontinuierlichen Überwachung oder *SageMaker Pipelines* für automatisierte Workflows. Für die Bereitstellung bietet *SageMaker* individuelle Endpunkte, die Modelle in einer skalierbaren und global hochverfügbaren Umgebung ausführen. Mit *Spot-Instances* können durch die effiziente Nutzung von Ressourcen zudem die Kosten beim Training gesenkt werden. Die Plattform bietet mit ihren Services die Nutzung von robusten MLOps-Tools und Praktiken an und ermöglicht die Integration in bestehende AWS-Infrastrukturen. *SageMaker* ist besonders für Unternehmen attraktiv, die bereits AWS-Dienste nutzen und eine umfassende und integrierte Lösung für die Entwicklung und Bereitstellung von ML-Modellen suchen.

**Microsoft Azure Machine Learning** verfolgt mit *Azure ML Studio* einen ähnlichen Ansatz, bietet jedoch mehr Anpassungsmöglichkeiten, beispielsweise durch benutzerdefinierte Docker-Container. Durch den starken Fokus auf AutoML können die Entwicklungszeiten verkürzt und die Effizienz gesteigert werden. Mit Funktionen wie dem *Azure ML Designer* sind auch nicht-technische Nutzer in der Lage, Modelle und Pipelines per Drag-and-Drop zu erstellen. Die enge Integration im Azure-Ökosystem ermöglicht die Nutzung von *Azure Kubernetes Service* und weiteren Diensten. Azure ML enthält ebenfalls Services, die den Einsatz von Modellen im IoT-Umfeld oder in Edge-Computing-Szenarien erleichtern. Die Plattform ist besonders für Unternehmen attraktiv, die bereits Azure-Dienste nutzen und eine höhere Anpassbarkeit und Flexibilität benötigen.

**Google AI Platform** hingegen bietet eine starke Integration mit den KI-Diensten der Google Cloud, darunter *TensorFlow Extended* und *Vertex AI*. Die Plattform zeichnet sich durch eine einfache Handhabung, den Zugang zu cutting-edge Technologien wie TPUs und die tiefe Integration in Googles Datenanalyse Tools aus. Google bietet zudem erweiterte AutoML Funktionen und eine Vielzahl vorgefertigter Modelle für spezifische Anwendungen. Für die Entwicklung von individuellen Modellen im Sprachverarbeitungs- oder Bilderkennungsbereich bietet Google AI Platform leicht zugängliche Dienste, die durch hoch entwickelte Algorithmen die Entwicklung von leistungsstarken Modellen erleichtern. Die Plattform ist besonders für Unternehmen attraktiv, die auf die Google Cloud setzen und eine einfache und effiziente Lösung für die Entwicklung und Bereitstellung von ML-Modellen suchen [44].

Cloudbasierte MLOps-Lösungen bieten eine Vielzahl von Vorteilen, darunter einfache Skalierbarkeit, globale Verfügbarkeit und integrierte Sicherheitsstandards. Da die Umgebungen bereits vorkonfiguriert, visuell zugänglich und durch die hohe Nutzerzahl gut dokumentiert sind, können Entwickler in kürzester Zeit Modelle bereitstellen und müssen sich nicht um die Infrastruktur oder die Auswahl und Installation einzelner Dienste kümmern. Durch die Nutzung von Cloud-Ressourcen lassen sich zudem die Kosten reduzieren. Nachteile bestehen jedoch in einem durch den Anbieter festgelegten und teils eingeschränkten Angebot, der geringen Transparenz einiger Dienste und fehlender Anpassungs- und Kontrollmöglichkeiten. Die Nutzung der Komplettlösung eines Dienstleisters schränkt zudem die Portabilität der entwickelten Anwendungen ein und führt zu einer Abhängigkeit von den jeweiligen Kostenstrukturen sowie zu einer potenziellen Vendor-Lock-in-Problematik. Zuletzt können Datenschutzvorgaben die Nutzung von Diensten bestimmter Unternehmen ausschließen [45].

## 2.7. Gründe für einen individuellen Ansatz

Vor Beginn der Entwicklung eines eigenen Systems zur Realisierung der MLOps Prinzipien und Methoden sollte die Notwendigkeit eines solchen Systems, besonders in Bezug auf die ausgereiften Komplettlösungen der Cloud-Dienstleister, kritisch hinterfragt werden. Die Entwicklung einer individuellen Lösung erfordert einen längeren Vorlauf und benötigt zusätzliche Investitionen. Ohne die tiefe Integration in die bestehende Infrastruktur eines Dienstleisters müssen benötigte Komponenten selbst ausgewählt, angepasst und gemeinsam bereitgestellt werden. Vor allem die hohe IT-Sicherheit, Skalierbarkeit und weltweite Verfügbarkeit der Cloud-Dienstleister sind mit einer Eigenentwicklung schwer zu erreichen.

Trotz der Vorteile einer Cloud-Komplettlösung spricht auch vieles für die Entwicklung einer maßgeschneiderten Eigenlösung. Diese bietet die beste Anpassungsfähigkeit an spezifische Anforderungen und hat zugleich den entscheidenden Vorteil der vollständigen Unabhängigkeit von externen Dienstleistern. Die einzelnen Komponenten des Systems können individuell ausgewählt werden, um die Umgebung besser in den Software-Stack der Entwicklerteams zu integrieren. Die Nutzung von Open-Source-Software erhöht die Transparenz und erlaubt die Anpassung einzelner Komponenten an besondere Anforderungen. Dank der Plattformunabhängigkeit lässt sich das System während der Entwicklung kostengünstig und ohne regulatorische Bedenken lokal betreiben und später auf beliebiger Infrastruktur bereitstellen. Damit kann das System selbst gehostet oder in einer Cloud-Umgebung betrieben werden, die den Vorgaben an Datenschutz und Sicherheit entspricht. Hohe Anforderungen an Skalierbarkeit und Verfügbarkeit können durch die individuelle Integration einzelner Cloud-Lösungen in das Gesamtsystem erreicht werden. Schließlich fördert die Eigenentwicklung ein tiefes Verständnis für die Funktionsweise des Systems sowie die Relevanz und Realisierung von MLOps in der Praxis. Damit wird eine wertvolle Wissensbasis aufgebaut, die als Grundlage für die erfolgreiche Realisierung von ML-Projekten im Unternehmen dienen kann.

# 3. Realisierung des individuellen Ansatzes

Versteckte technische Schulden erschweren die langfristig erfolgreiche Umsetzung von ML-Projekten im Unternehmen, weshalb die konsequente Einhaltung der erarbeiteten MLOps-Prinzipien für die erfolgreiche Realisierung dieser Projekte von entscheidender Bedeutung ist. Zuerst fällt durch die notwendige Implementierung von Tools zur Realisierung der Prinzipien jedoch ein hoher zusätzlicher Aufwand an, der aufgrund seiner Komplexität und der benötigten Kompetenzen oft aufgeschoben wird. Um dies zu verhindern, ist es wichtig, die Anwendung der Prinzipien in der Praxis so leicht wie möglich zu gestalten. Dafür soll eine Lösung entwickelt werden, die bewährte Tools und Technologien zur Realisierung der MLOps-Prinzipien nutzt und als flexibles Gesamtsystem mühelos in ML-Projekte integriert werden kann. Nachfolgend wird das selbst entwickelte System vorgestellt und dessen Einsatz in ML-Projekten erläutert.

## 3.1. Systemarchitektur

Das entwickelte System besteht aus mehreren Komponenten, die in zwei Hauptbereiche eingeteilt werden können. Auf der einen Seite steht der zugrundeliegende Programmcode, dessen Änderung ausschließlich durch das aktive Handeln eines Entwicklers erfolgt. Auf der anderen Seite befinden sich die datenbasierten Komponenten, deren Verhalten dynamisch durch den Datenstrom beeinflusst wird. Dieser wesentliche Unterschied liegt im Verhalten der Komponenten, sodass vom statischen und vom dynamischen System gesprochen werden kann.

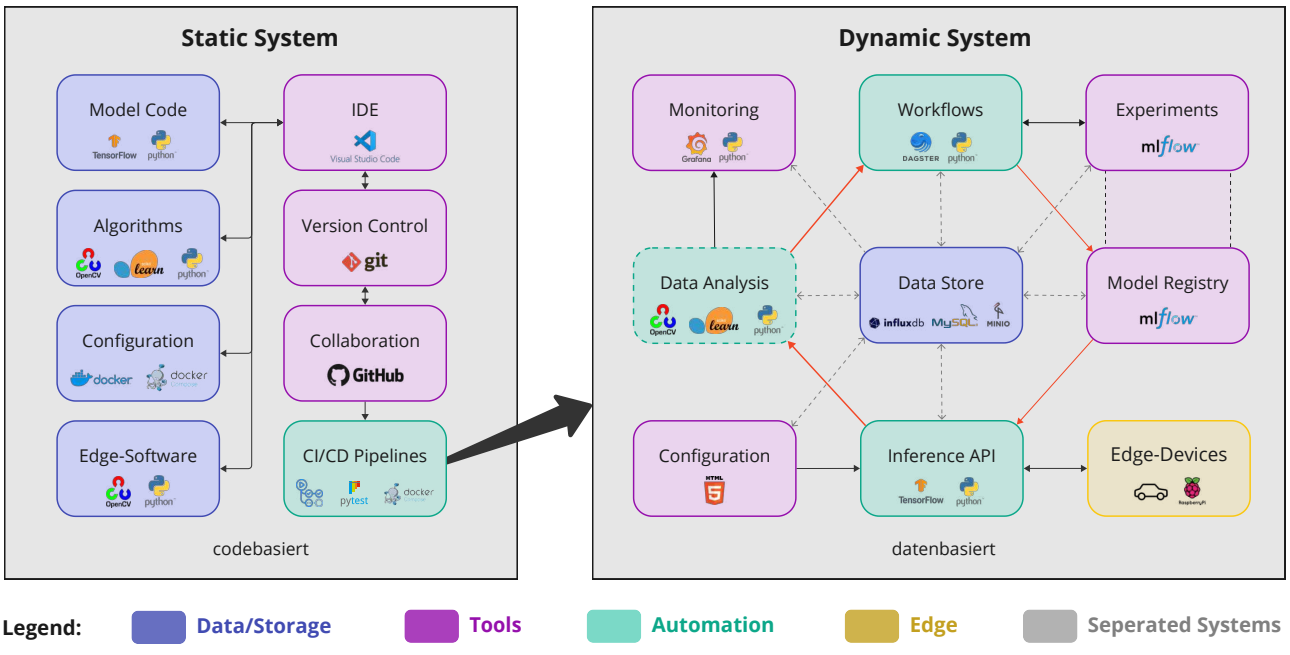



Abbildung 3.1.: Statisches und dynamisches System

Zum **statischen** Programmcode gehören nicht nur die konkrete Modellarchitektur, verschiedene Algorithmen zur Datenverarbeitung und die Client-Software, sondern auch die Konfiguration und Dokumentation aller Komponenten des Systems. Dies beinhaltet die Initialisierung der Datenbanken, das Bereitstellen der verwendeten Frameworks, das Kompilieren selbst entwickelter Dienste sowie die Definition von Schnittstellen, Prozessen, Pipelines und anderen Strukturen. All diese Komponenten werden lokal entwickelt, getestet und mit dem Versionsverwaltungssystem *git* organisiert. Ein Repository auf der Plattform GitHub dient als zentrale Instanz zur Zusammenarbeit und stellt durch CI-Pipelines die Einhaltung von Vorgaben bezüglich Standardisierung, Qualität und Funktionalität sicher. CD-Pipelines ermöglichen die automatisierte Bereitstellung der Software, sofern dies durch manuelle Änderungen am Main-Branch ausgelöst wird. Das statische System enthält folgenden Komponenten:

- **Modularer Modellcode:** Architektur, Datenvorverarbeitung, Training, Evaluation
- **Algorithmen:** Algorithmen zur Datenanalyse und Datenaufbereitung
- **Konfiguration:** Tools, eigene Container, Docker-Compose und Umgebungsvariablen
- **Edge-Software:** Client-Software zur Datenerfassung und Datenübertragung
- **IDE:** VSCode für die lokale Entwicklung an statischen Elementen
- **Versionskontrolle:** *Git* für die Verwaltung von Code und Konfiguration
- **Zusammenarbeit und Dokumentation:** GitHub als zentrale Plattform
- **CI/CD Pipelines:** Automatisierte Integration und Bereitstellung mit GitHub Actions

Das durch den Programmcode definierte **dynamische** System besteht aus vielen Softwarekomponenten, die verschiedene Aufgaben in der Datenverarbeitung und Datenverwaltung übernehmen. Dazu gehört der aktive Client, der Daten generiert und an das System sendet, um die Inference-Pipeline auszuführen und eine Vorhersage vom Modell zu erhalten. Gleichzeitig werden die empfangenen Daten in Datenbanken gesammelt und anschließend in mehreren Prozessen verarbeitet. Diese umfassen unter anderem Algorithmen zur statistischen Analyse, zur Erkennung von Veränderungen und Anomalien, die Konvertierung zu Datensätzen für das Modelltraining und die grafische Aufbereitung zur Darstellung in Dashboards. Eine Pipeline kann auf Basis dieser Daten automatisiert einen Modelltrainingsprozess starten, der das auf statischer Architektur basierende Modell mehrfach, jeweils mit variierenden Daten und Parametern, trainiert. Aus diesen trainierten Modellen wird anschließend das Beste ausgewählt und über ein Modellregister für den Client bereitgestellt. Bei allen Schritten werden ausführliche Protokolle

erstellt und zur Analyse durch Stakeholder grafisch aufbereitet, sodass Ergebnisse nachvollziehbar und reproduzierbar sind. Das dynamische System nutzt folgende Tools zur Realisierung der MLOps-Prinzipien:

- **Datenbanken:** MinIO, MySQL und Influx als spezifischen persistenten Speicher
- **Edge-Geräte:** Hardware mit Applikation, Schnittstelle zum Nutzer
- **Inference API:** Schnittstelle zum Edge-Gerät für die Modellvorhersage
- **Datenanalyse:** Algorithmen zur Aufbereitung und Analyse gesammelter Daten
- **Überwachung:** Grafische Darstellung der aufbereiteten Daten in Grafana-Dashboards
- **Workflows:** Orchestrierung von Abläufen und Automatisierung mit Dagster
- **Experiment Tracking:** Aufzeichnen und Visualisieren von Trainingsdaten mit MLFlow
- **Modellregister:** Verwaltung und Bereitstellung von Modellen mit MLFlow
- **Konfigurations UI:** Einfaches manuelles Anpassen der Inference-Pipeline

Nach diesem groben Überblick über die Systemarchitektur werden nachfolgend konkrete Abläufe im MLOps-Projekt mit den zur Realisierung verwendeten Technologien im Detail vorgestellt. Eine umfangreiche schematische Darstellung aller Komponenten, inklusive des Datenaustausches untereinander, ist im Anhang B, B.1 abgebildet.

## 3.2. MLOps-Lösung im Detail

Das durch den statischen Code vollständig definierte dynamische System kann sowohl lokal auf einem Entwicklungsrechner als auch auf einem Server oder in einer Cloud-Umgebung bereitgestellt werden. Aufgrund der Verwendung von Docker-Containern und Docker-Compose können alle Komponenten mit nur einem Befehl gestartet werden. Nach dem automatisierten Download der Container-Images und dem Kompilieren eigener Dienste ist das dynamische System vollständig einsatzbereit.

Zentraler Bestandteil ist der geschlossene Kreislauf, der in Abbildung 3.1 durch rote Pfeile dargestellt wird. Dieser ermöglicht die kontinuierliche Verbesserung des Modells anhand von realen und aktuellen Daten aus dem Betrieb in der Produktionsumgebung. Dadurch kann das Modell ständig an die sich ändernden Bedingungen angepasst werden, um Funktionalität und Qualität langfristig sicherzustellen. Damit der Aufwand beim Betrieb des Modells reduziert

wird, werden im Idealfall alle Abläufe automatisiert. Dies erfordert verschiedene Komponenten, die nicht nur die technische Umsetzung der Automatisierung unterstützen, sondern auch eine umfassende Überwachung durch Stakeholder ermöglichen. Daher müssen sämtliche Prozesse lückenlos protokolliert, die gesammelten Daten strukturiert und anschaulich visualisiert werden. Die folgenden Kapitel geben einen Einblick in die Funktionsweise dieses Kreislaufs und orientieren sich am Datenfluss durch das System.

### 3.2.1. Inference-API, Deployment und Konfiguration

Die Inference-API ist die zentrale Schnittstelle zu den Edge-Geräten. Diese senden eine Anfrage mit den Messdaten an die API und erwarten als Antwort eine Vorhersage des Modells. Nach erfolgreicher Autorisierung wird die Anfrage dafür mit der Inference Pipeline verarbeitet, in der die Daten verschiedene Schritte durchlaufen, die in Abbildung 3.2 schematisch dargestellt sind.

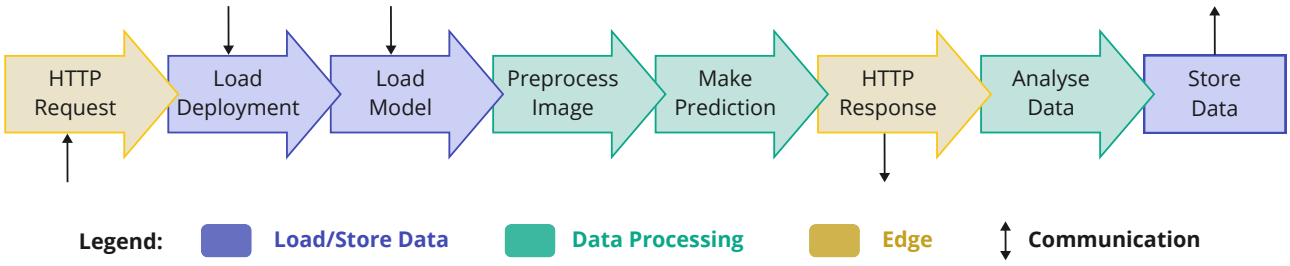



Abbildung 3.2.: Inference-Pipeline

Nach der Vollständigkeitsprüfung des empfangenen Datensatzes wird das zum Edge-Gerät zugeordnete Deployment aus der Datenbank geladen. Dieses enthält Informationen über das zugehörige Modell und die Gruppe, in der die gesammelten Daten gespeichert werden sollen. Dadurch können verschiedene Modelle für bestimmte Gruppen bereitgestellt und so, abhängig von der Deployment-Strategie, vorab unter bestimmten Bedingungen getestet werden. Um eine gute Organisation in der Datenbank zu gewährleisten, werden die Messdaten dem gewählten Deployment zugeordnet. Anschließend findet die Vorverarbeitung der Daten für das Modell statt, wozu mindestens das Konvertieren, Skalieren und Normalisieren der empfangenen Bilddatei gehört. Hiernach wird die Modellvorhersage durchgeführt und das Ergebnis direkt per HTTP-Response zurückgegeben, um eine möglichst kurze Antwortzeit für den Nutzer zu erzielen. Erst dann findet die ausführliche Analyse der Daten statt, wofür mit Algorithmen der Bibliothek *opencv* eine Vielzahl an Merkmalen extrahiert wird. Ist ein Messwert vorhanden, wird dieser mit der Modellvorhersage verglichen und der resultierende Fehler berechnet. Zuletzt folgt das Speichern aller gesammelten Daten in den Datenbanken.

Die Konfiguration von Clients und Deployments ist in den Datenbanken hinterlegt und über die Konfigurations-UI durch Stakeholder anpassbar.

### 3.2.2. Datenbanken

Während aller Abläufe findet die Verarbeitung von Daten statt, die zentral gespeichert werden müssen. Die verschiedenen Prozesse stellen dabei individuelle Anforderungen an das Speichersystem, weshalb drei verschiedene Datenbanken zum Einsatz kommen, die in separaten Containern betrieben werden. Abbildung 3.3 zeigt alle eingesetzten Datenbanken mit ihren Datenflüssen zu und von den einzelnen Diensten[1].

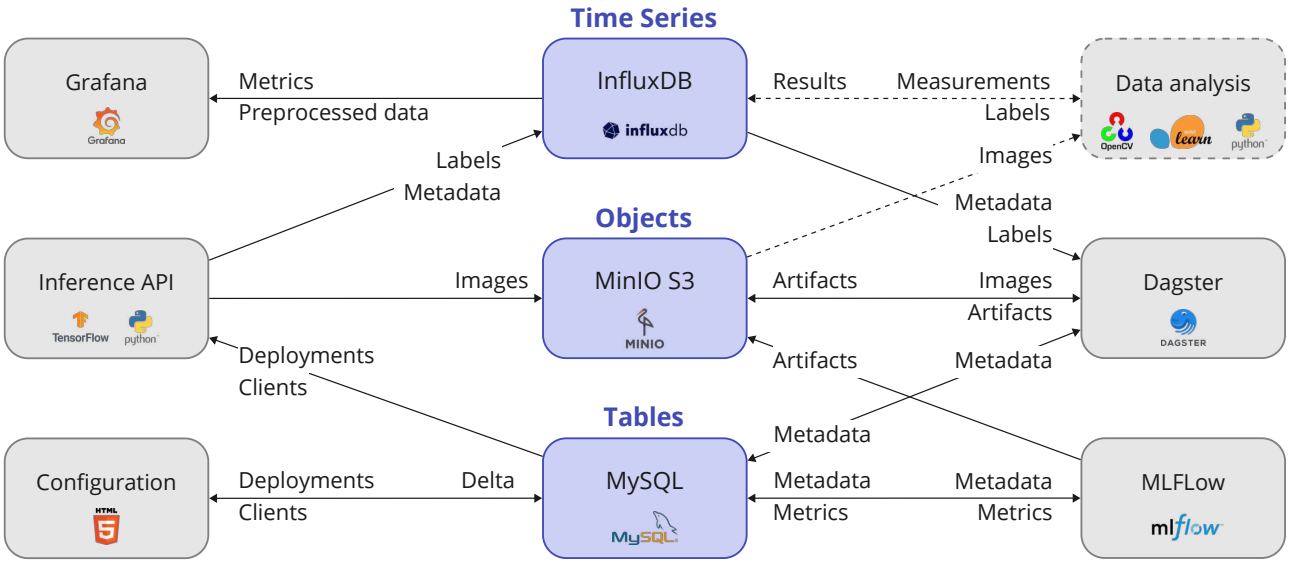



Abbildung 3.3.: Datenbanken mit ihren Kommunikationswegen

Anfragen an die Inference-API enthalten immer Messdaten und Ergebnisse, die abhängig von der Zeit sind. Diese Daten werden in Form von **Zeitreihen** in *InfluxDB* gespeichert, wobei jeder Datensatz einen Zeitstempel enthält. Die Zeitreihendatenbank ist speziell für das Speichern von sehr vielen kleinen, Datenpunkten optimiert und bietet spezifische Queries zum Filtern der Daten nach Zeitintervallen oder anderen Kriterien. Dadurch können bestimmte Datensätze schnell gefunden und abgefragt werden. Einzelne Datenpunkte werden in der Regel nicht nachträglich bearbeitet, sondern nur nach Ablauf einer Speicherfrist gelöscht.

Da Zeitreihendatenbanken nur bestimmte Datentypen speichern können und die Datenmenge pro Eintrag mit dem Ziel der schnellen Abfrage gering gehalten werden soll, werden die zu den Labels und Metadaten gehörenden Bilddaten als Datei in einem MinIO-Bucket gespeichert. Minio ist ein **Objektspeicher**, der beliebige Dateien speichern kann, denen je ein eindeutiger Schlüssel, ähnlich wie ein Dateipfad, zugeordnet ist. Ein Bucket hat bezüglich der Dateigröße oder des Dateityps nur wenige Einschränkungen, bietet aber keine Filter- oder Sortiermöglichkeiten, was die Abfrage erschwert. Der Objekt-Key der im Bucket abgelegten Bilddatei wird deshalb gemeinsam mit den anderen Messdaten in der Zeitreihendatenbank gespeichert und so

[1] der dargestellte Dienst „Datenanalyse“ ist kein eigenständiger Container, sondern Teil der Inference-API

mit der zugehörigen Messung verknüpft. Der Bucket wird auch für Artefakte wie Modelle oder Trainingsdaten genutzt, die während Dagster-Workflows und MLFlow-Experimenten entstehen.

Schließlich fallen auch viele strukturierte Daten an, die keinen Zeitstempel besitzen und regelmäßig geändert werden. Diese Daten werden in einer relationalen MySQL-Datenbank gespeichert, die speziell für **Tabellen** mit vielen Zeilen und Spalten optimiert ist. Den einzelnen Spalten wird immer ein fester Datentyp zugewiesen, der Abfrage und Bearbeitung erleichtert. Dafür bietet die Strukturierte Abfragesprache (SQL) spezielle Befehle, die Einträge filtern, sortieren und bearbeiten können. Die MySQL-Datenbank wird von Dagster und MLFlow für das Speichern von Metadaten und Konfigurationen genutzt und enthält auch Tabellen zur Verwaltung von Deployments und Edge-Geräten.

### 3.2.3. Datenaufbereitung in Grafana-Dashboards

Damit große Datenmengen aussagekräftig dargestellt werden können, ist die übersichtliche Visualisierung in Dashboards notwendig. Zur Erstellung und Anzeige der Dashboards wird Grafana verwendet. Grafana-Dasboards bestehen aus Panels, die verschiedene Daten in Form von Diagrammen, Tabellen oder Metriken anzeigen. Die Daten werden dabei immer einheitlich für einen bestimmten Zeitraum dargestellt, der durch den Nutzer in der UI festgelegt werden kann. Die Kommunikation mit den Datenbanken erfolgt über den InfluxDB-Adapter, der für die schnelle Abfrage gefilterter Daten im gewählten Zeitraum optimiert ist. Abgefragte Daten können vor der Darstellung in den vorgesehenen Panels noch mit verschiedenen mathematisch-statistischen Operationen aufbereitet werden. Besonders wichtig sind dabei Funktionen wie Min, Max, Mean, Std, Count und das Sortieren der Messungen in Gruppen zur Darstellung von Histogrammen. Jedes Dashboard ist individuell für eine bestimmte Aufgabe konfiguriert. Diese Konfiguration kann zwar dynamisch in der UI angepasst werden, ist aber grundsätzlich als JSON-Dokument im statischen Code hinterlegt. Langfristige Änderungen müssen deshalb immer in der Versionsverwaltung dokumentiert werden. Für das Monitoring des ML-Modells und der zur Bereitstellung benötigten Dienste gibt es verschiedene Grafana-Dashboards, die an das konkrete Modell und dessen Implementierung angepasst sind.

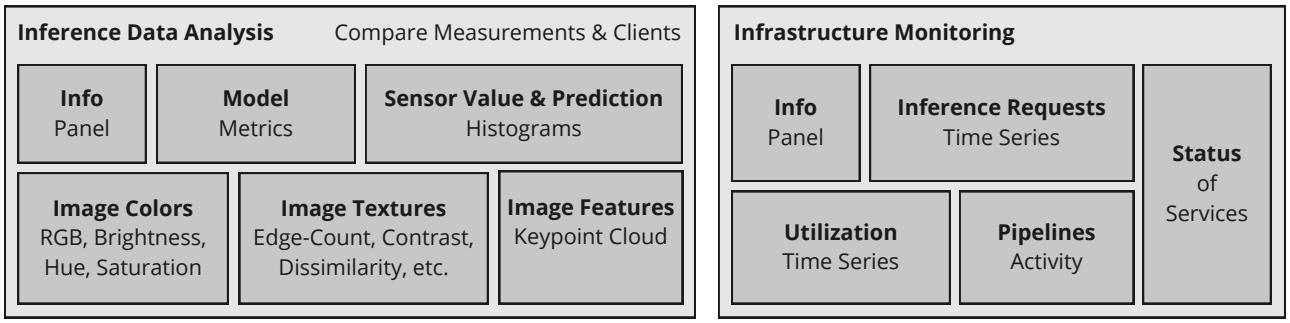



Abbildung 3.4.: Überblick Grafana-Dashboards

Abbildung 3.4 zeigt eine Übersicht über die Daten, die in den Dashboards dargestellt werden. Dabei kann grundlegend zwischen der Darstellung von modellbezogenen Daten und der Darstellung von systembezogenen Daten unterschieden werden. Das Dashboard **Inference Data Analysis**, welches auch im Anhang C.2 mit verschiedenen Messdaten dergestellt ist, dient zum Vergleich von gesammelten Daten unterschiedlicher Deployments und Edge-Geräten. Durch die Darstellung von Histogrammen des Modelloutputs, der Messdaten und der aus den Bilddaten extrahierten Informationen lassen sich Veränderungen sowie Anomalien in den Daten oder der Modellqualität erkennen. Gleichzeitig dient dieses Dashboard zur Identifikation der für das automatisierte Triggern des Modelltrainings-Zyklus relevanten Datenpunkte. Das **Infrastructure Monitoring** Dashboard enthält hingegen alle Informationen über die Nutzung sowie Auslastung von Servern und Diensten, die für die Bereitstellung des Gesamtsystems notwendig sind. Dazu gehört die Anzeige der CPU-, Speicher- und Netzwerkauslastung, die zeitliche Entwicklung der Anfragen an die Inference-API, die Pipeline-Aktivität sowie eine Statusübersicht aller Container. Mögliche Einschränkungen durch Fehler oder Überlast werden durch die grafische Analyse und individuelle Alarme frühzeitig erkannt und damit die Ausfallzeit des Systems minimiert. Weitere Dashboards zur Darstellung anwendungsspezifischer Informationen können durch minimale Änderung am statischen Code jederzeit hinzugefügt werden.

### 3.2.4. Modelltraining-Workflow, Experimente und Modellregister

Auf Basis der gesammelten Daten und den daraus gewonnenen Erkenntnissen kann das Modell kontinuierlich verbessert werden. Dafür wird ein Workflow benötigt, der die einzelnen Schritte von der Datenauswahl und Vorverarbeitung, über Training und Evaluation, bis hin zur Auswahl des besten Modells automatisiert ausführt und gleichzeitig alle relevanten Daten protokolliert. Dieser Workflow ist im Tool Dagster definiert. Dafür ist der Dagster-Container sowohl an alle Datenbanken als auch an den MLFlow-Server angebunden. Die einzelnen Schritte des Workflows sind als Python-Funktionen programmiert, die in der statischen Konfiguration in Dagster-Assets verpackt und miteinander verknüpft sind. Dabei wird an jedes Asset eine Konfiguration übergeben, die notwendige Parameter und Datenquellen für die Ausführung enthält.


Abbildung 3.5.: Dagster Asset-Gruppe zum Modelltraining in Dagster-UI

Der Zusammenhang und der aktuelle Status der Assets wird in der UI, siehe Abbildung 3.5, übersichtlich dargestellt. Jedes Asset steht dabei für ein generiertes Datenobjekt, das im Verlauf des Workflows erstellt wird. All diese Datenobjekte werden unter der eindeutigen ID des Workflow-Runs gruppiert im MinIO-Bucket gespeichert und der Object-Key im jeweiligen Asset hinterlegt.

Im ersten Schritt des Workflows wird der Datensatz für das Modelltraining zusammengestellt. Dafür wird eine Liste an Deployments angegeben, deren Daten für das Training verwendet werden sollen. Die einzelnen Messungen werden dann aus der InfluxDB gefiltert und die zugehörigen Bilder aus dem MinIO-Bucket geladen. Der fertige Datensatz wird konvertiert als JSON-Datei gespeichert. Im nächsten Schritt findet die Vorverarbeitung des Datensatzes statt. Dafür werden die Pixelwerte auf die Größe der Input-Schicht skaliert und normalisiert. Anschließend folgt das Aufteilen in einen Trainings- und Testdatensatz anhand des in der Assetkonfiguration übergebenen Test-Splits. Beide Datensätze werden unter derselben Run-ID im MinIO-Bucket gespeichert. Parallel wird das Modell anhand der statischen Modellarchitektur und gewählter Parameter definiert und in einer `.h5`-Datei gespeichert. Nun folgt das mehrfache Modelltraining mit variierenden Parametern, dessen Durchführung organisiert als MLFlow-Experiment erfolgt. Die Assetkonfiguration enthält die hierfür relevanten Informationen wie den Namen des Experiments oder die Liste der einzusetzenden Hyperparameter. Der Ablauf des Experiments ist in Abbildung 3.6 schematisch dargestellt.

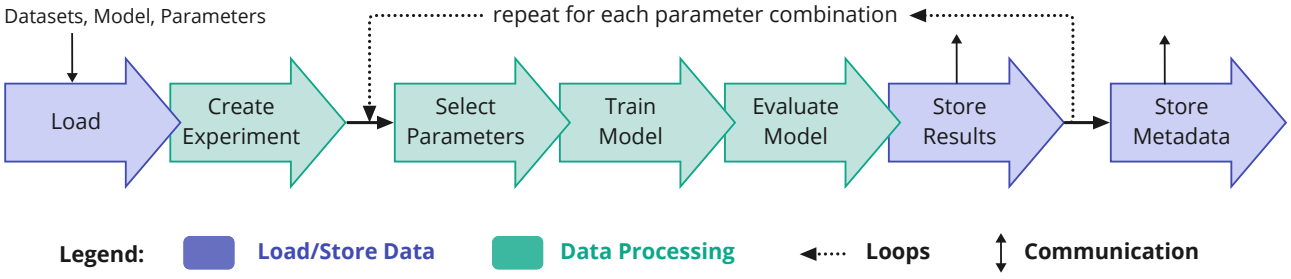



Abbildung 3.6.: Ablauf des Modelltraining-Experiments

Nach dem Import der zuvor generierten Trainings- und Testdaten und des Modells wird die Verbindung zum MLFlow-Server hergestellt und das Experiment angelegt. Nun wird der Trainingsprozess als MLFlow-Run mit einer bestimmten Parameterkombination gestartet. Nach Abschluss des Trainings und der Evaluation wird das Modell mit allen relevanten Informationen und Metriken im MLFlow-Run gespeichert, der in der MLFlow-UI grafisch dargestellt wird. Nach dem Modelltraining mit allen Parameterkombinationen wird das Experiment beendet und der Link zur MLFlow-UI im Asset-Output hinterlegt. Alle Informationen werden zusätzlich in einer JSON-Datei abgespeichert, sodass diese auch im folgenden Schritt - der Auswahl des besten Modells anhand bestimmter Evaluationsmetriken - zur Verfügung stehen. Hierfür werden

die Ergebnisse aller Runs des Experiments ausgelesen und verglichen. Die Run-ID des besten Modells wird im Asset-Output hinterlegt sowie in einer JSON-Datei gespeichert.

Optional kann nun noch die automatisierte Bereitstellung des neusten Modells erfolgen, indem das gewählte Modell nach Vergleich mit der bisherigen Version in das von MLFlow verwaltete Modellregister eingetragen wird. Dies kann in der MLFlow-UI alternativ auch manuell durch einen Entwickler erfolgen. Jedem Modell wird beim Registrieren eine eindeutige Versionsnummer zugeordnet, sodass die Historie der Modellversionen nachvollziehbar ist. Abhängig von der Konfiguration des Deployments in der Inference-Pipeline kann die höchste oder eine bestimmte Modellversion bereitgestellt werden. Dieser Zyklus kann nun mit nur einem Klick in der Dagster-UI gestartet werden und läuft dann vollständig automatisiert ab.

### 3.2.5. Feedback-Loops und Automatisierung

Als letzter Schritt zur kontinuierlichen Verbesserung des Modells wird ein Feedback-Loop implementiert, das durch die regelmäßige Analyse der Daten möglichen Drift am Modell erkennt und dann den Modelltrainingsprozess auslöst. Basis für die automatisierte Erkennung von Drift sind die Daten, die von den Anfragen an die Inference-Pipeline gesammelt, vorverarbeitet und in den Datenbanken gespeichert werden. Zur Identifikation der relevanten Datenpunkte dienen Dashboards, mit denen die Daten gefiltert, sortiert und vergleichen werden. Sind konkrete Datenpunkte, an denen sich Drift messen lässt, identifiziert und sinnvolle Grenzwerte festgelegt, werden die gewählten Datenpunkte mit jeweils einem Grafana-Alert überwacht. Diese werden dynamisch in der Grafana-UI erstellt und mit der Zeitreihendatenbank und der Pipeline verknüpft. Das Auslösen erfolgt per HTTP-Request an die Route `/retrain` der Inference-API, die eingehende Anfragen an die Dagster-API weiterleitet. Die neu trainierten Modelle werden dann im Modellregister hinterlegt und je nach Deployment-Strategie und Inference-Konfiguration automatisch bereitgestellt. Dadurch wird der Kreislauf der kontinuierlichen Anpassung und Verbesserung des Modells geschlossen. Das ML-Modell wird nun ohne aktiven Eingriff durch Stakeholder an die sich ändernden Bedingungen angepasst, wodurch Qualität und Funktionalität des ML-Produktes langfristig bei geringen Betriebskosten gewährleistet sind. Abbildung 3.7 stellt diesen Vorgang zur Veranschaulichung grafisch dar.

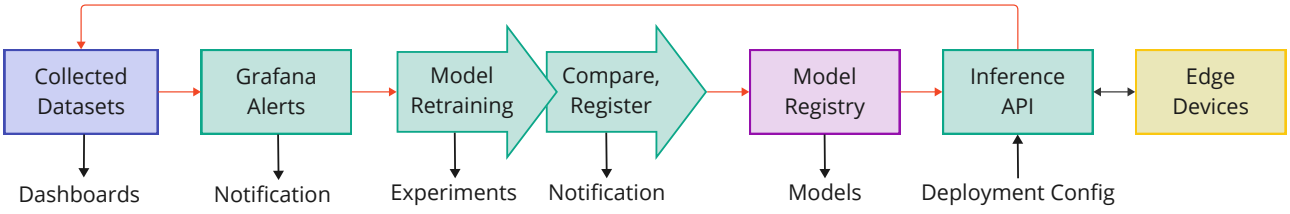



Abbildung 3.7.: Geschlossener Kreislauf zur kontinuierlichen Verbesserung des Modells

### 3.2.6. Logging, Nachvollziehbarkeit und Reproduzierbarkeit

Besonders in Bezug auf die weitreichenden Automatisierungen im System haben Nachvollziehbarkeit und Reproduzierbarkeit von Ergebnissen eine große Bedeutung. Jedes veränderte Verhalten des Modells basiert auf bestimmten Daten, die aus verschiedenen Prozessen und Algorithmen stammen. Um daraus Erkenntnisse zu gewinnen, ist es entscheidend, dass alle Abläufe nicht nur protokolliert, sondern auch sinnvoll organisiert werden. Aufgrund der verschiedenen Tools sind die Daten in unterschiedlichen Formaten und in unterschiedlichen Datenbanken gespeichert. Eine Auswahl der anfallenden Daten ist - nach Herkunft geordnet und nach Speicherort farblich gekennzeichnet - in Abbildung 3.8 dargestellt.

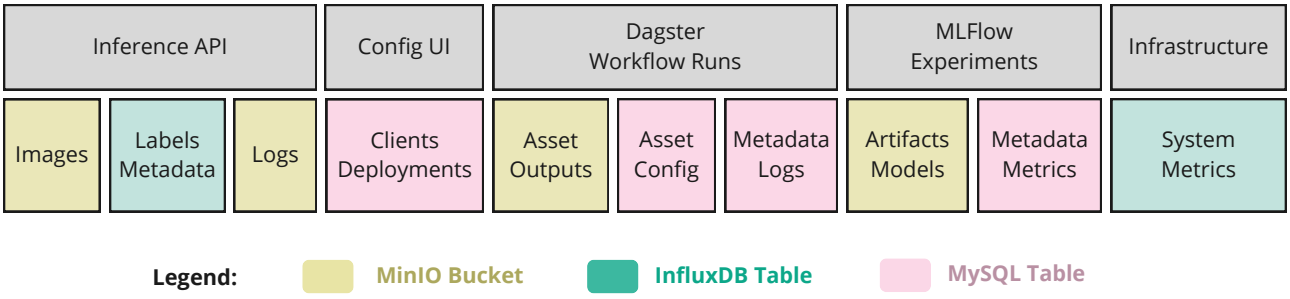



Abbildung 3.8.: Verschiedene Daten und ihre Speicherorte

Die verschiedenen Daten müssen miteinander verknüpft und für Stakeholder zugänglich gemacht werden. Dafür werden in den Tools Verknüpfungen in Form von Metadaten erstellt, die das schnelle Wechseln zwischen den grafischen Oberflächen der Tools ermöglichen. Dies wird nachfolgend am Beispiel einer Qualitätsminderung des Modells beschrieben, die durch einen Alarm in Grafana erkannt wurde und nun nachvollzogen und reproduziert werden soll, um mögliche Ursachen zu identifizieren.

Stakeholder erhalten die Informationen zum ausgelösten Alarm per Mail und können mit einem spezifischen Link die betroffenen Daten in der Grafana-UI aufrufen. Dort werden alle relevanten Informationen, nach dem betroffenen Deployment und dem relevanten Zeitraum gefiltert, übersichtlich dargestellt. Kann ein Fehler aufgrund von Rohdaten oder der Serverauslastung ausgeschlossen werden, wird per Klick im Modellinfo-Panel in das MLFlow-Modellregister zur konkreten Version des registrierten Modells gewechselt. Dort können erweiterte Informationen zum Modell und die den bestimmten Versionen zugehörigen Runs eingesehen werden. Der Run enthält konkrete Informationen und Metriken zum Trainingsprozess, die Aufschluss über den Trainingsverlauf und dessen Erfolg geben. Das zugehörige Experiment kann in der MLFlow-UI ebenfalls direkt geöffnet und grafisch verglichen werden, sodass Rückschlüsse bezüglich der Auswirkungen von gewählten Hyperparameter möglich sind. Dabei fällt auf, dass die verwendeten Batch-Größen aufgrund der nun größeren Trainingsdatensätze möglicherweise zu klein gewählt

wurden. Durch die Verknüpfung der MLFlow-Runs zum Dagster-Run können die Assets des Workflows aufgerufen und das Experiment-Asset mit abgeänderten Batch-Größen, jedoch mit identischen Eingangsdaten, wiederholt werden. Die Ergebnisse des neuen Experiments können dann wiederum im verlinkten MLFlow-Experiment grafisch verglichen und der Erfolg bewertet werden. Jeder einzelne Datensatz ist im System auf Basis seiner Herkunft über die grafischen Oberflächen abrufbar und über seine Metadaten anhand des logischen Datenflusses verknüpft. Dadurch kann der Grund für ein bestimmtes Verhalten identifiziert und das Ergebnis reproduziert werden.

## 3.3. Entwicklung an statischen Komponenten

Das System wird vollständig durch den statischen Code definiert, der im GitHub-Repository verwaltet wird. Für die Anpassung des Systems an ein konkretes ML-Projekt ist ein grundlegendes Verständnis dessen Aufbaus sowie des eigentlichen Codes notwendig. Nachfolgend wird die Realisierung wichtiger Bestandteile des Systems durch den statischen Code vorgestellt.

### 3.3.1. Aufbau des Repositories

Das Repository enthält verschiedene Ordner und Dateien, siehe Abbildung 3.9, die Programmcode und Konfiguration der Komponenten des dynamischen Systems enthalten. Im Ordner `.github/workflows/` befindet sich die Konfiguration der GitHub Actions CI/CD Pipeline, die für das automatisierte Testen und Bereitstellen des Systems verantwortlich ist. Die einzelnen Schritte dieser Pipeline sind in Form von Bash-Scripten im separaten Ordner `automation/` definiert, was die lokale Entwicklung und das Testen der Pipeline unterstützt. Die Inference-API und die Config-UI sind vollständig lokal als Flask-App entwickelt und gemeinsam mit dem `Dockerfile` zur Container-Definition in Unterordnern `src/..` gespeichert. Einige Services, wie Dagster und MLFlow, werden lokal kompiliert. Die entsprechenden Dockerfiles führen verschiedene Befehle zum Kompilieren und Starten der Services aus und sind ebenfalls in Unterordnern `src/..` abgelegt. Andere Dienste, wie die Datenbanken und Grafana, werden direkt als fertige Images aus der Docker-Registry bezogen und müssen nicht selbst kompiliert werden. Lediglich für den Grafana-Container wird die Datenbankdatei `src/grafana/grafana.db` als externes Volume mit dem Container verbunden, um Dashboards und Alerts zu übertragen. Die Konfiguration aller Container erfolgt über die `docker-compose.yml`, in welcher die verschiedenen Container und Volumes angegeben sind. Jedem Container werden dabei auch die benötigten Umgebungsvariablen wie Zugangsdaten, Ports, Datenbanknamen oder Dateispeicherorte aus der Datei `.env` übergeben. Die einzelnen Module des Modelltrainings, darunter auch die Modellarchitektur, sind im Ordner `src/model/` abgelegt. Für Tests der Module während der lokalen Entwicklung, siehe Abschnitt 3.3.2, kann das Jupyter Notebook `src/model/model_development.ipynb` genutzt werden. Bei der lokalen Entwicklung kann außerdem auf das `Makefile` zurückgegrif-

fen werden, die wichtige Befehle zum lokalen Starten und Testen der Dienste enthält. In der Markdown-Datei `readme.md` sind weitere Dokumentationen zu finden. Zuletzt enthält das Repository auch die Anwendung für das Edge-Gerät, die unter `src/edge/` abgelegt ist. Diese besteht aus verschiedenen Python-Scripten und muss an das konkrete Projekt angepasst werden.

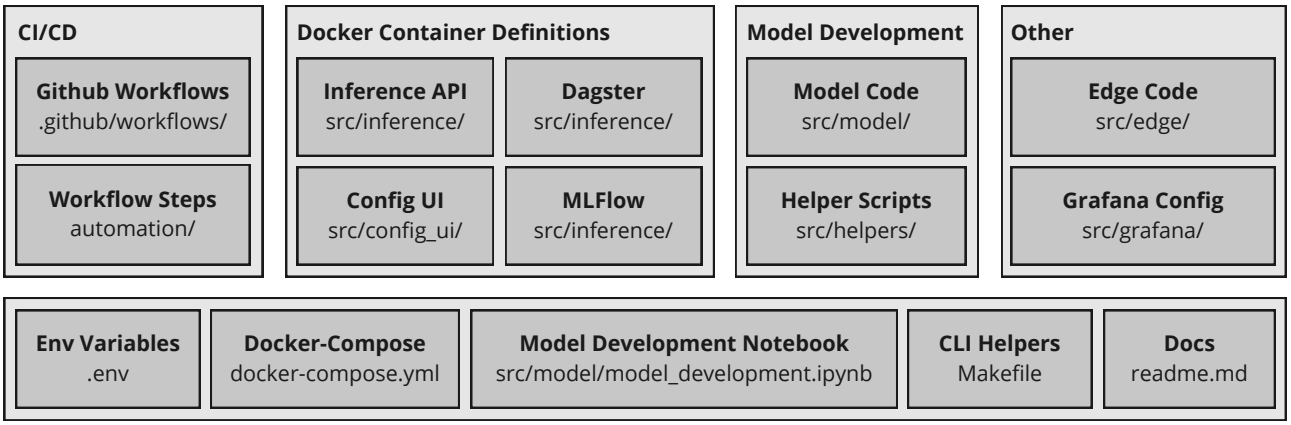



Abbildung 3.9.: Struktur wichtiger Elemente im Repository

### 3.3.2. Lokale Modellentwicklung im Jupyter Notebook

Die einzelnen Komponenten des Modelltrainings sind in getrennten Python-Modulen definiert, die später von den jeweiligen Dagster-Asset-Definitionen importiert werden. Einzelne Module sind dabei unabhängig voneinander, sodass mit korrekten Übergabewerten auch ein isolierter Test möglich ist.

- **Create Dataset:** Erstellen des Datensatzes aus Messdaten bestimmter Deployments
- **Preprocess Data:** Skalieren, Normalisieren und Teilen des Datensatzes
- **Create Model:** Definieren der Modellarchitektur und Rückgabe als Keras-Modell
- **Fit Model:** Setzen der Hyperparameter, Kompilieren des Modells und Training
- **Evaluate Model:** Evaluieren des Modells mit Testdaten und Berechnen der Metriken

Für das Entwickeln und das Testen der Module wird ein Jupyter-Notebook verwendet, in dem die einzelnen Module schrittweise importiert, ausgeführt und evaluiert werden können. Dafür wird jeder Schritt in einer eigenen Zelle definiert, die den Code zur Nutzung der Module enthält und entsprechend dokumentiert. Das Notebook erweitert die einzelnen Python-Module zusätzlich um Konfigurationen, Textausgaben sowie Visualisierungen und ermöglicht damit die interaktive und schnelle Durchführung von Änderungen am Modellcode (siehe Abbildung C.18).

Der konkrete Aufbau des Keras-Modells ist statisch und kann während des Betriebs nicht ohne Weiteres geändert werden, da dies eine vollständige Aktualisierung der MLOps-Systemversion erfordert. Dennoch kann es sinnvoll sein, die Modellstruktur während des Experiments zu verändern, um die Auswirkung verschiedener Architekturen auf die Modellqualität zu untersuchen. Zur Änderung des Dropout-Parameters[2] steht eine einfache Funktion zur Verfügung, die einzelne Schichten des Modells nachträglich durchläuft und den Parameter verändert. Für komplexere Änderungen gibt es die Möglichkeit des Hinterlegens verschiedener Variationen der Modellarchitektur im Modul, die mit der Eingangsvariable `use_variation` während des Experimentes dynamisch geladen werden können. Änderungen der Modellarchitektur und ihren Variationen werden durch das Versionskontrollsystem *git* dokumentiert und sind damit nachvollziehbar.

### 3.3.3. Konfigurationsverwaltung und Anpassung des Systems

Abhängig von den Anforderungen der ML-Anwendung muss die Konfiguration des Systems angepasst werden. Für einfache Anpassungen von Umgebungsvariablen, Datenbanknamen oder Ports wird die `.env`-Datei genutzt, die alle Konfigurationsvariablen enthält. Dazu gehören auch die Zugangsdaten der einzelnen Services, die vor dem produktiven Einsatz unweigerlich geändert werden müssen. Anpassungen an den einzelnen Diensten können in den Ordnern vorgenommen werden, in denen die jeweils zugehörigen Dockerfiles hinterlegt sind. So kann zum Beispiel ein Dagster-Asset im Ordner `src/dagster/asset/` hinzugefügt oder die Inference-API `src/inference/flask_app.py` um eine Route erweitert werden. Die yaml-Datei `docker-compose.yml` enthält eine Liste aller Container und Volumes und ist für die ganzheitliche Definition des Gesamtsystems verantwortlich. Dafür wird jedem Container eine Quelle zugewiesen, die entweder ein Build-Kontext, also der Verweis auf ein lokales Dockerfile, oder ein Image aus der Docker-Registry ist. Anschließend folgt die Definition der freigegebenen Ports, die Weitergabe der benötigten Umgebungsvariablen aus der `.env`-Datei, die Zuweisung eines Volumes und die Angabe von Abhängigkeiten zu anderen Containern. Ist die Docker-Engine auf dem Entwicklungsrechner installiert, kann das System mit dem Befehl `docker-compose up` gestartet und lokal bereitgestellt werden. Zur besseren Übersicht über die Container während der Entwicklung empfiehlt sich die Verwendung der Docker Desktop-Anwendung oder der VSCode-Erweiterung. Mit diesen grafischen Oberflächen können Logs und Zustände der Container eingesehen, die zugrundeliegenden Dateien inspiziert und die Container gestoppt oder neu gestartet werden. Jede Änderung an der statischen Systemkonfiguration wird durch das Versionskontrollsystem *git* dokumentiert. Verschiedene Versionen können kollaborativ auf verschiedenen Branches entwickelt und vor dem Commit in den main-Branch in dedizierten Testumgebungen evaluiert werden.

[2]Der Dropout-Parameter steuert die zufällige Deaktivierung von Neuronen zur Vermeidung von Overfitting.

### 3.3.4. Integration und Deployment mit GitHub Actions

Nachdem Änderungen am System lokal implementiert und getestet wurden, sollen diese in die Produktionsumgebung übertragen werden. Um dabei die Anforderungen an Qualitätssicherung und Automatisierung aus dem DevOps-Kontext zu erfüllen, wird das CI/CD-Tool *GitHub Actions* genutzt. Die in das Repository eingepflegten Änderungen wurden in einem Feature-Branch entwickelt, der durch das Öffnen eines Pull-Requests in den Main-Branch und damit in das Produktivsystem überführt werden soll. Ein Pull-Request vergleicht die Änderungen zwischen den beiden Branches, überprüft diese auf Konflikte bei der Zusammenführung und bietet Raum für den Austausch zwischen Stakeholdern. Gleichzeitig wird mit dem Erstellen eines Pull-Requests auf den Main-Branch eine Pipeline gestartet, deren Schritte in Abbildung 3.10 dargestellt sind.

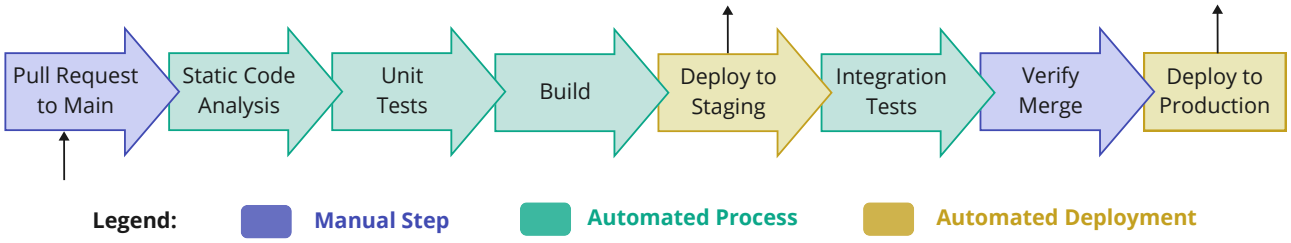



Abbildung 3.10.: Schritte der CI/CD-Pipeline in GitHub Actions

Die Pipeline beginnt mit einer statischen Codeanalyse, in der Python-Code und Dockerfiles unter Verwendung der Tools *flake8* und *hadolint* auf Syntaxfehler und einheitliche Formatierung geprüft werden. Anschließend führt *pytest* definierte Unittests aus, welche die Funktionalität und korrekte Standardisierung der einzelnen Module verifizieren. Dadurch wird die Codequalität überprüft und eine frühzeitige Fehlererkennung ermöglicht, sodass im nächsten Schritt alle Docker-Images des Systems, die in der `docker-compose.yml` definiert sind, mit dem Befehl `docker-compose build` kompiliert werden können. War dies erfolgreich, findet das Deployment in die dedizierte Testumgebung **Staging** statt, deren Konfiguration als *GitHub Environment* hinterlegt ist. Die Testumgebung wird auf Hardware ausgeführt, die mit der des Produktivsystems vergleichbar ist, sodass im nächsten Schritt die Funktionalität und Interaktion der Komponenten im Gesamtsystem getestet werden können. Nach der Verfügbarkeitsprüfung aller Dienste werden dafür die Routen der Inference-API mit simulierten Anfragen getestet. Für die automatisierte Validierung der weiteren Dienste wird die Route `/test` der API aufgerufen, welche einen erweiterten Test der Datenbanken und der Pipeline durchführt.

Erst wenn jeder dieser Schritte ohne Fehler durchgeführt wurde und verantwortliche Stakeholder ihre Zustimmung gegeben haben, wird der Merge in den Main-Branch freigegeben. Mit Ausführung des Merge-Commits wird der Code mit dem Main-Branch zusammengeführt und das System automatisch in die Produktionsumgebung **Production** ausgerollt.

Sofern die Edge-Software als Teil des statischen Systems betrachtet wird, muss auch diese in einer eigenen Testumgebung evaluiert werden. Da die konkrete Realisierung sehr individuell und die Anbindung an die Inference-API standardisiert ist, sind kontinuierliches Deployment und Integration der Edge-Software im Rahmen des praktischen Beispiels nicht implementiert. Im folgenden Kapitel wird die konkrete Realisierung eines spezifischen Edge-Geräts beschrieben.

## 3.4. Hardware und Software am Edge

Das Edge ist die Schnittstelle zwischen den Benutzern und dem ML-Modell. Dabei kann es sich sowohl um ein Hardwareprodukt mit entsprechender Software oder um eine reine Softwarelösung handeln. Wie genau das Edge aufgebaut ist, hängt von den Anforderungen des jeweiligen Projekts ab. Während die Anbindung an die Inference-API über HTTP standardisiert erfolgt, ist die Implementierung in spezifische Software- und Hardwareplattformen sehr individuell. Nachfolgend wird die mögliche Umsetzung eines spezifischen Edge-Geräts beschrieben, das während der praktischen Evaluation des Systems zum Einsatz kommt. Es basiert auf einem Raspberry Pi 4, der auf einer Roboterplattform montiert und mit verschiedenen Sensoren ausgestattet ist.

### 3.4.1. Anbindung an die Inference-API mit Python

Die Inference-API nimmt HTTP-Anfragen an die Route `/predict` entgegen, die alle erforderlichen Daten in festgelegter Form enthalten. Dazu gehören die ID des Edge-Gerätes, ein Authentifizierungstoken und - konkret für das praktische Beispiel - eine komprimierte Bilddatei im JPG-Format. Jedes Edge-Gerät muss zuvor in der Datenbank registriert werden, um eine eindeutige ID und sein Authentifizierungstoken zu erhalten. Der POST-Request zum Aufruf der API wird mit der Python-Bibliothek *requests* erstellt. Dabei ist zu beachten, dass das Senden der Anfrage eine unbestimmte Zeit in Anspruch nimmt und währenddessen die Ausführung des Programms blockiert. Um dies zu verhindern, wird die Anfrage mit dem Python-Modul *threading* in einem eigenen Thread gestartet, der parallel zur Hauptanwendung läuft. So kann die Anwendung weiterhin auf Benutzereingaben reagieren und gleichzeitig die Anfragen verarbeiten. Die Antwort der API wird als JSON-Objekt zurückgegeben und enthält die Vorhersage des Modells, die in der Edge-Anwendung beliebig weiterverarbeitet werden kann.

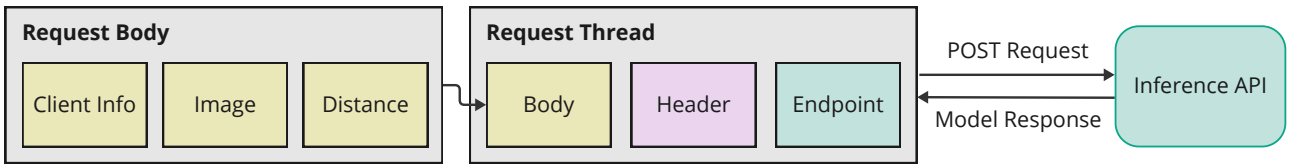



Abbildung 3.11.: Anfrage an die Inference-API

### 3.4.2. Konkrete Realisierung des Praxisbeispiels

Das entwickelte System wird an einem praktischen Beispiel demonstriert, getestet und evaluiert. Als Anwendungsszenario dient die Einparkhilfe eines Fahrzeugs, die allein auf Basis von Bilddaten der Rückfahrkamera den Abstand zu einem Hindernis berechnen soll, siehe Kapitel 1.3. Dies wird zur Vereinfachung und besseren Isolation von anderen Einflüssen im Modell realisiert, dessen Hardware- und Softwarekomponenten im Folgenden beschrieben werden.

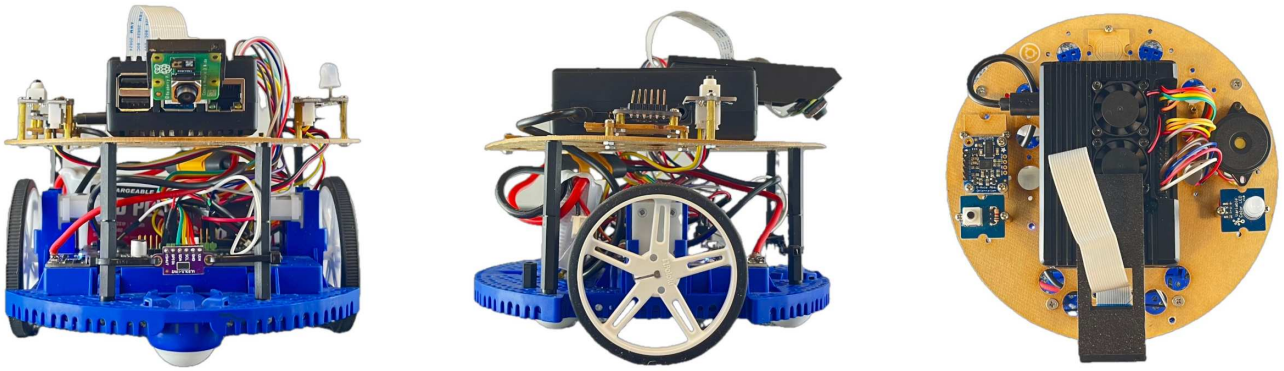


Abbildung 3.12.: Aufbau des Edge-Geräts mit Raspberry Pi 4 und Differenzialantrieb

Die in Abbildung 3.12 dargestellte Roboterplattform dient als Basis für das selbst entwickelte Fahrzeug. Sie verfügt über zwei unabhängig voneinander angetriebene Räder, wodurch die Steuerung des Fahrzeugs durch Anpassung der Radgeschwindigkeiten möglich ist. Der Antrieb ist mit der erforderlichen Elektronik ausgestattet, um die Motoren mittels eines PWM[3]- und Richtungssignals anzusteuern. Zudem ermöglicht der integrierte Motor-Encoder die Ausgabe einer festgelegten Anzahl positiver Taktflanken pro Radumdrehung. Als Steuerungseinheit wird ein Raspberry Pi 4 verwendet, der über seine General Purpose Input/Output (GPIO)-Pins mit der Elektronik des Antriebs verbunden ist. Die Stromversorgung erfolgt über einen an die Roboterplattform angeschlossenen Akku. Zur Aufnahme der Bilddaten wird die *Raspberry Pi Camera V3 Wide* verwendet und dafür mit einem 3D-gedruckten Halter an der Rückseite des Fahrzeugs befestigt sowie mit einem Flachbandkabel angeschlossen. Für die Abstandsmessung wird anstelle eines Ultraschallsensors, der nur eine Messauflösung im Zentimeterbereich aufweist, ein *VL53L0X* Time-of-Flight Laser Abstandssensor verwendet, dessen Messungen eine höhere Genauigkeit im Millimeterbereich aufweisen. Der Sensor wird, wie im realen Auto, unterhalb der Kamera an der Rückseite des Fahrzeugs angebracht und über den I2C-Bus mit dem Raspberry Pi verbunden. Zur Simulation der akustischen Einparkhilfe wird ein Piezo-Summer eingesetzt, der zusammen mit einer LED an die GPIO-Pins angeschlossen ist. Zusätzlich sind

[3]Pulsweitenmodulation (PWM) ist eine digitale Modulationsart zur Steuerung der Motorleistung

noch ein Taster und ein Beschleunigungssensor montiert, die jedoch vorerst nicht verwendet werden. Das Steuern des Fahrzeugs und das Starten der Anwendung erfolgt über einen universellen Controller, der drahtlos mit Bluetooth verbunden ist. Für die Kommunikation mit der Inference-Pipeline wird die integrierte WLAN-Schnittstelle des Raspberry Pi genutzt. Zur vereinfachten Softwareentwicklung auf dem Egde-Gerät wird eine Konsolenverbindung zu VSCode aufgebaut, mit welcher die Software direkt auf dem Raspberry Pi bearbeitet und ausgeführt werden kann.

Die Software besteht aus mehreren Python-Modulen, die je eine bestimmte Aufgabe übernehmen. Abbildung 3.13 veranschaulicht die verschiedenen Module des Systems, deren Zusammenspiel sowie den Austausch von Daten untereinander.

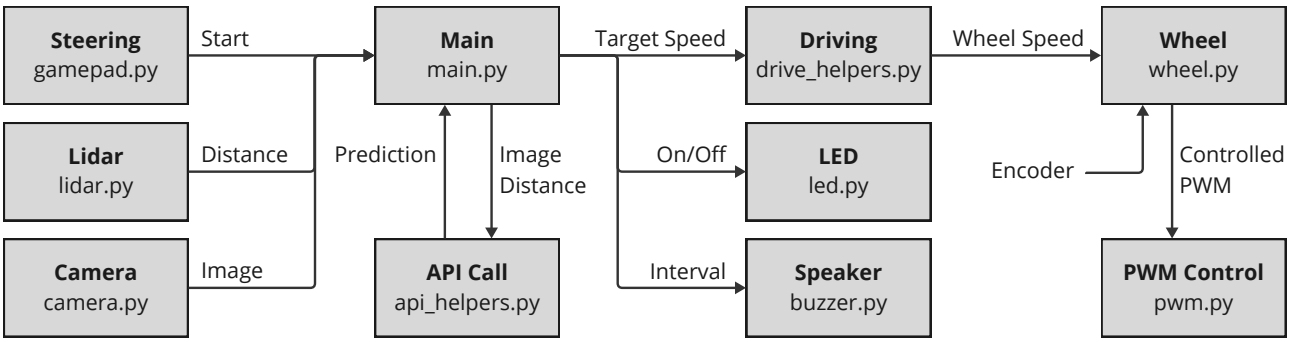



Abbildung 3.13.: Aufbau der Edge-Software aus Python-Modulen

Das Main-Modul bildet die zentrale Instanz des Programms und steuert die Hauptschleife. Es verbindet auf der einen Seite die Skripte zur Ansteuerung der Sensoren und auf der anderen Seite die Skripte zur Anbindung der Aktoren. Des Weiteren importiert das main-Modul die Hilfsanwendung zum Ausführen der HTTP-Anfragen an die Inference-API als Thread. Für die Ansteuerung der Sensoren und Aktoren werden hardwarespezifische Bibliotheken verwendet. Die Fahrregelung basiert auf der selbst implementierten Klasse `Wheel`, die einen PID-Regler zur Regelung der Radgeschwindigkeiten enthält und hardwarebasierte Interrupts sowie PWM-Generatoren zur Interaktion mit den Motorencodern und zur Steuerung der Motorleistung nutzt. Nur durch den geschlossenen Regelkreis kann sich das Fahrzeug unabhängig vom Untergrund ausreichend präzise bewegen, um ohne weitere Umgebungssensorik kurze automatisierte Fahrmanöver durchzuführen. Dies ist für das selbstständige Einparken an einer Wand notwendig, das ein zentraler Bestandteil des Vorgangs zur Messdatenaufnahme im Anwendungsszenario ist. Hierfür muss sich das Fahrzeug auch bei niedrigen Geschwindigkeiten exakt geradeaus bewegen. Der vollständige Edge-Code ist statisch unter `src/edge/` abgelegt.

Um Messdaten mit dem Fahrzeugmodell aufzunehmen und an die Inference-API zu senden, wird es mit dem Controller in eine beliebige Startposition gebracht, in der das Fahrzeug in einiger Entfernung rückwärts zur Wand steht, vor der es einparken soll. Nun wird zum Start des automatischen Einparkvorgangs der Knopf X auf dem Controller betätigt, was durch das Einschalten der LED auf dem Fahrzeug quittiert wird. Das Fahrzeug fährt nun selbstständig rückwärts auf die Wand zu und nimmt dabei in gleichbleibenden Intervallen Bilder und Abstandsmessungen auf, die an die Inference-API gesendet werden. Nun kann wahlweise die Modellantwort oder der Messwert des Sensors als Signal für die akustische Ausgabe und die Geschwindigkeitsregelung verwendet werden. Hierbei wird die Geschwindigkeit mit sinkender Entfernung zur Wand reduziert, um das Fahrzeug beim Erreichen eines bestimmten Abstands zu stoppen. Damit ist der Einparkvorgang beendet und die Steuerung wird wieder an den Controller übergeben, sodass nach manueller Wahl einer neuen Startposition ein weiterer Einparkvorgang gestartet werden kann. Mit jedem neuen Einparkvorgang werden etwa 25 Bilder und Abstandsmessungen aufgenommen die mit einer Frequenz von 4 Bildern pro Sekunde an die Inference-API gesendet werden. Dadurch ist Sammeln großer Mengen an Messdaten unter konkreten Umgebungsbedingungen möglich, welche die Grundlage für die Modellentwicklung und Modellanpassung in der nachfolgenden Evaluation bilden. Eine Auswahl von Messdaten bei verschiedenen Abständen ist zur Referenz im Anhang C.4 abgebildet.

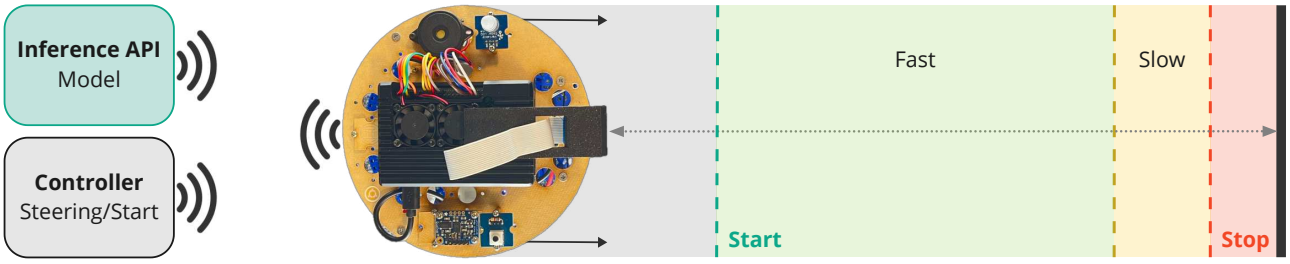



Abbildung 3.14.: Ablauf des automatischen Einparkvorgangs

Soll das System anhand der Beispielanwendung ohne Edge-Gerät getestet werden, können die Anfragen an die Inference-API auf Basis gespeicherter Messdaten simuliert werden. Eine detaillierte Anleitung zur lokalen Ausführung des Systems und zur Simulation von Messreihen aus der nachfolgenden Evaluation ist im Anhang B beschrieben.

# 4. Praktische Evaluation des Systems

Das entwickelte System dient als Werkzeug zur Realisierung der MLOps-Prinzipien in ML-Anwendungen. Der wichtigste Bestandteil ist der geschlossene Kreislauf zur kontinuierlichen Modellanpassung, mit welchem die Modellqualität langfristig sichergestellt werden soll. Modelle müssen überwacht werden, um Drift zu erkennen und diesen durch Neutraining nachvollziehbar auszugleichen. Dieses Kapitel behandelt den Einsatz des entwickelten Systems im Testumfeld, mit welchem die Funktion der genannten Bestandteile getestet und bewertet wird.

## 4.1. Versuchsplanung und Vorgehensweise

Die Evaluation des Systems erfolgt in einem vereinfachten Testumfeld, in dem verschiedene Drifte durch gezielte Veränderungen simuliert werden. Als Basis dient das in Kapitel 3.4 beschriebene Fahrzeugmodell, das vor einer weißen Wand einparkt, wobei Untergrundfarbe, Abstand und Ausrichtung variabel sind. Das entwickelte System wird auf einem lokalen Server mit Docker bereitgestellt. Dafür wurden der statische Code aus dem Repository geklont und die Variablen in der `.env`-Datei an die Umgebung angepasst. Der statische Code soll während des Experiments nicht verändert werden, sodass der Fokus ausdrücklich nicht auf der Optimierung der Modellarchitektur, sondern auf der Modellverbesserung durch Veränderung der Trainingsdaten liegt. Für die Bewertung ist die relative Veränderung der Metriken zwischen den trainierten Modellen entscheidend. Der Versuch soll zeigen, welche Veränderungen an der Umgebung die in Kapitel 2.3.2 analysierten Driften auslösen und wie diese mit dem entwickelten System anhand der protokollierten Daten (Anhang C.2) erkannt und behandelt werden können.

Für aussagekräftige Messergebnisse ist wichtig, dass die Umgebung, abgesehen von gezielten Veränderungen, konstant bleibt. Deshalb wird das Testumfeld vom Tageslicht abgeschirmt und künstlich beleuchtet. Die Kamerafixierung sowie der gleichbleibende Fahrzeugschwerpunkt werden vor jeder Messung überprüft. Die Durchführung wird in mehrere getrennte Phasen unterteilt, die mit je einer Änderung am Testumfeld einen bestimmten Drift auslösen sollen. Jede Phase umfasst zwei Messungen: eine vor und eine nach der Modellanpassung. Die Modellanpassung erfolgt durch das manuelle Ausführen der Dagster-Trainingspipeline.

- **Phase 1:** Training des Referenzmodells ohne Drift
- **Phase 2:** Simulation von Label Shift durch Vergrößerung des Abstands zur Wand
- **Phase 3:** Simulation von Covariate Shift durch Änderung der Untergrundfarbe
- **Phase 4:** Simulation von Concept Drift durch veränderte Ausrichtung zur Wand

Abbildung 4.1 zeigt den grundlegend identischen Ablauf jeder Versuchsphase. Nach Einrichtung der während der Phase konstanten Simulationsumgebung wird in der Config-UI ein neues Deployment angelegt und das aktuelle Modell zugeordnet. Dann werden etwa 500 Messdaten aufgenommen, im Grafana-Dashboard analysiert und mit der vorherigen Messung ohne Drift verglichen. Nimmt die Modellperformance wie erwartet ab, wird die Trainingspipeline manuell gestartet, sodass neue Modelle mit umfangreicheren Trainingsdaten trainiert werden, von denen das Beste im Modellregister eingetragen wird. Anschließend wird das verbesserte Modell mit einem neuen Deployment bereitgestellt und eine weitere Messreihe aufgenommen, die wiederrum mit der Vorherigen verglichen wird. Durch das Neutraining wird der Drift ausgeglichen, was durch eine erhebliche Verbesserung der Modellperformance bestätigt werden sollte.

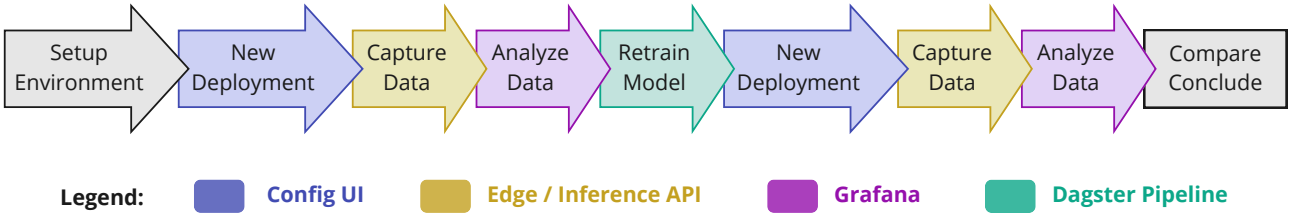



Abbildung 4.1.: Ablauf einer Versuchsphase

In jeder Phase wird die Modellperformance vor und nach der Modellanpassung anhand der Veränderung der Metriken gemessen. Gleichzeitig werden die protokollierten Datenpunkte analysiert und für die Erkennung von Drift relevante Datenpunkte identifiziert. Das System ist funktionsfähig, wenn sich verschiedene Arten von simuliertem Drift erkennen lassen, die Modellleistung mit wenigen Klicks durch ein Neutraining verbessert werden kann und die Anforderungen an Nachvollziehbarkeit sowie Reproduzierbarkeit erfüllt sind. Als Referenz zur Datenanalyse dient der mittlere Fehler in der Vorhersage, der aus der Differenz zwischen der Modellvorhersage und dem tatsächlich gemessenen Abstand berechnet wird. Die Nachvollziehbarkeit und Reproduzierbarkeit ist nur dann gewährleistet, wenn alle Daten vorhanden, verknüpft und zugänglich sind sowie alle Schritte der Modellanpassung nachvollzogen werden können.

Abschließend soll die automatisierte Modellverbesserung bei erneutem Covariate-Shift getestet werden, indem ein hierfür relevanter Datenpunkt identifiziert und mit einem Grafana-Alert permanent überwacht wird. Dies soll das automatische Starten der Trainingspipeline bei Überschreitung eines Schwellenwertes ermöglichen, sodass das System in der Lage ist, die kontinuierliche Modellanpassung bei Drift ohne manuellen Eingriff durchzuführen. So können der Aufwand im Betrieb reduziert und die Qualität der ML-Anwendung langfristig sichergestellt werden.

## 4.2. Einsatz der entwickelten Lösung im Testumfeld

### 4.2.1. Systemkonfiguration und initiales Modelltraining

In der ersten Phase wird ein Modell trainiert und evaluiert, das noch nicht von Drift betroffen ist. Um initiale Trainingsdaten zu sammeln, wird in der Config-UI das Deployment `env_beige_default` ohne ein verknüpftes Modell angelegt und aktiviert. Der Untergrund im Testaufbau ist mit einer beige-gefärbten Pappe ausgelegt (siehe Abbildung C.1). Das Fahrzeug wird immer möglichst senkrecht zur Wand ausgerichtet und fährt dann auf diese zu. Dabei werden erst Anfragen an die Inference-API gesendet, wenn der Abstand zur Wand kleiner als 250 Millimeter ist. Abbildung 4.2 zeigt die gewählte Konfiguration schematisch.

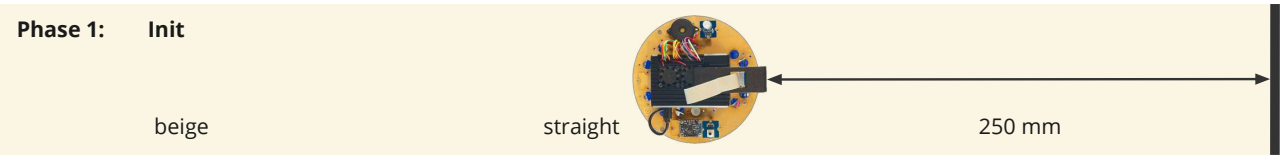



Abbildung 4.2.: Konfiguration von Fahrzeug und Umgebung in Phase 1

Der Einparkvorgang wird nun gestartet und etwa 20 Mal wiederholt, bis der resultierende Datensatz mindestens 500 Einzelmessungen enthält. Dabei wird das Fahrzeug nach jedem Einparkvorgang manuell in eine neue Startposition gebracht. Die Messdaten werden im Grafana-Dashboard `Compare Measurements` visualisiert und ausgewertet (siehe Abbildung C.5). Das Histogramm der Messwerte in Abbildung 4.3 (Panel `Sensor Value` im Dashboard) zeigt, dass die Messungen nicht gleichmäßig über den Messbereich verteilt sind. Dies ist darauf zurückzuführen, dass die Fahrzeuggeschwindigkeit beim Parken distanzabhängig reduziert wird und die Aufnahme der Messungen unabhängig davon in konstanten Intervallen erfolgt. Da dem Deployment kein Modell zugeordnet ist, sind alle Vorhersagen gleich null und die Fehler somit identisch zum Messwert. Der minimal und maximal gemessene Abstand entspricht dem eingestellten Bereich. Die aufgezeichneten Daten haben die gewünschte Qualität und sind für das Training des Modells geeignet.

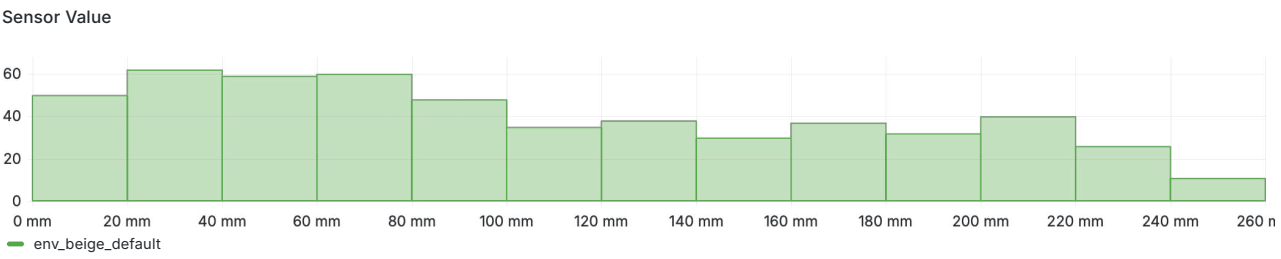



Abbildung 4.3.: Histogramm der Abstandsmessungen beim Einparken in beiger Umgebung

Um ein Modell auf Basis der Messdaten zu trainieren, wird im MLFlow-Modellregister ein neues Modell mit dem Namen `Evaluation` angelegt und die Trainingspipeline in der Dagster-UI mit Angabe von Deployment, Hyperparametern, Experiment-Namen und Registerziel gestartet. Nach erfolgreicher Ausführung der Trainingspipeline ist das Modell mit dem geringsten Evaluationsfehler automatisch - versioniert als `v1` - im Modellregister abgelegt. Dieses Modell wird nun dem Deployment `val_env_beige_default` zugeordnet und eine weitere Messreihe aufgenommen. Alle Messungen enthalten jetzt auch die Vorhersagen des Modells, sodass der mittlere und maximale Fehler im Dashboard (siehe Abbildung C.6) analysiert und damit die Modellqualität bewertet werden kann. Der mittlere Fehler beträgt 6,85 Millimeter, woraus abgeleitet werden kann, dass das Modell mit den gewählten Hyperparametern und seiner Architektur grundsätzlich funktioniert und in der vereinfachten Umgebung erstaunlich gute Vorhersagen liefert. Auffällig ist jedoch, dass die Vorhersagen des Modells den Messbereich nicht vollständig abdecken und eine einzelne Vorhersage weit außerhalb liegt. Daraus kann abgeleitet werden, dass weiterhin Fehler und somit Verbesserungspotential bestehen, auf die im Rahmen dieser Evaluation jedoch nicht weiter eingegangen wird. Die weiteren Daten zeigen, dass der Farbraum beider Messungen nahezu identisch ist, die Metriken `Textur` jedoch eine leichte Differenz aufweisen. Insbesondere beim Kontrast ist eine deutliche Veränderung zu erkennen, die aufgrund einer Fokusverschiebung der Kamera entstanden sein könnte. Dieser Umstand wird während der weiteren Phasen berücksichtigt, um die Vergleichbarkeit der Messdaten zu gewährleisten. Das Ergebnis der ersten Phase ist ein trainiertes Modell, das funktionsfähig ist und in der konkreten Umgebung eine zufriedenstellende Modellqualität aufweist.

### 4.2.2. Simulation von Label Shift

Die Testumgebung wird in der zweiten Phase so verändert, dass Label Shift auftritt. Der Bereich in dem das Fahrzeug parkt und Daten an die API sendet, wird dafür - wie in Abbildung 4.4 dargestellt - auf 350 Millimeter erweitert. Dadurch enthalten die neuen Messdaten auch einige Labels, die nicht in den bisherigen Trainingsdaten vorkamen. Zu erwarten ist deshalb die Verschlechterung der Performance, welche durch das Neutraining auf erweiterter Datenbasis ausgeglichen werden soll.


Abbildung 4.4.: Konfiguration von Fahrzeug und Umgebung in Phase 2

Für die Messung wird das Deployment `env_beige_distant` angelegt und nach der Zuordnung des Modells `v1` aktiviert. Nachdem die Messdaten, wie in der ersten Phase beschrieben, aufge-

nommen sind, werden die Metriken mit und ohne Drift im Grafana-Dashboard (siehe Abbildung C.7) verglichen. An beiden Histogrammen, die zusätzlich in Abbildung 4.5 dargestellt sind, ist eine deutliche Veränderung zur vorherigen Messung zu erkennen. Die Messwerte sind nun - wie erwartet - über einen größeren Bereich verteilt, wohingegen die Vorhersagen des Modells weiterhin nur den ursprünglichen Bereich abdecken. Die große Zahl an Vorhersagen im Bereich von 220 bis 260 Millimetern ist ein guter Indikator für auftretenden Drift in Form von Label Shift, da alle größeren Abstände nicht im Trainingsdatensatz enthalten sind und das Modell diese mit bekannten Maximalwerten labelt. Am Anstieg des mittleren Vorhersagefehlers auf 19 Millimeter wird deutlich, dass die Modellperformance durch den Label-Shift erheblich abgenommen hat. Auch die Farbräume zeigen eine leichte Verschiebung der RGB-Verteilung, die auf neue Bilddaten bei größerem Abstand zurückzuführen ist. Mit den gesammelten Messdaten kann das Modell neu trainiert werden, um den erkannten Drift auszugleichen.

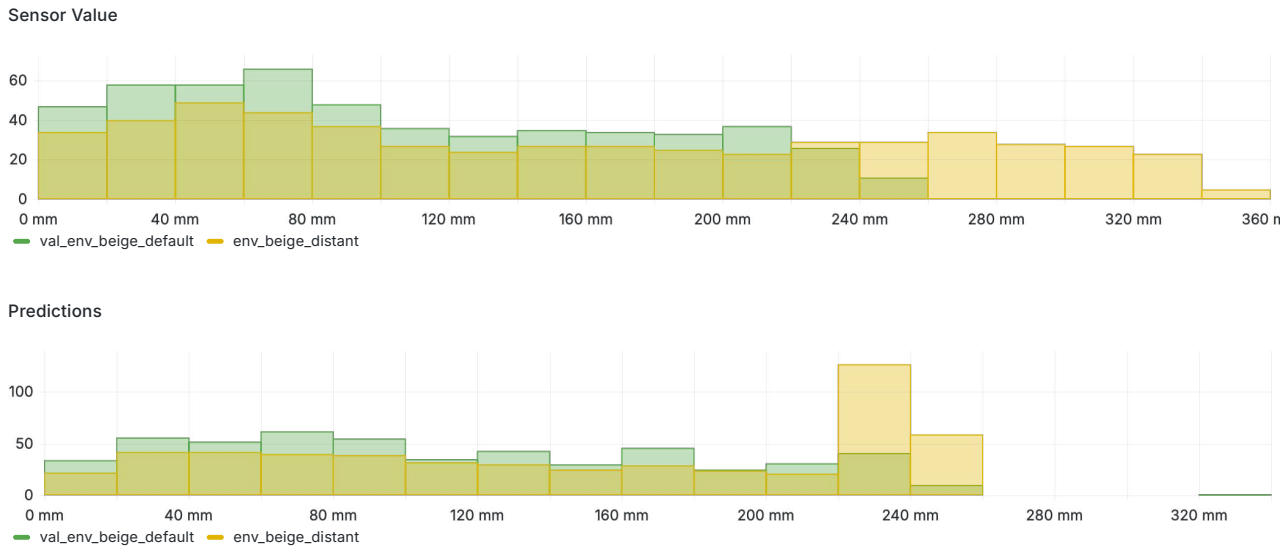



Abbildung 4.5.: Histogramme von Messwert und Vorhersage bei Label Shift

Die Trainingspipeline wird - wie zuvor - in der Dagster-UI gestartet und nutzt dabei sowohl die neuen als auch die alten Messdaten. Das beste Modell wird als `v2` im Modellregister abgelegt, dem Deployment `val_env_beige_distant` zugeordnet und mit einer weiteren Messreihe evaluiert. Die Ergebnisse im Grafana-Dashboard (siehe Abbildung C.8) zeigen, dass sich die Modellperformance durch das Neutraining erheblich verbessert hat. Die Vorhersagen sind gleichmäßig verteilt und decken den gesamten Messbereich ab. Der mittlere Fehler ist auf 4,83 Millimeter gesunken, was vermutlich auf die größere Anzahl an Trainingsdaten zurückgeführt werden kann. Farbraum und Texturen sind wie erwartet unverändert geblieben. Die Ergebnisse der zweiten Phase bestätigen die Funktionalität des Systems. Der Label-Shift war an der Veränderung der statistischen Verteilung der Vorhersagen deutlich erkennbar und wurde durch das gezieltes Neutraining auf erweiterter Datenbasis ausgeglichen.

### 4.2.3. Simulation von Covariate Shift

In der dritten Phase wird der beige Untergrund gegen einen grünen Untergrund ausgetauscht (siehe Abbildung C.2), sodass Covariate Shift auftritt. Ähnlich wie beim zuvor simulierten Label Shift ist auch beim Covariate Shift davon auszugehen, dass die Modellperformance beeinträchtigt wird. Dies soll durch ein erneutes Training des Modells kompensiert werden.

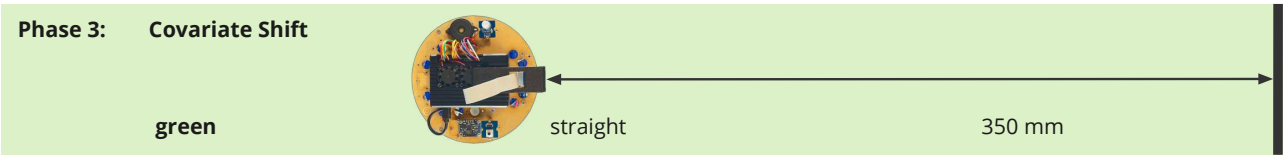



Abbildung 4.6.: Konfiguration von Fahrzeug und Umgebung in Phase 3

Die Messung wird im Deployment-Umfeld `env_green` durchgeführt und im Grafana-Dashboard im Vergleich zur vorherigen Messung dargestellt (siehe Abbildung C.9). Aus den Histogrammen geht hervor, dass die Vorhersagen des Modells im neuen Umfeld nicht mehr gleichmäßig über den Messbereich verteilt, sondern im oberen Bereich konzentriert sind. Dies weist auf eine Verschlechterung der Modellperformance hin. Zudem ist der mittlere Fehler ist aufgrund des Covariate Shifts auf 178 Millimeter gestiegen, was dazu führt, dass das Modell seine Funktion in der aktuellen Umgebung nicht mehr erfüllt und Kollisionen zu erwarten sind. An den Farbräumen ist zu erkennen, dass sich die Umgebung tatsächlich verändert hat. Die Histogramme der mittleren RGB-Farbwerte, siehe Abbildung 4.7 zeigen eine deutliche Verschiebung, die sich auch in den weiteren Farbmetriken widerspiegelt. Während die Helligkeit dabei nahezu konstant bleibt, weist die Sättigung eine starke Veränderung auf. Auch Texturen und Keypoints weichen aufgrund von höheren Kontrasten zwischen Farben in der neuen Umgebung voneinander ab.

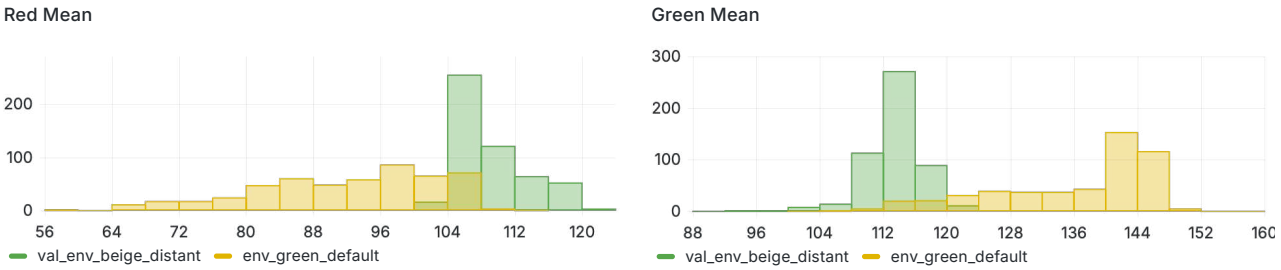



Abbildung 4.7.: Farbhistogramme in beiger und grüner Umgebung

Die Korrektur des Drift erfolgt wie zuvor durch Neutraining des Modells, sodass auch die Zusammenhänge in der neuen Umgebung erlernt werden. Dafür wird im Deployment `val_env_green` eine neue Messreihe erfasst und anschließend ausgewertet (siehe Abbildung C.10). Die Vorhersagen sind nun wieder gleichmäßig über den Messbereich verteilt, wodurch der mittlere Fehler

auf 15,5 Millimeter gesunken ist. Damit ist die Modellperformance erneut auf ein zufriedenstellendes Niveau zurückgekehrt, jedoch weiterhin niedriger als in der ersten und zweiten Phase. Dies ist darauf zurückzuführen, dass die Trainingsdaten nun aus zwei verschiedenen Umgebungen stammen und somit komplexere Zusammenhänge abgebildet werden müssen. Entgegen der Erwartung zeigt das Histogramm der blauen Farbmetrik eine leichte Verschiebung, die sich ebenfalls negativ auf die Modellperformance auswirken könnte und vermutlich durch ungewollte Schatten in der Umgebung entstanden ist. Die Ergebnisse der dritten Phase zeigen, dass Veränderungen an Vorhersagen und Farbräumen ein Indikator für Covariate Shift sind und das System in der Lage ist, diesen durch Neutraining auszugleichen.

### 4.2.4. Simulation von Concept Drift

Concept Drift tritt auf, wenn sich die grundlegende Beziehung von Eingang und Ausgang des Modells verändert. Dies kann zum Beispiel durch eine andere Montage der Kamera oder durch eine andere Ausrichtung des Fahrzeugs beim Parken ausgelöst werden. In der vierten Phase wird der Einparkvorgang deshalb in unveränderter Umgebung, jedoch schräg mit einem Winkel von maximal 30 Grad zur Wand, durchgeführt (siehe Abbildung C.3).

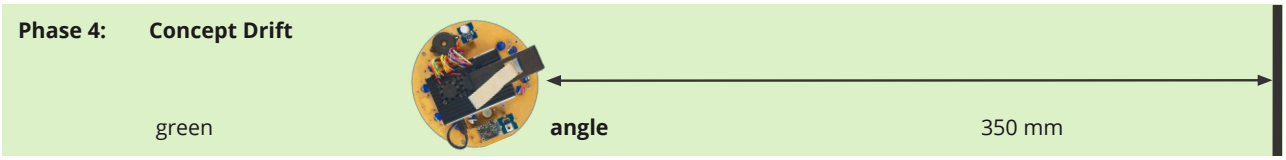



Abbildung 4.8.: Konfiguration von Fahrzeug und Umgebung in Phase 4

Für die Evaluation wird das Deployment `env_green_angle` angelegt und eine Messreihe aufgenommen, die in Grafana ausgewertet wird (siehe Abbildung C.11). Der Vergleich mit der vorherigen Messung ohne Drift zeigt, dass sich die Modellperformance durch den Concept Drift verschlechtert hat. Auffällig ist jedoch, dass der mittlere Fehler bei 30,0 Millimetern und damit nur geringfügig über dem Ausgangswert liegt. Der maximale Fehler ist hingegen auf über 100 Millimeter angestiegen und die Vorhersagen decken nur den Bereich bis 300 Millimeter ab. Die leichte Verschiebung des Vorhersagen-Histogramms Richtung Null zeigt, dass das Modell den neuen Zusammenhang nicht korrekt abbildet und ein Großteil der Vorhersagen zu niedrig ausfällt. Im Nahbereich bis 20 Millimeter wurden deutlich weniger Vorhersagen getroffen.

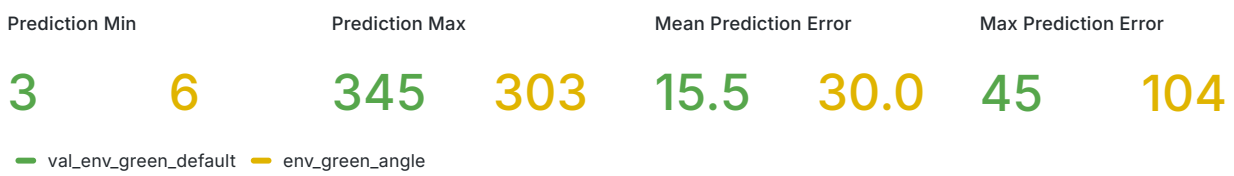



Abbildung 4.9.: Vorhersagen und Fehler bei Concept Drift

Dieses Ergebnis ist plausibel, da das Modell auf einen senkrechten Einparkvorgang trainiert wurde und die neue Perspektive beim schrägen Einparken mit dem gelernten Zusammenhang zwar teilweise, aber nicht vollständig übereinstimmt. Demzufolge verändert sich der Fehler je nach Messbereich unterschiedlich stark und die Modellperformance sinkt im Mittel mäßig.

Durch Neutraining soll auch der Concept Drift korrigiert werden, wofür die Trainingsdaten nun auch schräge Einparkvorgänge enthalten sollen. Die neue Messreihe wird im Deployment `val_env_green_angle` aufgenommen und im Grafana-Dashboard mit der vorherigen Messung verglichen (siehe Abbildung C.12). Durch das Neutraining wurde die Modellperformance erheblich verbessert. Der mittlere Fehler ist trotz eines erheblichen Ausreißers auf 12,9 Millimeter gesunken. Farben, Texturen und Features beider Messreihen weisen wie gewünscht nur geringe Unterschiede auf. Die Ergebnisse zeigen, dass Concept Drift etwas schwieriger zu erkennen sein kann, da die Veränderungen nicht so offensichtlich wie bei anderen Formen von Drift sind. Für die Erkennung von Concept Drift kann die statistische Verteilung der Vorhersagen auf Änderungen, wie eine Verschiebung oder Verzerrung, untersucht werden. Auch Concept Drift konnte durch das Neutraining auf erweiterter Datenbasis erfolgreich ausgeglichen werden.

### 4.2.5. Automatisierte Modellverbesserung bei Drift

Bisher musste der Drift manuell von einem Entwickler in den Daten erkannt werden, woraufhin die Trainingspipeline zur Modellanpassung gestartet wurde. Um eine kontinuierliche Modellanpassung ohne hohe Betriebskosten zu gewährleisten, wird nun die automatisierte Modellverbesserung im Falle von Drift getestet. Dafür wird im Testumfeld ein weiterer Covariate Shift erzeugt, indem der Untergrund von beige zu rot geändert wird (siehe Abbildung C.4). Für den automatischen Start der Trainingspipeline bei Drift wird ein Grafana-Alert eingerichtet, der eine bestimmte Metrik überwacht und bei Überschreitung einen Webhook über die Inference-API an Dagster sendet. Als Metrik könnte zwar direkt der mittlere Vorhersagefehler verwendet werden, jedoch liegt dieser in anderen Anwendungsfällen nur selten vor. Aus diesem Grund wird die über einen Zeitraum von fünf Minuten gemittelte Sättigung der Bilddaten gewählt, die sich in Phase zwei als guter Indikator für den konkreten Covariate Shift erwiesen hat. Das Mitteln der Messwerte ist notwendig, da die Sättigung auch abhängig von der Distanz zur Wand ist und hierdurch Schwankungen unterliegt. Da sich die Sättigung in beiger Umgebung zwischen 40 und 60 bewegt, wird der untere Grenzwert auf 40 und der obere Grenzwert auf 60 festgelegt. Der Grafana-Alert löst aus, sobald ein Grenzwert für länger als eine Minute überschritten wird.

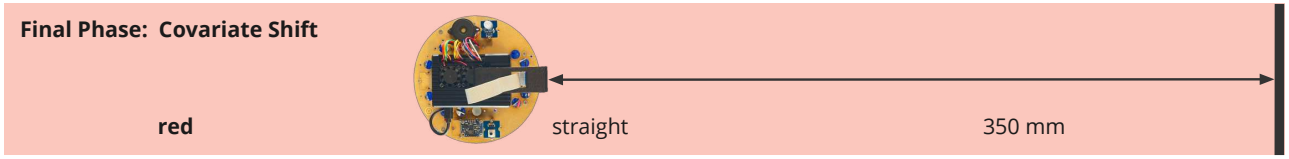



Abbildung 4.10.: Konfiguration von Fahrzeug und Umgebung zur automatisierten Verbesserung

Nach erfolgreicher Einrichtung des Alerts wird das Deployment `test_retraining_trigger` angelegt, das durch die Angabe der Version `latest` immer das aktuellste Modell verwendet. Nun wird die Messung kontinuierlich über einen Zeitraum von zwanzig Minuten durchgeführt, wobei der Untergrund nach der Hälfte der Zeit von beige auf rot wechselt. Aufgrund der Überschreitung des Grenzwertes hat der Grafana-Alert die Trainingspipeline - wie gewünscht - nach einigen Minuten automatisch gestartet. Durch die große Anzahl an Trainingsdaten und die eingeschränkte Rechenleistung des Testsystems dauerte das Neutraining etwa 40 Minuten, sodass nach erfolgreichem Deployment der neuen Modellversion zur Validierung weitere zehn Minuten gemessen wurde. Die Messdaten des Deployments sind im Grafana-Dashboard in die Zeitabschnitte vor Drift, (siehe Abbildung C.13) nach Drift (siehe Abbildung C.14) und nach Neutraining (siehe Abbildung C.15) unterteilt, sodass die Veränderung wie in den Phasen zuvor analysiert werden kann. Da die Messdaten bei diesem Versuch zeitlich abhängig sind, wird in einem weiteren Dashboard zusätzlich der Verlauf von Sättigung, Vorhersage und Fehler, jeweils gemittelt über die vorherigen fünf Minuten, dargestellt (siehe Abbildung C.16). Die Ergebnisse zeigen, dass sich der Vorhersagefehler durch den verursachten Covariate Shift von 8,47 Millimeter auf 45,5 Millimeter verschlechtert hat. Gleichzeitig ist die Sättigung der Bilddaten von 45.0 auf 156 angestiegen, sodass der Grafana-Alert sicher ausgelöst wurde. Der zeitliche Verlauf der Sättigung zeigt einen, durch Mittelwertbildung langsamen, Anstieg nach Umgebungsänderung und das Überschreiten des Grenzwertes von 60 bereits nach weniger als einer Minute. Auch an den gemittelten Vorhersagen ist eine deutliche Veränderung zu erkennen, was ebenfalls ein Indikator für den Drift ist. Deutlich sichtbar ist als Referenz auch der Anstieg des mittleren Vorhersagefehlers, der durch den Drift nach der Änderung verursacht wurde.

Nach vollständig automatisierter Ausführung des Trainings- und Deploymentkreislaufs ist der mittlere Fehler wieder auf 6,17 Millimeter gesunken, sodass die Vorgaben an die Modellqualität wieder erfüllt sind. Die Ergebnisse dieser Phase zeigen, dass das System in der Lage ist, Drift zu erkennen und ohne manuellen Eingriff durch Neutraining zu korrigieren. Dafür können relevante Metriken in Dashboards leicht identifiziert und mit Grafana-Alerts überwacht werden, sodass bei Drift automatisch ein Zyklus der kontinuierlichen Modellanpassung ausgelöst wird.

### 4.2.6. Nachvollziehbarkeit, Reproduzierbarkeit und Dokumentation

Da viele Schritte des Modelltrainings und der Modellanpassung automatisiert ablaufen, ist es wichtig, dass alle Prozesse nachvollziehbar und reproduzierbar sind. Hierfür sollten die Abläufe und Daten direkt in den jeweiligen Services protokolliert worden sein. Nachfolgend werden die Daten der vorausgegangenen Messungen und Modellanpassungen in den verschiedenen Services verortet und auf Vollständigkeit geprüft. Konkret soll die automatische Modellanpassung bei Drift in der fünften Phase rückverfolgt und mit anderen Hyperparametern auf gleicher Datenbasis reproduziert werden.

Alle Deployments werden mit Name, ID, Modell, Version und Status in der MySQL-Datenbank und in Config-UI (siehe Abbildung C.19) visualisiert. Zusätzlich wird hier auch die Anzahl der aufgenommenen Messdaten angezeigt, deren Analyse durch die Auswahl des Deployments in Grafana erfolgt. Neben den Deployments ist in der Config-UI auch der Client mit Name und ID einsehbar. Das zum Deployment `test_retraining_trigger` gehörende Modell wird als `Evaluation` mit Version `latest` identifiziert.

Im MLFlow-Modellregister (siehe Abbildung C.21) kann das Modell `Evaluation` mit seinen Versionen dargestellt werden. Der letzte Trainingsdurchlauf hat als neueste Version `v9` erzeugt, deren Details mit einem Klick auf die Version angezeigt werden, um zum zugehörigen Run `bustling-bird-389` zu gelangen. Hier können alle angelegten Modellmetriken und Konfigurationen eingesehen werden (siehe Abbildung C.23). In der Ecke links oben wird außerdem das zugehörige Experiment angezeigt (siehe Abbildung C.22), das auch alle anderen Runs des Trainingsdurchlaufs enthält, deren Metriken in der MLFlow-UI miteinander verglichen werden können (siehe Abbildung C.24). Jedes registrierte Modell und jeder Experiment-Run ist mit dem Tag `dagster_run` versehen, das die zugehörige Ausführung der Dagster-Pipeline verknüpft.

In der Dagster-UI (siehe Abbildung C.25) werden alle Runs der Trainingspipeline gekennzeichnet mit ihrer ID zeitlich geordnet dargestellt. Der identifizierte Run mit der ID `4533b6ff` wird ganz oben angezeigt und ist zusätzlich mit dem Tag `trigger: Grafana Alert` gekennzeichnet, das auf die automatische Auslösung durch den Grafana-Alert hinweist. Die Details des Runs können durch Klick auf die ID eingesehen werden. Hier werden die einzelnen Assets des Runs (siehe Abbildung C.26) sowie zugehörige Konfigurationen und Logs grafisch dargestellt. Mit Klick auf ein konkretes Asset können dessen Details, wie zum Beispiel die Metadaten des Experiment-Assets, eingesehen werden (siehe Abbildung C.27). Zu den Metadaten gehört auch der Verweis auf das Experiment in der MLFlow-UI und auf die Output-Datei des Assets im MinIO-Bucket. Zum Durchführen eines neuen Experiments kann das Experiment-Asset nun einzeln gestartet werden, wobei in der Konfiguration die neuen Hyperparameter und die Run-ID angegeben werden. So verwendet das Asset die identischen und bereits im vorherigen Run vorverarbeiteten Datensätze.

Die Datensätze aller Assets und Runs sind im Minio S3-Bucket abgelegt, der ebenfalls über eine UI verfügt. Auf oberster Ebene wird im Bucket ein Ordner für jeden Service (siehe Abbildung C.28) dargestellt, der ausschließlich die jeweils zugehörigen Daten enthält. Im Ordner `/features` sind die aufgenommen Bilder gespeichert und nach Deployment sortiert (siehe Abbildung C.29). Die Dateien können einzeln heruntergeladen und geöffnet werden. Die zugehörigen Labels und Metadaten sind in der Zeitreihendatenbank abgelegt. Der Ordner `/dagster` enthält einen Unterordner für jeden Run, in welchem sich die Output-Dateien der Assets befinden (siehe Abbildung C.30). Hierzu gehören die Datensätze, das Modell und einfache JSON-Dateien

mit Metadaten. Innerhalb des Ordners `/mlflow` sind die Artefakte aller Runs in Unterordnern nach Run-ID sortiert (siehe Abbildung C.31).

Die Nachverfolgung der Daten des konkreten Modells hat gezeigt, dass alle generierten Daten wie gewünscht angelegt und miteinander verknüpft wurden, sodass der Durchlauf der Trainingspipeline vollständig nachvollziehbar und reproduzierbar ist. Für die Arbeit in großen Teams fehlt noch eine zentrale Oberfläche, die alle Daten der Services zusammenführt und die einfache Suche und Filterung bestimmter Datensätze ermöglicht. Zusätzlich sollten Anpassungen am dynamischen Code, - wie etwa die Gründe für ein Neutraining, Überlegungen zur Konfiguration von Grafana-Alerts oder die Auswahl der Hyperparameter - zentral dokumentiert werden, was momentan noch nicht möglich ist.

## 4.3. Auswertung mit Bezug auf die Zielsetzung

Die Evaluation des Systems zeigte, dass die entwickelte Lösung ihr Ziel erreicht hat, indem sie Drift in den Daten zuverlässig erkannte und eigenständig darauf reagierte, ohne dass ein manueller Eingriff erforderlich war. Über die Config-UI ließen sich Clients und Deployments einfach anlegen und verwalten. Die Inference-API empfing Anfragen von Clients, führte das Modell aus, extrahierte Metriken aus Bildern und speicherte anfallende Daten strukturiert in InfluxDB und MinIO. Dank Grafana konnten diese Daten umfassend visualisiert und analysiert werden. Dadurch war es möglich, relevante Metriken für die Driftüberwachung zu identifizieren und mittels Alerts effektiv zu überwachen. Die Trainingspipeline, implementiert mit Dagster, wurde erfolgreich ausgeführt und war durch die Assets mit umfänglich gespeicherten Metadaten nachvollziehbar sowie reproduzierbar. Das Training wurde mehrfach mit unterschiedlichen Parametern durchgeführt und in MLFlow als Experiment gebündelt dokumentiert. Die jeweils besten Modelle wurden automatisiert im Modellregister gespeichert und durch die Auswahl der Modellversion `latest` direkt bereitgestellt.

In den Versuchen konnte gezeigt werden, dass Label Shift, Covariate Shift und Concept Drift mit dem entwickelten System anhand bestimmter Metriken erkannt und durch Neutraining ausgeglichen werden können. Die Überwachung der Sättigung mit einem Grafana-Alert ermöglichte die automatische Erkennung von simuliertem Covariate Shift, sodass der entstandene Fehler ohne manuelles Eingreifen durch Modellanpassung korrigiert wurde. Alle anfallenden Daten wurden wie gewünscht gespeichert und mit den Services verknüpft, sodass die Anforderungen an Nachvollziehbarkeit und Reproduzierbarkeit erfüllt waren.

Für den Einsatz in großen Teams könnte das System noch um eine zentrale Oberfläche erweitert werden, die alle Daten zusammenführt und die individuelle Dokumentation von gewählten Konfigurationen ermöglicht.

# 5. Zusammenfassung und Ausblick

## 5.1. Ergebnisse der Arbeit

MLOps ist eine zentrale Schlüsseltechnologie, um die Herausforderungen beim Einsatz von Machine Learning in Produkten eines Unternehmens zu bewältigen. Durch tiefgreifende Recherche wurde mit dieser Arbeit ein umfassendes Verständnis für die Hintergründe, Prinzipien und Methoden von MLOps erlangt, auf dessen Basis eine universelle Softwarelösung für die Anwendung in ML-Projekten implementiert und am praktischen Beispiel evaluiert wurde.

Um technische Schulden in ML-Systemen zu vermeiden, sind in der Literaturrecherche konkrete Prinzipien und die zu deren Einhaltung benötigten Funktionalitäten identifiziert worden. Die größte Herausforderung bestand in der Vielzahl unterschiedlicher Ansätze und Methoden, die analysiert und zu einer kohärenten Lösung kombiniert werden mussten. Zur Realisierung der einzelnen Funktionalitäten wurden Tools und Frameworks aus dem DevOps und MLOps-Bereich ausgewählt und vorgestellt. Insbesondere die Untersuchung fertiger Produkte führender Cloudanbieter zeigte nicht nur deren Stärken, sondern auch die Potenziale und Vorteile einer eigenen Implementierung auf.

Die entwickelte Softwarelösung vereint die identifizierten Prinzipien und Funktionalitäten zu einem umfassenden System zur Anwendung von MLOps in ML-Projekten. Sie besteht aus einem statischen Teil zur manuellen Konfiguration der Komponenten und einem dynamischen Teil, dessen Eigenschaften sich mit der Datenbasis verändern können. Durch Anpassungen des statischen Codes kann das System flexibel auf unterschiedliche Anwendungsfälle und Anforderungen zugeschnitten werden. Die zentrale Versionierung gewährleistet die Synchronisierung und die Nachvollziehbarkeit aller Änderungen, was die Zusammenarbeit in Teams erleichtert und die Wartbarkeit des Systems erhöht. Automatisierte CI/CD-Pipelines überwachen die Einhaltung aller Vorgaben und stellen die Funktionalität des Systems sowohl mit Unit-Tests als auch mit Integrationstests sicher. Nach erfolgreicher Prüfung wird das System automatisch als Multi-Container-Anwendung bereitgestellt, die unabhängig von der Infrastruktur betrieben werden kann. Das bereitgestellte dynamische System stellt das ML-Modell über eine API zur Verfügung und protokolliert alle Anfragen und Ausgaben kontinuierlich in einer Datenbank. Diese Daten können mithilfe von Dashboards visualisiert und analysiert werden, um Veränderungen und Auffälligkeiten zu erkennen. Eine modulare Trainingspipeline ermöglicht das automatisierte und reproduzierbare Neutraining des Modells unter Verwendung variierender Daten und Parameter. Die Ergebnisse der Trainingsdurchläufe werden als Runs gebündelt in Experimenten gespeichert, sodass sie in einer grafischen Benutzeroberfläche analysiert und verglichen werden können. Ein Modellregister verwaltet die verschiedenen Modellversionen und stellt diese für

die API bereit. Vordefinierte Trigger ermöglichen die automatische Anpassung des Modells bei Veränderungen, sodass die Qualität der ML-Anwendung kontinuierlich gewährleistet ist.

Für die praktische Evaluation wurde ein Modell zur Kamera-basierten Abstandsmessung am Fahrzeug entwickelt und gezielt verschiedenen Arten von Drift ausgesetzt. Als Edge diente ein selbst entwickeltes Miniaturfahrzeug, das in spezifischen Testumgebungen eingesetzt wurde, um Daten für das Training und die Evaluation zu generieren. Label Shift, Covariate Shift und Concept Drift konnten anhand der gesammelten Daten in den Dashboards zuverlässig erkannt und durch das manuelle Starten der Trainingspipeline erfolgreich ausgeglichen werden. Nach Identifikation relevanter Datenpunkte wurde dieser Prozess vollständig automatisiert, wodurch Drift in Echtzeit kompensiert und die Qualität der Anwendung kontinuierlich verbessert werden konnte. Die Anforderungen an Nachvollziehbarkeit, Reproduzierbarkeit und Erklärbarkeit sind durch die umfassende Dokumentation und Visualisierung aller Daten und Abläufe vollständig erfüllt. Durch die Evaluation am praktischen Beispiel konnte gezeigt werden, dass das entwickelte System die identifizierten Prinzipien von MLOps erfolgreich anwendet und damit die Herausforderungen beim Einsatz von ML in der Produktionsumgebung bewältigt.

## 5.2. Schlussfolgerungen

Die Ergebnisse der Arbeit zeigen die Relevanz von MLOps für den erfolgreichen Einsatz von ML im Unternehmen. Gleichzeitig verdeutlichen sie die Vielschichtigkeit dieses Themas: Es existiert nicht die eine Standardlösung, sondern eine Vielzahl unterschiedlicher Ansätze für spezifische Problemstellungen. Die eigentliche Herausforderung besteht darin, diese einzelnen Komponenten effizient zu einem ganzheitlichen und praxistauglichen System zu integrieren.

Mit der praktischen Realisierung wurde verdeutlicht, dass ein solches System zwar technisch umsetzbar ist, jedoch stets an den jeweiligen Anwendungsfall und dessen spezifische Anforderungen angepasst werden muss. Das entwickelte System wurde deshalb möglichst anpassbar gestaltet und besteht aus modularen Komponenten, die jeweils spezifische Funktionalitäten abdecken und eng miteinander verzahnt sind. Eine zentrale Herausforderung bestand darin, geeignete Tools und Frameworks zu identifizieren, eigene Dienste zu entwickeln und alle Komponenten gemeinsam in einem System zu vereinen.

Eine weitere Schwierigkeit stellte die Versionierung des gesamten ML-Projekts dar, da es sowohl herkömmlichen Programmcode, der aktiv von Entwicklern angepasst wird, als auch große Mengen dynamischer Daten umfasst. Um diesen Anforderungen gerecht zu werden, wurde eine klare Trennung zwischen der klassischen Versionsverwaltung und einer individuellen Versionierung für die dynamischen Daten eingeführt. Damit sind alle Änderungen nachvollziehbar und reproduzierbar, jedoch würde eine zentrale Verwaltungsoberfläche, die alle Daten aus den ver-

schiedenen Services zusammenführt und aufbereitet, komplexe Strukturen noch übersichtlicher machen. Für manuelle Anpassungen im dynamischen System könnten geeignete Werkzeuge zur Dokumentation von Änderungen implementiert werden.

Die Evaluation am praktischen Beispiel zeigte, dass das System für eine konkrete Anwendung unter Laborbedingungen erfolgreich eingesetzt werden kann. Beim Transfer in die Praxis müssen jedoch noch weitere Herausforderungen, wie die Skalierbarkeit der Dienste oder das Verarbeiten nicht gelabelter Daten, bewältigt werden.

Mit dieser Arbeit wurde ein umfassendes Verständnis für MLOps erlangt und darauf basierend eine Softwarelösung entwickelt, die eine solide Grundlage für die erfolgreiche praktische Anwendung von Machine Learning Operations in Unternhemen bildet.

## 5.3. Ausblick

Fortlaufende Forschung und Weiterentwicklung im Bereich ML und MLOps zeigen, dass diese Themen weiterhin großes Potenzial für Innovationen und neue Lösungsansätze bieten. Damit das entwickelte System auch in Zukunft erfolgreich eingesetzt werden kann, sind kontinuierliche Anpassungen an den aktuellen Stand der Technik unerlässlich. Fortschritte in Bereichen wie AIOps, AutoML und Edge Computing bieten vielversprechende Möglichkeiten zur Steigerung von Effizienz und Agilität der MLOps-Prozesse und sollten daher zukünftig integriert werden.

Darüber hinaus ist es essenziell, das System kontinuierlich anhand von praktischen Erfahrungen aus ML-Projekten weiterzuentwickeln, um neue Funktionalitäten zu integrieren und bestehende Schwächen zu adressieren. Dabei rücken Aspekte wie die Skalierbarkeit der Komponenten, die Verarbeitung nicht gelabelter Daten, die zentrale Dokumentation von Änderungen sowie Sicherheits- und Datenschutzanforderungen in den Fokus, die in der Laborumgebung noch nicht relevant waren. Auch die Modellausführung direkt auf den Edge-Geräten spielt eine wichtige Rolle, um Latenzen zu reduzieren und eine unabhängige Funktion ohne permanente Internetverbindung zu gewährleisten. Dies erfordert auch Lösungen zur kontinuierlichen Integration und Bereitstellung der Edge-Software sowie zum Management einer großen Anzahl an Edge-Geräten.

Langfristig wird jedoch nicht nur die technologische Weiterentwicklung, sondern auch die Integration der MLOps-Prinzipien in die Unternehmenskultur entscheidend für den Erfolg des MLOps-Systems sein. Dies erfordert nicht nur geeignete Werkzeuge und Prozesse, sondern auch die gezielte Sensibilisierung und Schulung der beteiligten Teams. Nur durch die enge Verknüpfung von Technologie, Organisation und Kultur lässt sich das volle Potenzial von MLOps ausschöpfen, um die Transformation von Prozessen, Dienstleistungen und Produkten durch Machine Learning im Unternehmen erfolgreich zu realisieren.

# Anhang

# A. Weiterführende Themen

## A.1. Intelligente Automatisierung mit AIOps

Artificial Intelligence for IT Operations (AIOps) ist ein Konzept, das sich mit der Automatisierung und Optimierung von IT-Betriebsprozessen durch die Nutzung von KI beschäftigt. Es umfasst die Anwendung von Algorithmen und ML-Modellen, um die Infrastruktur, Netzwerke und Anwendungen in Echtzeit zu überwachen, Probleme zu identifizieren und proaktiv Lösungen zu implementieren. Während AIOps im Bereich der IT-Operation zunehmend an Bedeutung gewinnt, besteht die Verwechslungsgefahr mit Machine Learning Operations, da beide Konzepte auf KI und Automatisierung beruhen und ihre Bezeichnungen sehr ähnlich sind. Im Folgenden wird der wesentliche Unterschied zwischen beiden Konzepten erläutert, um die ergänzende Rolle von AIOps in einem MLOps-System zu verdeutlichen.

AIOps und MLOps verfolgen zwar ähnliche Ziele der Automatisierung, unterscheiden sich jedoch in ihrem Fokus und ihrer Anwendung. AIOps konzentriert sich auf die Optimierung von IT-Infrastrukturen und die Automatisierung von Betriebsabläufen. Dabei wird KI eingesetzt, um große Mengen an Betriebsdaten zu analysieren und fundierte Entscheidungen zu treffen, mit denen Anomalien und Ausfälle vorhergesagt sowie Probleme in Echtzeit behandelt werden können. MLOps hingegen befasst sich mit dem Lebenszyklus von ML-Modellen von der Entwicklung über das Training bis hin zu Deployment und Monitoring in der Produktionsumgebung. Dabei geht es nicht darum, Automationen durch eine KI zu realisieren, sondern um den effizienten und zuverlässigen Betrieb von ML-Modellen. Die Konzepte haben gemeinsame Schnittstellen und können sich gegenseitig ergänzen, um den Betrieb von Modellen in der Produktion zu optimieren. AIOps-Tools ermöglichen die automatisierte Erkennung von Modell-Drift, Änderungen in den Eingabedaten und anderen Anomalien, die die Modellleistung beeinträchtigen können. Während diese Überwachung derzeit manuell über Dashboards und Grafana-Alerts erfolgt, kann AIOps den Monitoring-Prozess automatisieren und so eine umfassendere und effizientere Überwachung der Modelle gewährleisten. Ein praktisches Beispiel hierfür ist die automatische Generierung von Alarmmeldungen bei Modellen, die aufgrund von Drift oder unerwarteten Labels ungenaue Vorhersagen liefern.

Mit AIOps kann der Betrieb von ML-Modellen - durch den Einsatz von KI - weiter optimiert werden. Die automatisierte Überwachung und Analyse ermöglicht die frühzeitige Identifikation potenzieller Probleme und deren unmittelbare Behebung ohne manuellen Eingriff. In Verbindung mit MLOps gewährleistet dies eine konstant hohe Modellqualität und Verfügbarkeit bei reduzierten Betriebs- und Instandhaltungskosten [46][47].

## A.2. Modellausführung auf Edge-Geräten

Die Bereitstellung des ML-Modells für Edge-Geräte ist ein vielseitiges Themenfeld, das neben der beschriebenen Inference-API auch andere Ansätze umfasst. Im Moment findet die Modellausführung auf leistungsstarken Servern in der Cloud statt, was zwar der Skalierbarkeit und Flexibilität zugutekommt, jedoch auch den Nachteil mit sich bringt, dass alle Daten über eine Netzwerkverbindung an die Cloud gesendet, dort verarbeitet und zurückgegeben werden müssen. Dies erfordert eine permanente Internetverbindung und führt zu Latenzzeiten sowie Datenschutzproblemen. Im Fall des praktischen Anwendungsbeispiels, bei dem ein Fahrzeug mittels Rückfahrkamera eine Abstandsmessung durchführt, sind all diese Aspekte von entscheidender Bedeutung: Das Fahrzeug hat nicht immer eine stabile Internetverbindung, die Latenzzeiten müssen minimal sein und nicht jeder Kunde möchte die benötigten Bilddaten in die Cloud senden. Die ausschließlich cloudbasierte Modellausführung ist daher nicht ideal.

Um mit diesen Einschränkungen umzugehen, kann Edge Computing, also das Ausführen des Modells direkt auf dem Edge-Gerät, eingesetzt werden. Anstatt die Inference-API für die Bearbeitung vieler einzelner Anfragen zu nutzen, wird mit jedem Deployment einmalig das gesamte Modell auf das Edge-Gerät übertragen. Das Modell kann nun ohne Internetverbindung und ohne Netzwerklatenz lokal ausgeführt werden. Dies ermöglicht eine schnelle und zuverlässige Verarbeitung der Daten und erhöht die Datensicherheit, da keine Verarbeitung außerhalb des Geräts erfolgt. Mit der lokalen Berechnung werden auch Cloudressourcen eingespart, die nun jedoch von den Edge-Geräten selbst bereitgestellt werden müssen. Dies stellt eine Herausforderung dar, da Rechenleistung und Speicherplatz auf Edge-Geräten begrenzt sind. Je weniger Ressourcen zur Verfügung stehen, desto wichtiger wird die Optimierung des Modells und der Inferenzprozesse. Dafür können leichtgewichtige Versionen der gängigen ML-Bibliotheken wie *TensorFlow Lite* oder *PyTorch Mobile* zum Einsatz kommen, die speziell für die ressourceneffiziente Ausführung am Edge entwickelt wurden. Sie ermöglichen die Konvertierung und Komprimierung von Modellen in ein kompaktes Format, das auf mobilen Geräten wie Smartphones und sogar Mikrocontrollern ausgeführt werden kann. Neben den eingeschränkten Ressourcen haben Edge-Geräte eine dezentrale Natur, welche die Verwaltung und Aktualisierung der Modelle erschwert. So kann es vorkommen, dass das Modell auf einem Gerät veraltet ist oder sich Unterschiede der Hardware auf die Modellausführung auswirken, was im MLOps-System berücksichtigt und überwacht werden muss. Da die Inference-API nun nicht mehr für die Modellausführung zuständig ist, stehen viele Metriken und Messdaten auch nicht mehr automatisch in der Cloud zur Verfügung. Um die Qualität und Leistung des Modells zu überwachen und die kontinuierliche Verbesserung durch MLOps zu gewährleisten, müssen Mechanismen implementiert werden, die - mit Einverständnis des Nutzers - relevante Daten lokal sammeln und bei Bedarf in die Cloud übertragen. Hierfür muss die API um entsprechende Routen erweitert und die Datenverarbeitung auf das neue Format angepasst werden [48].

Um die Vorteile von Edge-Computing zu nutzen und dadurch entstehende Herausforderungen zu bewältigen, muss das System erweitert und angepasst werden. Anstelle der bisherigen Inferenz-API wird eine Service-API benötigt, die das Modell im Ganzen bereitstellt und gebündelte Datenpakete für das Monitoring empfängt (siehe Abbildung A.1). Damit die Daten nachträglich in die Zeitreihendatenbank eingefügt werden können, müssen sie mit Zeitstempeln und weiteren relevanten Informationen versehen sein. Während die Farbeigenschaften des Bildes weiterhin in der Cloud berechnet werden, können Metadaten wie die Latenz, der Sensor-Messwert und andere relevante Informationen nur lokal erfasst und dort zum Datensatz hinzugefügt werden. Die Clientkonfiguration muss um Datenpunkte wie Modellversion und Hardwareinformationen erweitert werden, sodass der zentrale Überblick über die aktiven Modelle und deren Performance sowie die Nachvollziehbarkeit und Reproduzierbarkeit aller Prozesse gewährleistet bleiben. Hierbei ist zu berücksichtigen, dass neue Modellversionen nur asynchron verteilt werden können, da für die Aktualisierung eine stabile Internetverbindung und eine vom Edge-Gerät ausgehende API-Anfrage erforderlich ist. Weitere Bestandteile des MLOps-Systems wie Dashboards, Trainingspipeline und Modellregister werden wie gewohnt in der Cloud betrieben.

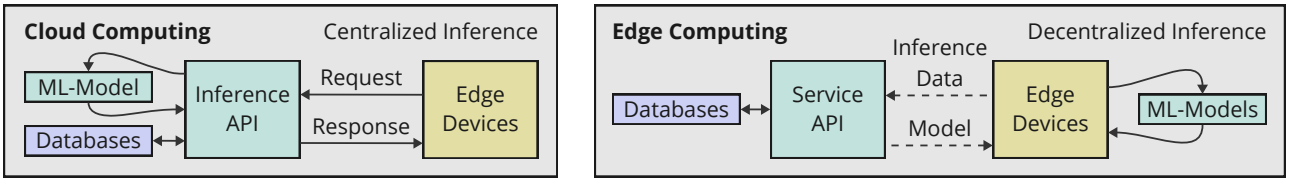



Abbildung A.1.: Zentrale und dezentrale Modellbereitstellung am Edge

Die Modellausführung auf Edge-Geräten bietet erhebliche Vorteile in Bezug auf Latenz, Datensicherheit und Unabhängigkeit von einer stabilen Internetverbindung, die jedoch mit Herausforderungen wie begrenzten Ressourcen, der Verwaltung dezentraler Geräte und der Sicherstellung einer konsistenten Modellversion einhergehen. Durch den Einsatz speziell optimierter Bibliotheken sowie Mechanismen zur Synchronisierung und Überwachung kann Edge-Computing in das bestehende MLOps-System integriert werden. Die zusätzlichen Möglichkeiten durch die hybride Nutzung von Cloud- und Edge-Computing stellen eine vielversprechende Weiterentwicklung des Systems dar, die leistungsfähigere und flexiblere ML-Anwendungen ermöglicht.

# B. Anwendung in eigenen ML-Projekten

Für die erfolgreiche Anwendung des entwickelten Systems in eigenen ML-Projekten sind mehrere Schritte erforderlich, die im Folgenden als Leitfaden aufgeführt sind. Ziel ist es, den Lesern die Möglichkeit zu geben, selbst praktische Erfahrungen zu sammeln und das System als Grundlage für ihre eigenen Projekte zu nutzen. Die Daten sind verfügbar unter:

GitHub-Repository mit statischem Code: `https://github.com/MBenediktF/mlops.git`
Datensätze zur Simulation: `https://mbenediktf.de/end-to-end-mlops/simdata`

## B.1. Einrichten des Repositories in eigener Umgebung

Um das entwickelte System für den Einsatz in eigenen Projekten zu nutzen, muss das Repository zunächst in die eigene Entwicklungsumgebung kopiert und als neues Repository initialisiert werden. Grundlegende Kenntnisse in der Arbeit mit *git* sind hierfür vorausgesetzt. Git und Docker müssen auf dem Entwicklungsrechner installiert sowie konfiguriert sein. Die Verwendung der Entwicklungsumgebung VSCode mit den Erweiterungen GitHub und Docker wird empfohlen. Folgende Schritte sind notwendig, um das Repository lokal einzurichten:

1. **Klonen des Repositories**
   - Navigation in das gewünschte Verzeichnis
   - Ausführen des Befehls `git clone -bare <repository-link>`
   - Wechsel in das geklonte Verzeichnis mit `cd <repository-name>.git`
2. **Initialisieren des eigenen Repositories**
   - Erstellen eines neuen Repositories auf GitHub
   - Push der geklonten Daten mit `git push -mirror <own-repository-link>`
3. **Anpassen der Konfiguration**
   - Öffnen der Umgebungskonfiguration in `.env`
   - Ändern der Zugangsdaten für enthaltene Services
   - Konfiguration der SMTP-Parameter für Email-Benachrichtigungen
   - Optional: Anpassen von Host, Ports und weiteren Parametern

**Wichtig:** Da in der Datei `.env` sensible Daten wie Passwörter und Zugangsdaten enthalten sind, dürfen diese Änderungen nicht in das Repository eingecheckt werden.

## B.2. Lokale Systemausführung während der Entwicklung

Das System kann nun mit `docker-compose up` lokal kompiliert, gestartet und getestet werden. Der Status einzelner Container wird in der Docker-Erweiterung von VSCode überwacht, mit der jeweils auch die Logs und Dateien eingesehen werden können. Die Infercence-API loggt ihre eingehenden Anfragen und generierten Ergebnisse zusätzlich in die Datei `inference.log`, welche im Volume `log_data` persistent gespeichert wird. Der Zugriff auf die UIs der Services erfolgt über die in der `.env` angegebenen Ports, die auch in der Docker-Erweiterung von VSCode angezeigt werden. Konkret stehen folgende Benutzeroberflächen zur Verfügung:

- **MLFlow**: Tracking von Experimenten, Verwalten von Modellen
- **Dagster**: Einsicht, Konfiguration und Ausführung von Pipelines, Assets und Zeitplänen
- **Grafana**: Datenanalyse mit Dashboards und Konfiguration von Alarmen
- **Config-UI**: Hinzufügen und Verwalten von Clients und Deployments
- **MinIO**: Einsicht der gespeicherten Dateien von MLFlow, Dagster und Inference-API

Während der Entwicklung können zusätzlich folgende Tools eingesetzt werden, welche durch ihre grafischen Oberflächen die Interaktion mit Programmbestandteilen erleichtern:

- **InfluxDB-Weboberfläche**: Einsicht in die Zeitreihendatenbank (Port 8086)
- **MySQL Workbench**: Einsicht und Bearbeitung der MySQL-Datenbank (Programm)
- **Postman**: Testen der Inference-API durch simulierte Anfragen (Programm)

Für ein tieferes Verständnis der Systemkomponenten empfiehlt sich vor der Anpassung auf individuelle Anforderungen das Ausprobieren aller Benutzeroberflächen und Tools. Als Leitfaden hierfür dient die Evaluation (Kapitel 4), in welcher der gesamte Prozess - von der ersten Messdatenaufnahme bis hin zur automatisierten Modellverbesserung - beschrieben wird. Die Hilfsanwendung `src/scripts/simulate_measurement.py` kann dabei genutzt werden, um das Edge-Gerät aus dem Anwendungsbeispiel zu simulieren und dafür Daten aus bestimmten Umgebungen an die Inference-API zu senden. Dafür wird ein neues Gerät in der Config-UI hinzugefügt, dessen Zugangsdaten zusammen mit dem Ordnernamen des Datensatzes im genannten Python-Skript eingetragen werden. Die Datensätze aus bestimmten Umgebungen müssen separat heruntergeladen und in `src/scripts/simdata` abgelegt werden. Alle Ergebnisse der Evaluation sind damit reproduzierbar und können als Vorlage für eigene Projekte dienen. Abbildung B.1 zeigt die Gesamtarchitektur des dynamischen Systems. Besonders die gezeigten Datenflüsse zwischen den Komponenten tragen zum Verständnis der Systemfunktionalität bei.

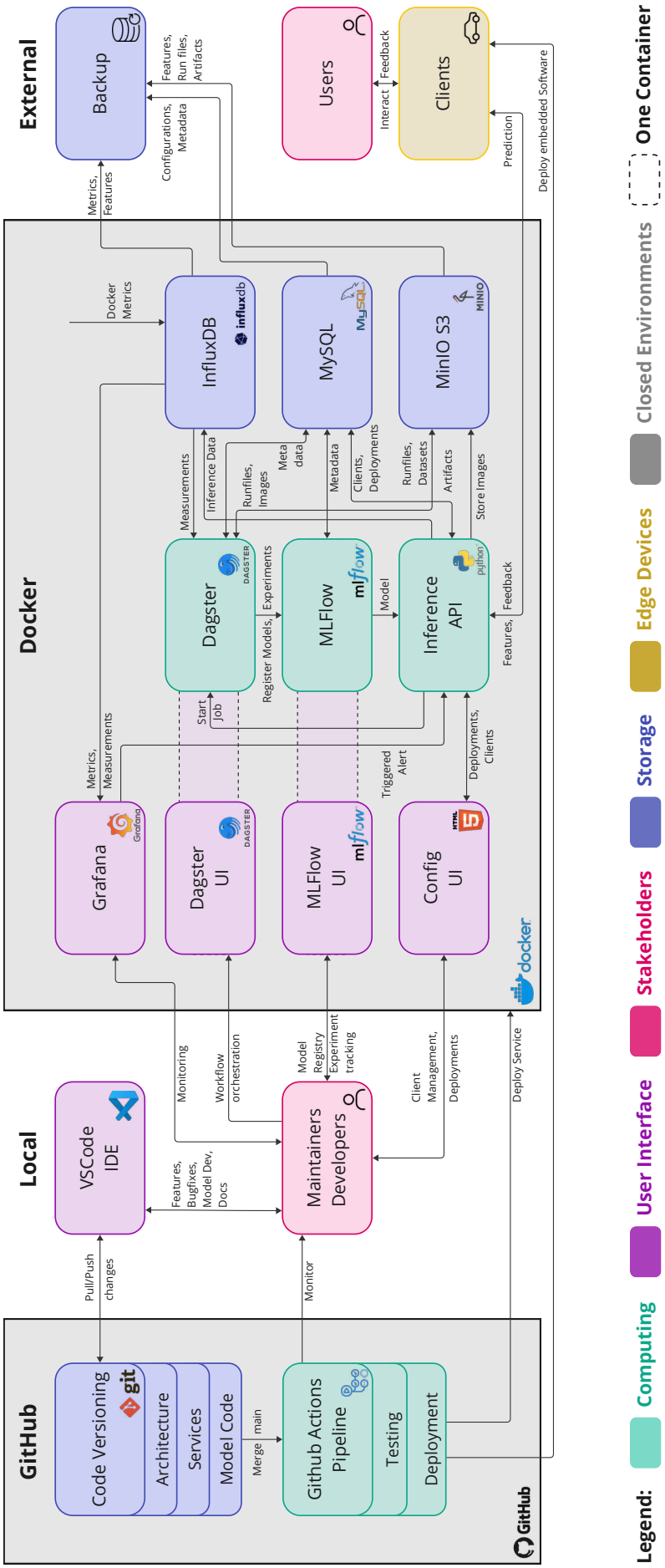



Abbildung B.1.: Systemarchitektur

## B.3. Anpassen des Systems an individuelle Anforderungen

Das System ist darauf ausgelegt, möglichst einfach an die individuellen Anforderungen eines konkreten ML-Projekts angepasst zu werden. In Kapitel 3.3 wurde der Aufbau des Repositories beschrieben und der statische Code der Komponenten vorgestellt. Die folgenden Schritte dienen als Unterstützung zur systematischen Anpassung der Komponenten und stellen zusätzliche Informationen für die Realisierung bereit.

### Schritt 1: Ändern von Datenstruktur und -verarbeitung

Abhängig von der Art der Daten einer konkreten Anwendung müssen die Datenstruktur und -verarbeitung angepasst werden. Da das System auf diesen Daten aufbaut, sollten diese Änderungen zuerst erfolgen. In der aktuellen Version ist das System auf Zeitreihendaten ausgelegt, die aus beliebigen Metadaten bestehen können und jeweils mit Bilddaten verknüpft sind. Die Interaktion mit den Datenbanken erfolgt durch die Hilfsfunktionen in `src/helpers/`, die sowohl von Modellcode und Dagster-Assets, als auch von der Inference-API genutzt werden. Ändert sich die Art der Daten grundlegend, müssen diese Funktionen angepasst und in den jeweiligen Services neu eingebunden werden. Dies beeinflusst auch die Kommunikation mit der Inference-API und die Extraktion von relevanten Metriken. Die Anpassungen der betroffenen Routen erfolgt in der Flask-App `src/inference/flask_app.py` und in den zugehörigen Python-Modulen `src/inference/`. Die Merkmalsextraktion findet ebenfalls in der Inference-API statt und erweitert den Datenbankeintrag um Kennzahlen, die das Bild beschreiben und zum Vergleich der Messdaten im Dashboard dienen. Um möglichst aussagekräftige Metriken zu erhalten, sollte das hierfür zuständige Python-Modul `src/inference/collect_image_characteristics.py` ebenfalls angepasst werden. Während der Entwicklung empfiehlt sich die Verwendung der grafischen Oberflächen der Datenbanken zur Inspektion und Anpassung der gespeicherten Daten sowie die Nutzung des Tools *Postman* zum Testen einzelner Routen der Inference-API.

### Schritt 2: Anpassen und Testen des modularen Modellcodes

Ist das System auf die neuen Datenstrukturen angepasst, kann mit der Entwicklung des statischen Modellcodes begonnen werden. Für die lokale Modellentwicklung steht das Jupiter-Notebook `model/model_development.ipynb` zur Verfügung, das auch als Vorlage und Dokumentation der einzelnen Python-Module für die Schritte der Trainingspipeline dient. Diese können im Ordner `src/model/` individuell angepasst werden, wobei die Schnittstellen zwischen den Modulen beibehalten werden müssen. Ist eine Änderung von Ein- und Ausgaben eines Moduls erforderlich, muss zuerst der Test `tests/model/` und dann der Aufruf im zugehörigen Dagster-Asset `src/dagster/assets` angepasst werden. Das Notebook ermöglicht den systematischen Test der Module sowie durch individuelle grafische Ausgaben die umfangreiche Auswertung der Rückgabewerte. Es dient als wichtigstes Werkzeug bei der Entwicklung von Modellcode.

## Schritt 3: Bearbeiten von Dashboards und Alerts

Die Dashboards und Alerts in Grafana basieren auf Metriken, welche durch die Inference-API aus den Messdaten extrahiert werden, sodass die Grafana-Konfiguration an neue Datenstrukturen angepasst werden muss. Auch wenn diese als Teil des statischen Codes unter `src/grafana/grafana.db` abgelegt ist, empfiehlt sich die Verwendung der grafischen Oberfläche von Grafana, die das Anordnen per Drag-and-Drop ermöglicht und übersichtliche Tools zur Visualisierung bereitstellt. Die `.db`-Datei ist als externes Volume mit dem Grafana-Container verknüpft und verändert sich dadurch, wenn der Container aktiv ist. Sofern relevante Anpassungen in Grafana vorgenommen wurden, die im Repository gesichert werden sollen, müssen die Änderungen an der Datei `grafana.db` eingecheckt werden. Neben den Dashboards sind auch die Alerts in Grafana zu konfigurieren, die aus Regeln (Alert Rules) und Alertmanagern bestehen. Der Alertmanager definiert die Ziele der Benachrichtigungen, zum Beispiel eine E-Mail-Adresse oder eine Webhook-URL, während die Regeln die Bedingungen für das Auslösen eines Alarms festlegen. Die Regeln sind in Gruppen organisiert und basieren auf einer Query, die benötigte Metriken zyklisch aus der Datenbank abfragt. Diese werden dann mit mathematischen Ausdrücken verarbeitet und dabei zum Beispiel mit Schwellenwerten verglichen, sodass der Alarm ausgelöst wird, wenn die Bedingung für die unter `pending period` festgelegte Zeit erfüllt ist. Wurden keine Anpassungen an der Grafana-Konfiguration vorgenommen, müssen durch den Betrieb entstandene Veränderungen der `grafana.db` auch nicht eingecheckt werden.

## Schritt 4: Hinzufügen weiterer Funktionalitäten

Zuletzt kann das System beliebig um neue Funktionen erweitert werden. Dabei ist es wichtig, dass dies auf modularer Basis durchgeführt wird, um die grundlegende Struktur beizubehalten. Alle Änderungen sind ausführlich zu dokumentierten und in das Versionskontrollsystem einzuchecken. Es ist darauf zu achten, dass DevOps-Konzepte wie das Test Driven Development und CI/CD konsequent umgesetzt und die Codequalität durch statische Codeanalyse und Code-Reviews sichergestellt werden. Alle Umgebungsvariablen sind zentral in der `.env` anzulegen und von dort in die Services einzubinden.

Die Konfiguration weiterer Dagster-Assets und -Jobs erfolgt im Ordner `src/dagster/`, dessen Inhalte durch Anweisungen im `Dockerfile` in den Container eingebunden werden. Für neu hinzugefügte Elemente ist zudem die Aufnahme in die Datei `src/dagster/definitions.py` erforderlich, die von Dagster eingelesen wird. Das Erweitern der Inference-API um neue Routen erfolgt in der Flask-App `src/inference/flask_app.py`. Hinzugefügte Unterordner sind auch hier in das `Dockerfile` einzubinden. Die Config-UI ist in der Datei `src/config_ui/index.html` definiert und über Routen der Inference-API mit den Datenbanken verbunden.

Um Änderungen zu kompilieren, wird der Befehl `docker-compose up --build` ausgeführt.

## B.4. Bereitstellen des Systems für den Produktivbetrieb

Ist das System an die individuellen Anforderungen des Projekts angepasst und in der lokalen Umgebung funktionsfähig, kann es automatisiert zuerst in eine dedizierte Testumgebung und dann für den Produktivbetrieb bereitgestellt werden (siehe Kapitel 3.3.4). Die statische Codeanalyse, die Unit-Tests, das Kompilieren, das Deployment in die Testumgebung und die Integrationstests werden in der CI/CD-Pipeline `.github/workflows/deploy_to_development.yml` durchgeführt, die automatisch beim Erstellen eines Merge-Requests auf den `main`-Branch ausgelöst wird. Die Umgebungsvariablen, die bei der lokalen Ausführung aus der `.env` geladen wurden, werden nun durch je ein GitHub-Environment für die Test- und Produktivumgebung ersetzt. Dabei werden alle vertraulichen Daten wie Passwörter und Zugangsdaten als Secrets hinterlegt, sodass sie nicht von Dritten eingesehen werden können. Das Environment enthält zusätzlich Informationen über das Deployment-Ziel, also den Server, auf dem die Anwendung bereitgestellt werden soll. Das konkrete Vorgehen ist abhängig von der gewählten Infrastruktur. Im einfachsten Fall wird eine sichere Konsolenverbindung (SSH) zum Server hergestellt, um die erforderlichen Daten zu übertragen und die Anwendung dort zu starten. Erst nachdem die Funktionalität in der Testumgebung sichergestellt wurde, dürfen die Änderungen auf den `main`-Branch übertragen werden, um das System durch die automatisierte Ausführung der Pipeline `.github/workflows/deploy_to_development.yml` in die Produktionsumgebung auszurollen.

Der produktive Betrieb bringt mehrere Herausforderungen mit sich, die in Testumgebungen oft nur eingeschränkt simuliert werden können. Eine zentrale Anforderung ist die Skalierbarkeit des Systems, um Lastspitzen oder eine steigende Anzahl von Nutzern zuverlässig zu bewältigen. Dies erfordert eine dynamische Ressourcenverwaltung, die beispielsweise durch Container-Orchestrierung mit Kubernetes oder das Nutzen von Cloud-Computing umgesetzt werden kann. Ein weiterer entscheidender Aspekt ist der Datenschutz. Sensible Daten müssen sowohl während der Übertragung als auch bei der Speicherung geschützt werden, was den Einsatz von Verschlüsselungsmechanismen, Zugriffsrichtlinien und regelmäßigen Sicherheitsüberprüfungen erforderlich macht. Zudem müssen gesetzliche Vorgaben wie die Datenschutz-Grundverordnung (DSGVO) eingehalten werden. Auch die Verfügbarkeit und Fehlertoleranz des Systems spielen im produktiven Betrieb eine entscheidende Rolle. Ausfallzeiten durch Überlastung oder Qualitätseinbußen durch die Wahl der falschen Deploymentstrategie können kostspielig sein und das Vertrauen der Nutzer beeinträchtigen. Damit Probleme schnell erkannt und behoben werden können, müssen neben den Modellmetriken auch die Auslastungsmetriken des Systems überwacht und regelmäßige Backups erstellt werden.

Das entwickelte System bietet eine solide Grundlage, die für den produktiven Einsatz angepasst und erweitert werden kann. Neue Features, Verbesserungen oder Anregungen sind willkommen und können gerne in GitHub beigetragen werden.

# C. Ergebnisse der praktischen Evaluation

## C.1. Fotos der Umgebungen

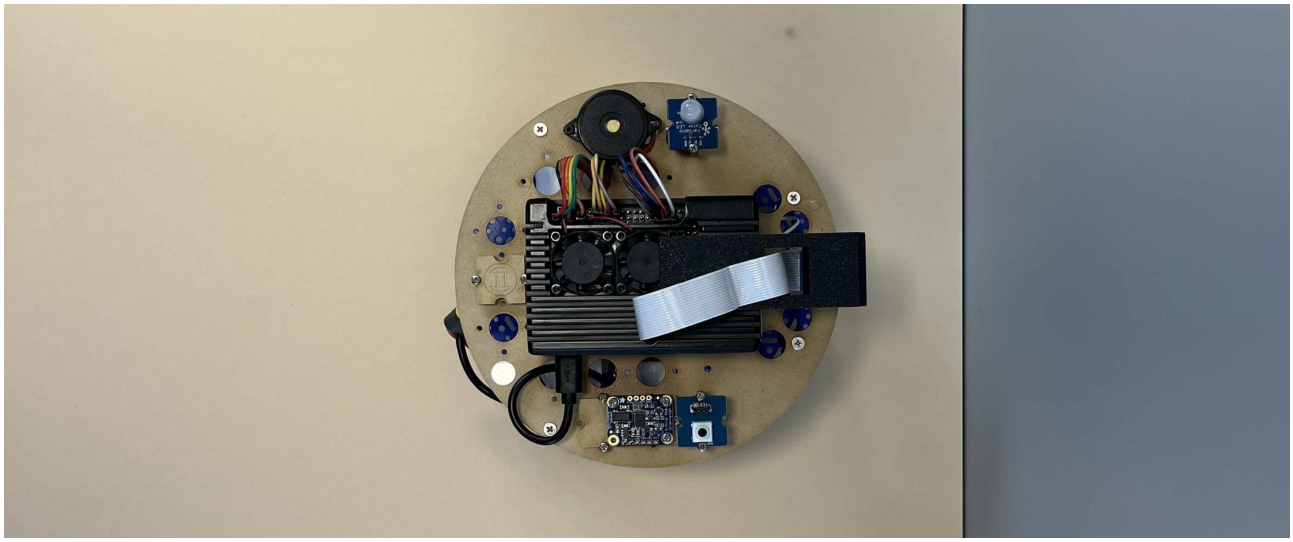


Abbildung C.1.: Umgebung Beige

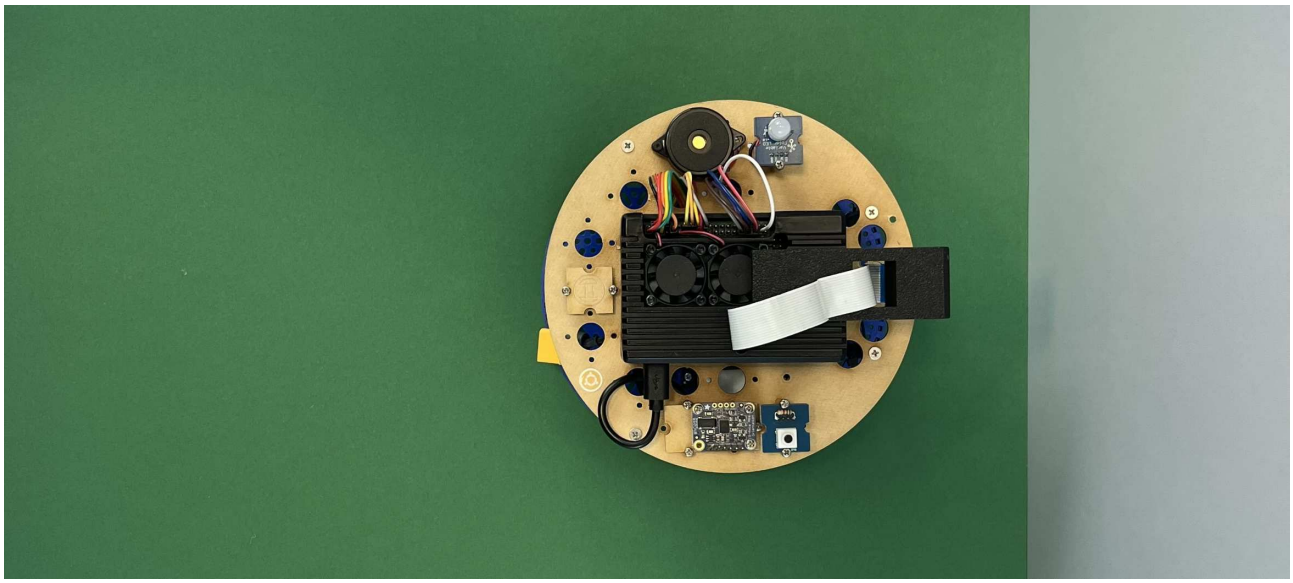


Abbildung C.2.: Umgebung Grün

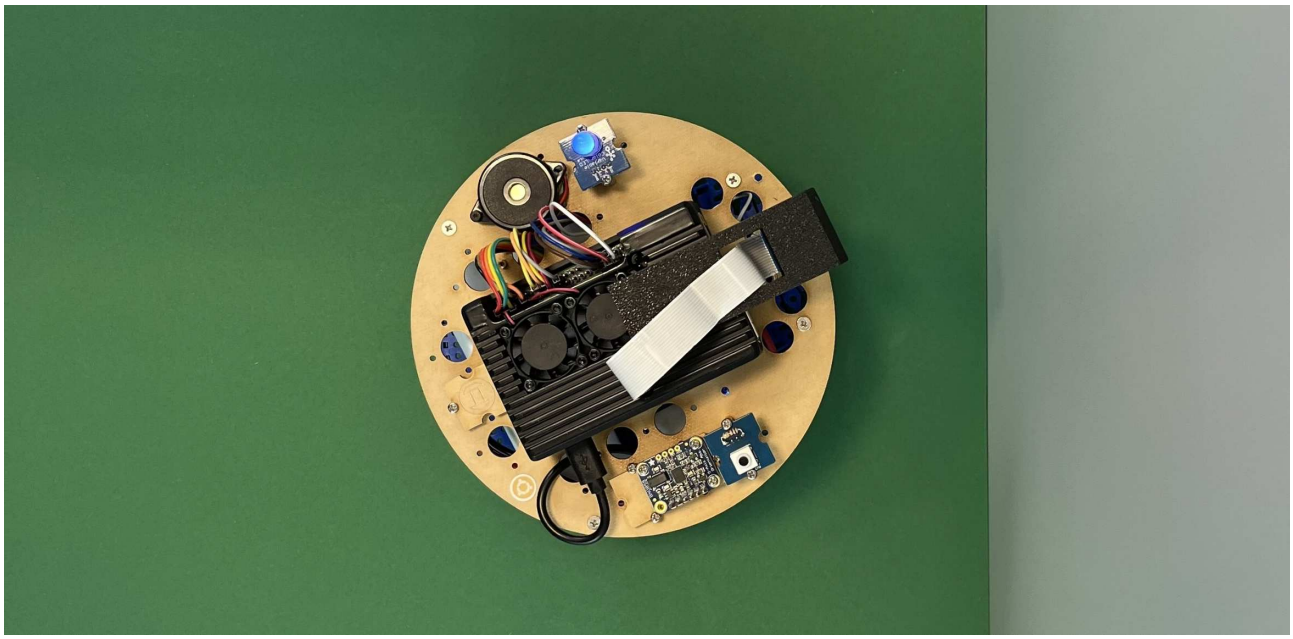


Abbildung C.3.: Umgebung Grün, schräge Anfahrt

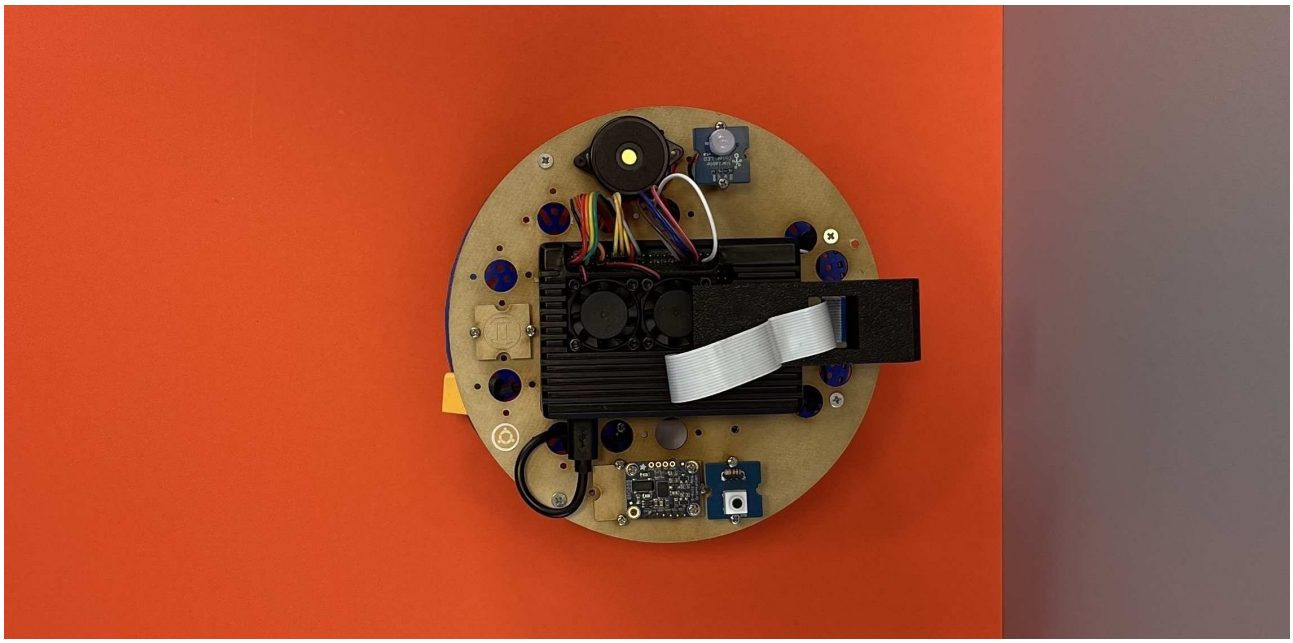


Abbildung C.4.: Umgebung Rot

## C.2. Grafana-Dashboards mit Messreihen

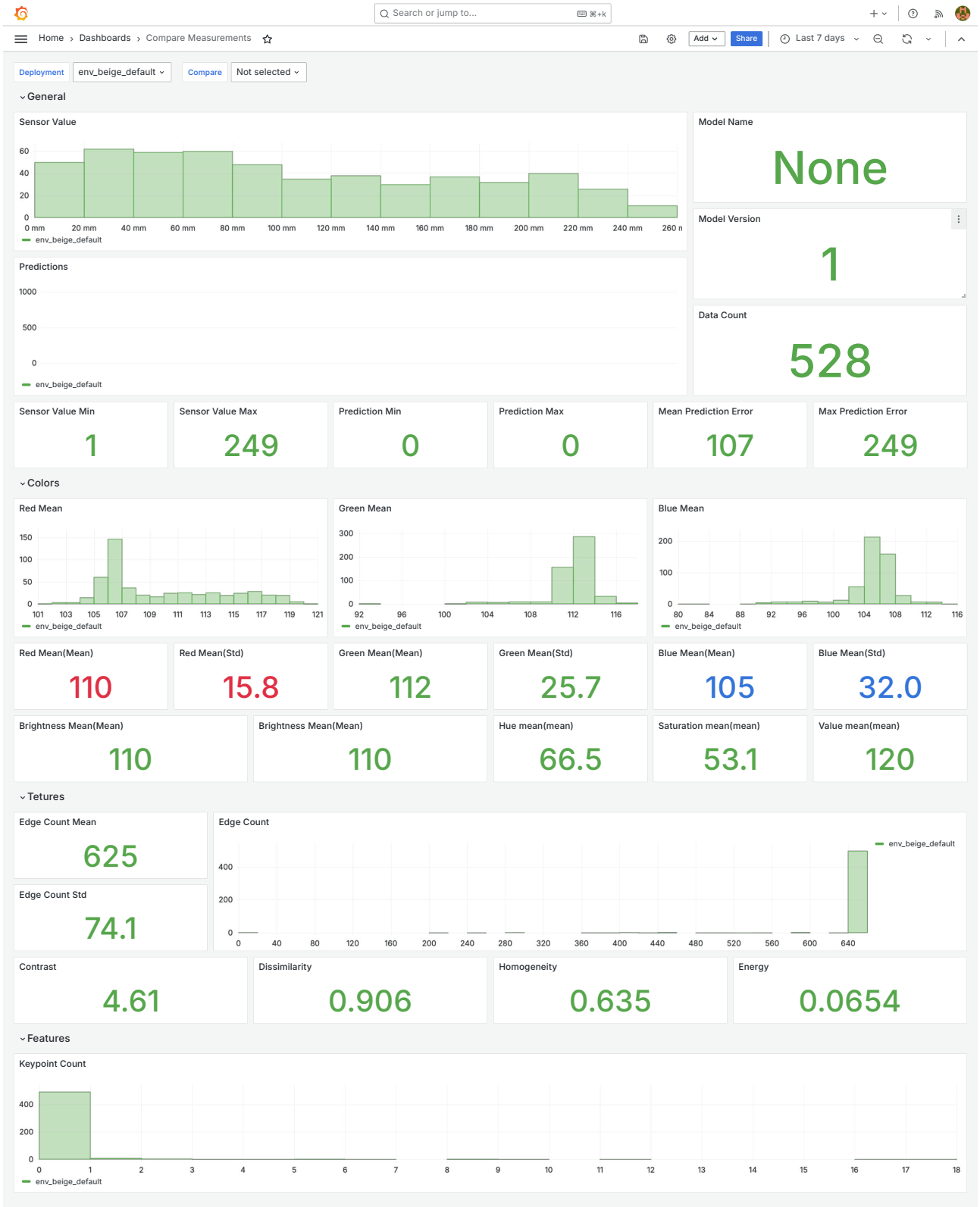



Abbildung C.5.: Initiale Messung in beiger Umgebung, ohne Modell

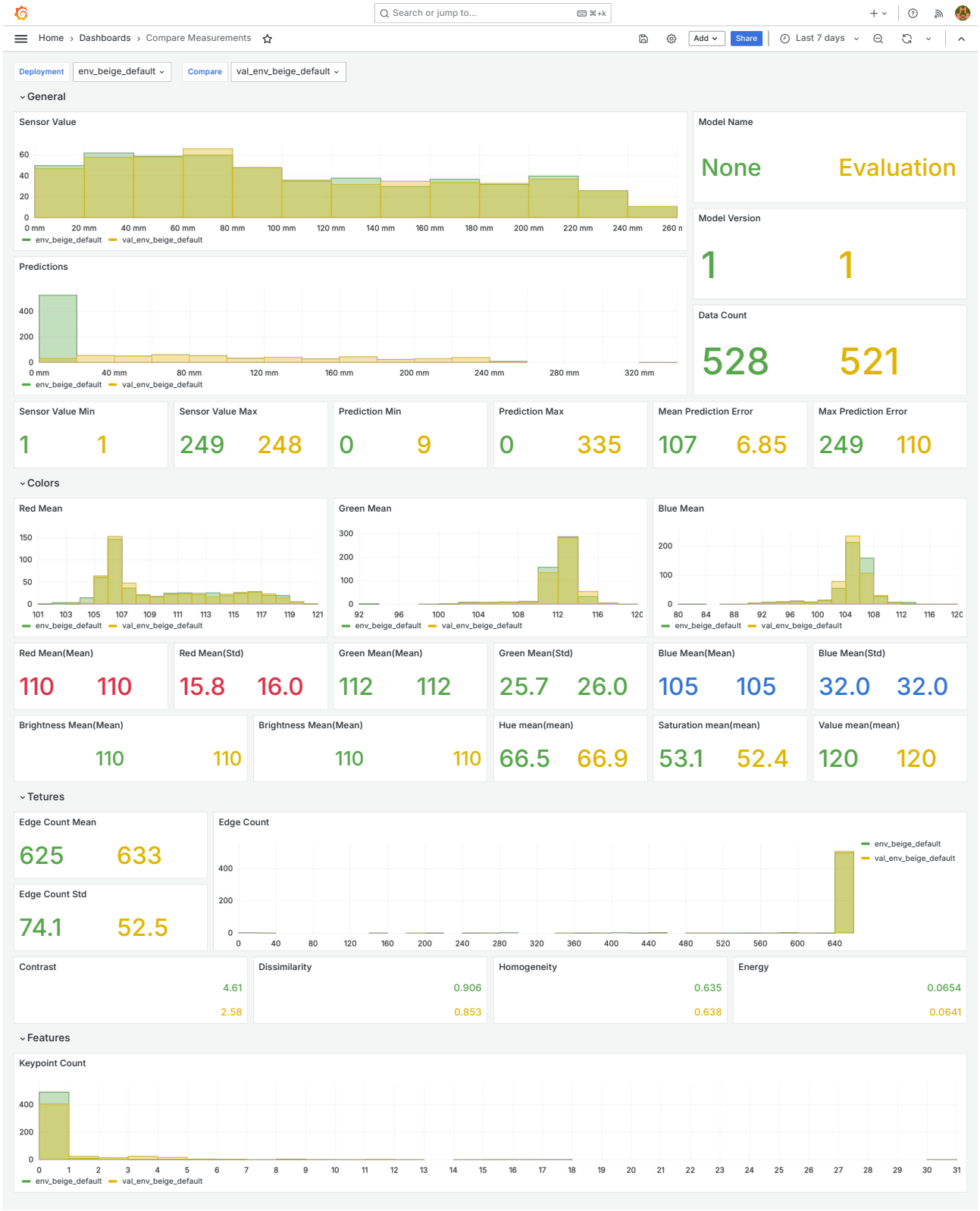



Abbildung C.6.: Messung in beiger Umgebung, nach Training

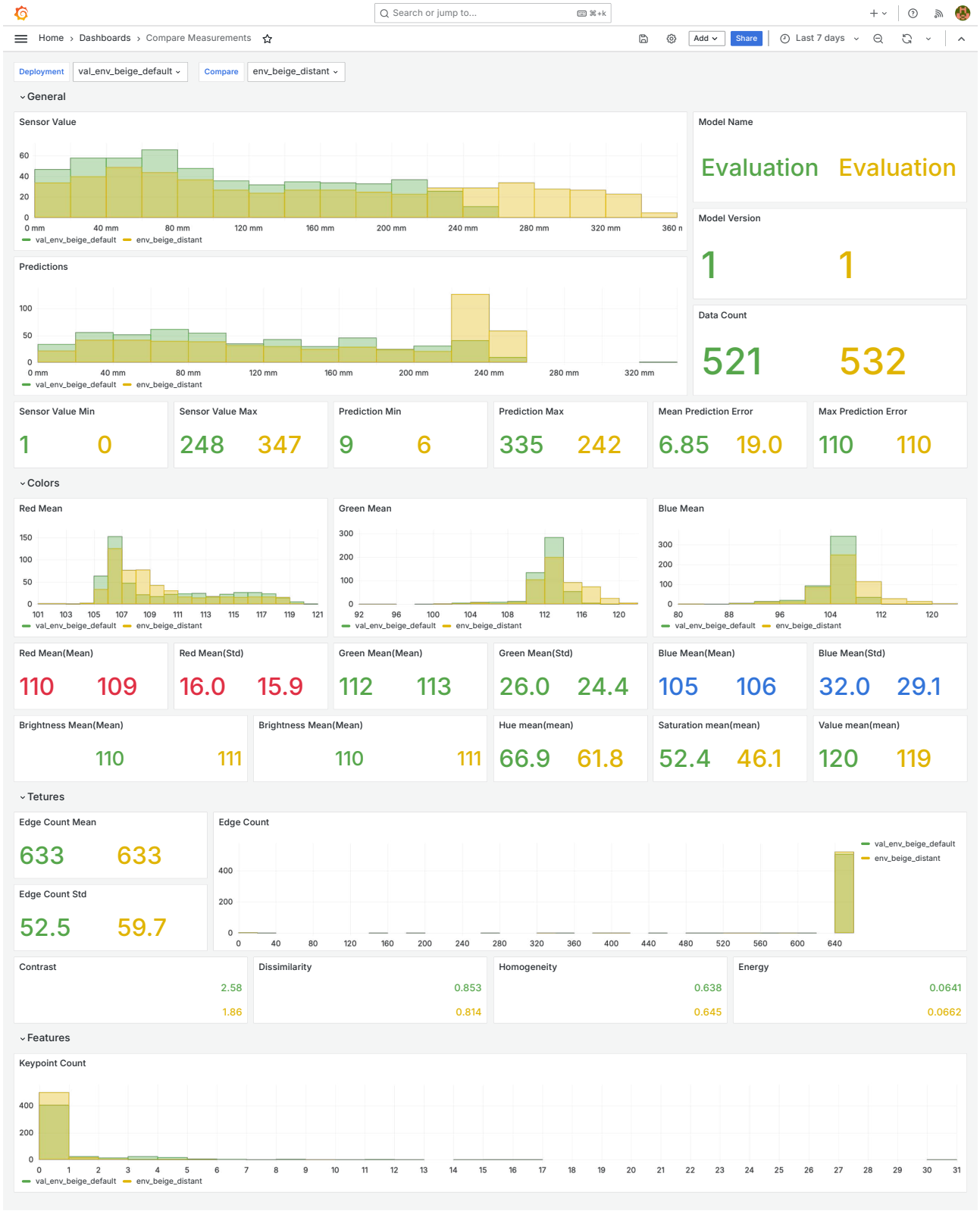



Abbildung C.7.: Messung mit erweitertem Messbereich in beiger Umgebung, vor Neutraining

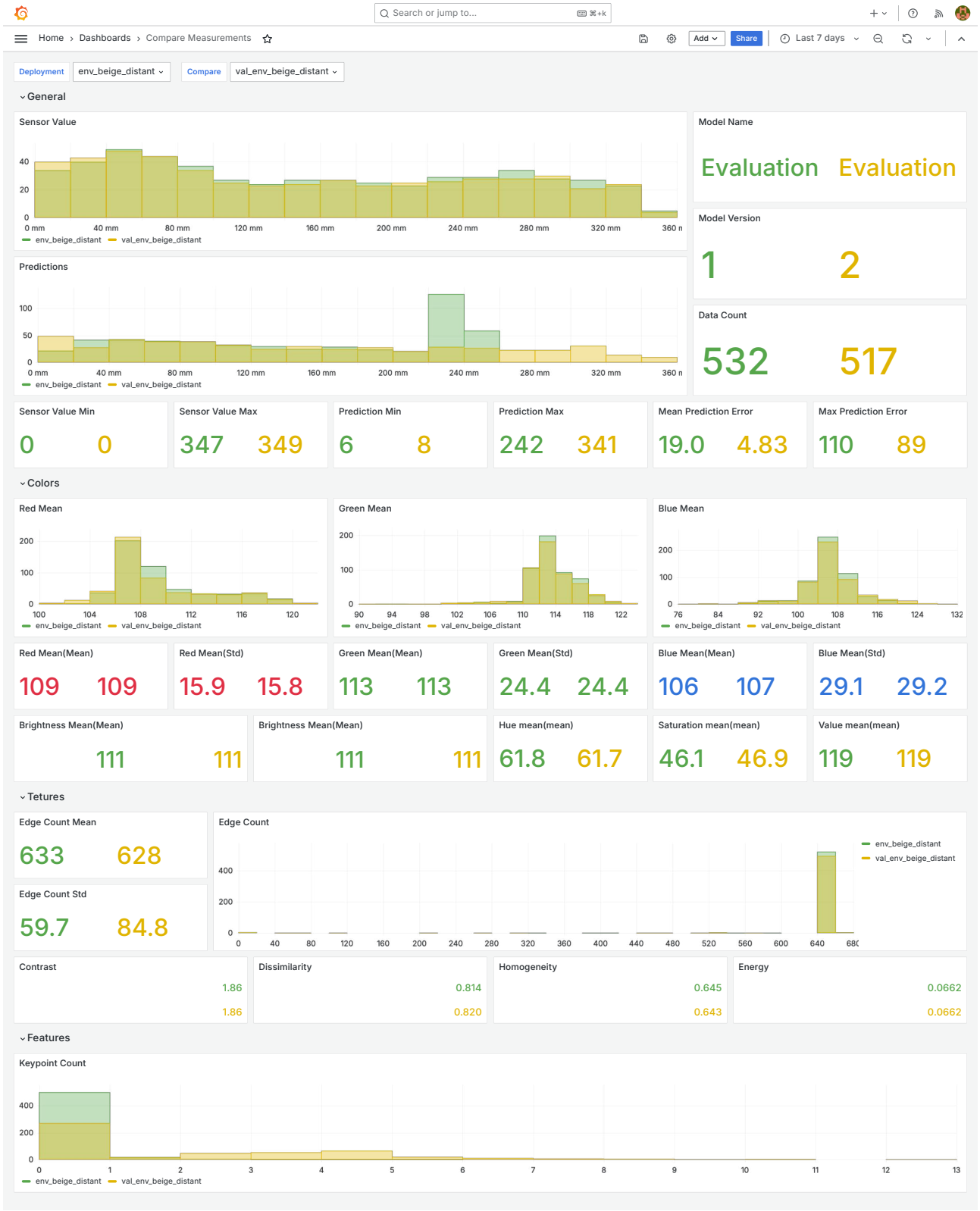



Abbildung C.8.: Messung mit erweitertem Messbereich in beiger Umgebung, nach Neutraining

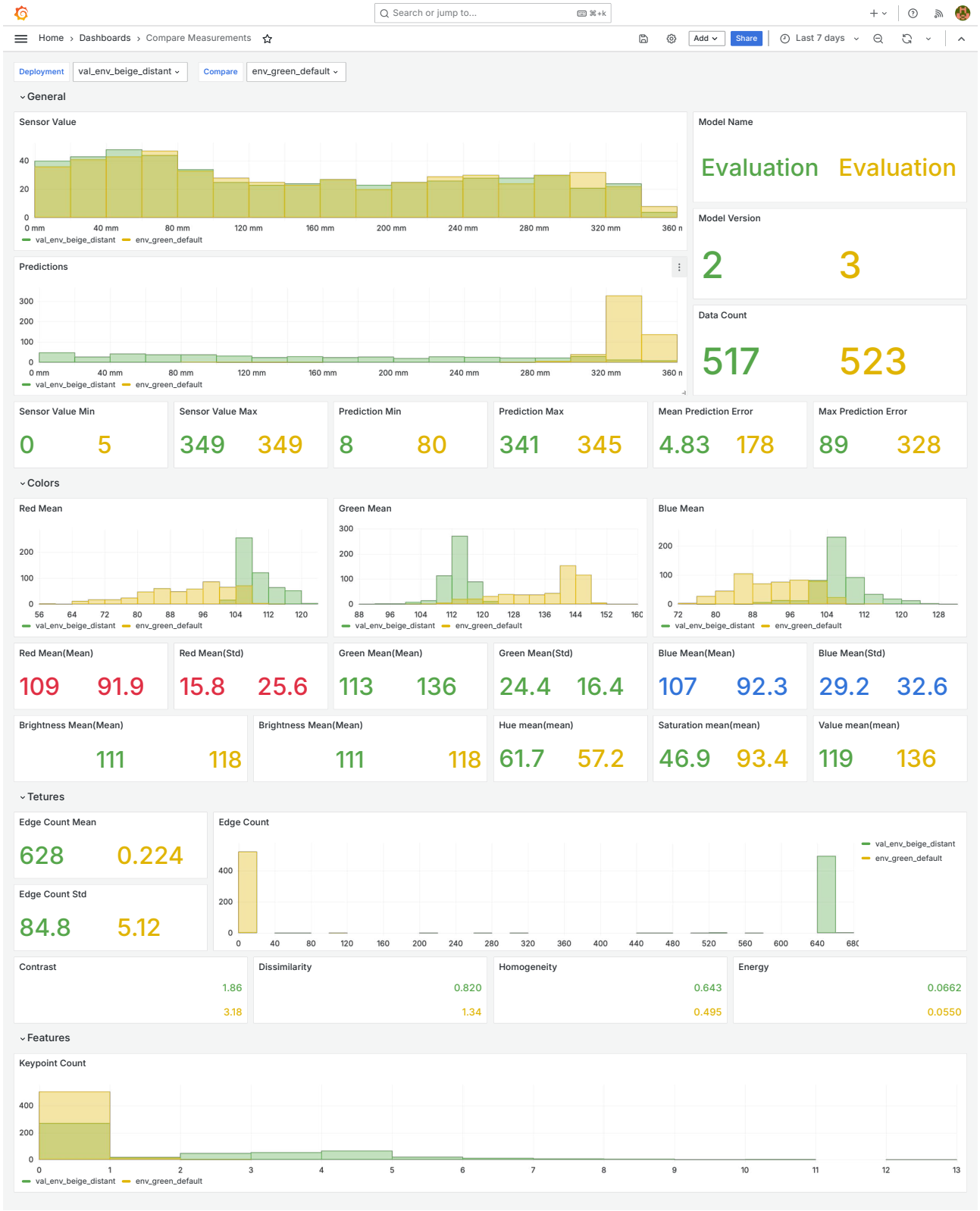



Abbildung C.9.: Messung in grüner Umgebung vor Neutraining

Hinweis: Modellversion 2 und Modellversion 3 sind identisch und liefern gleiche Vorhersagen.

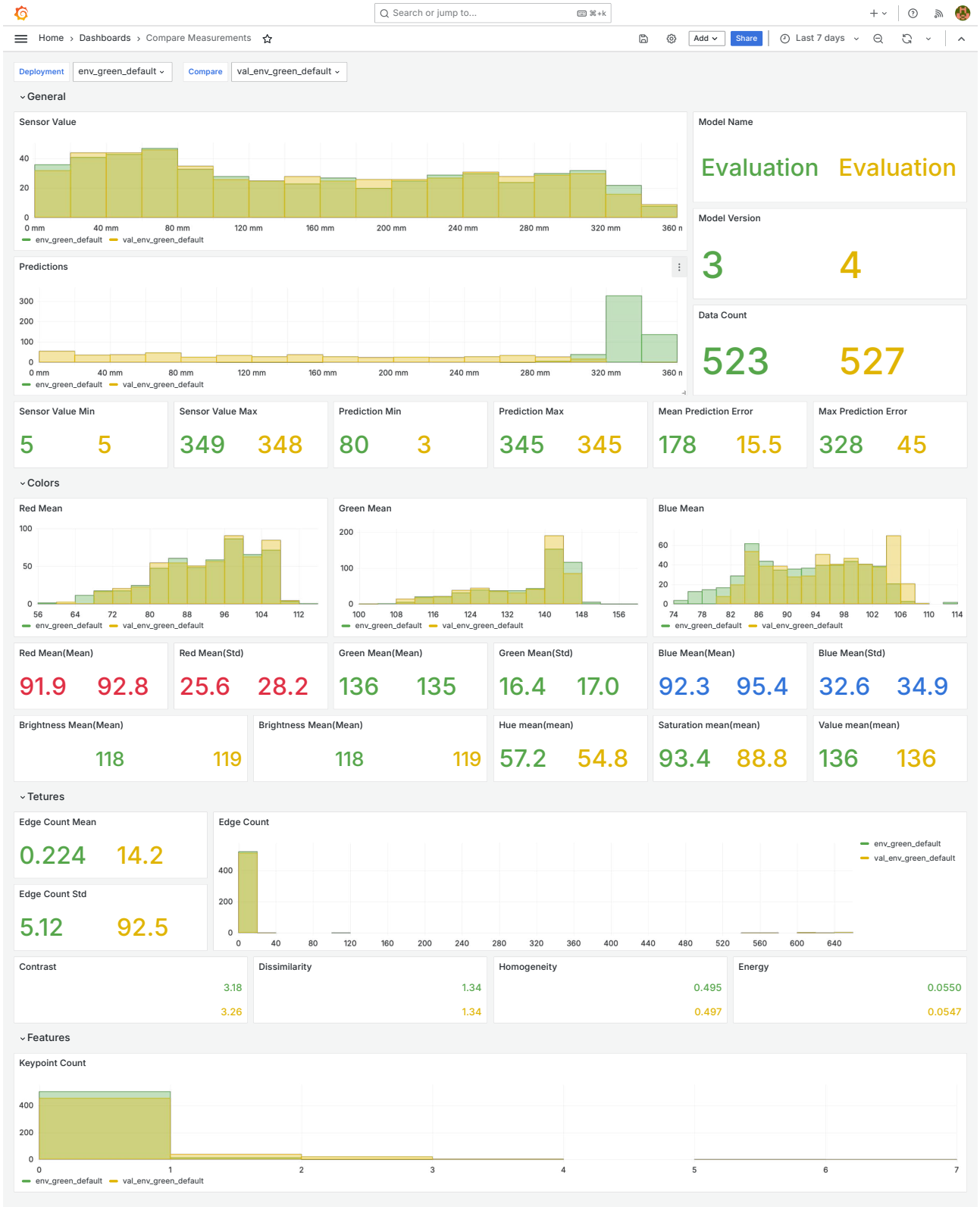



Abbildung C.10.: Messung in grüner Umgebung nach Neutraining

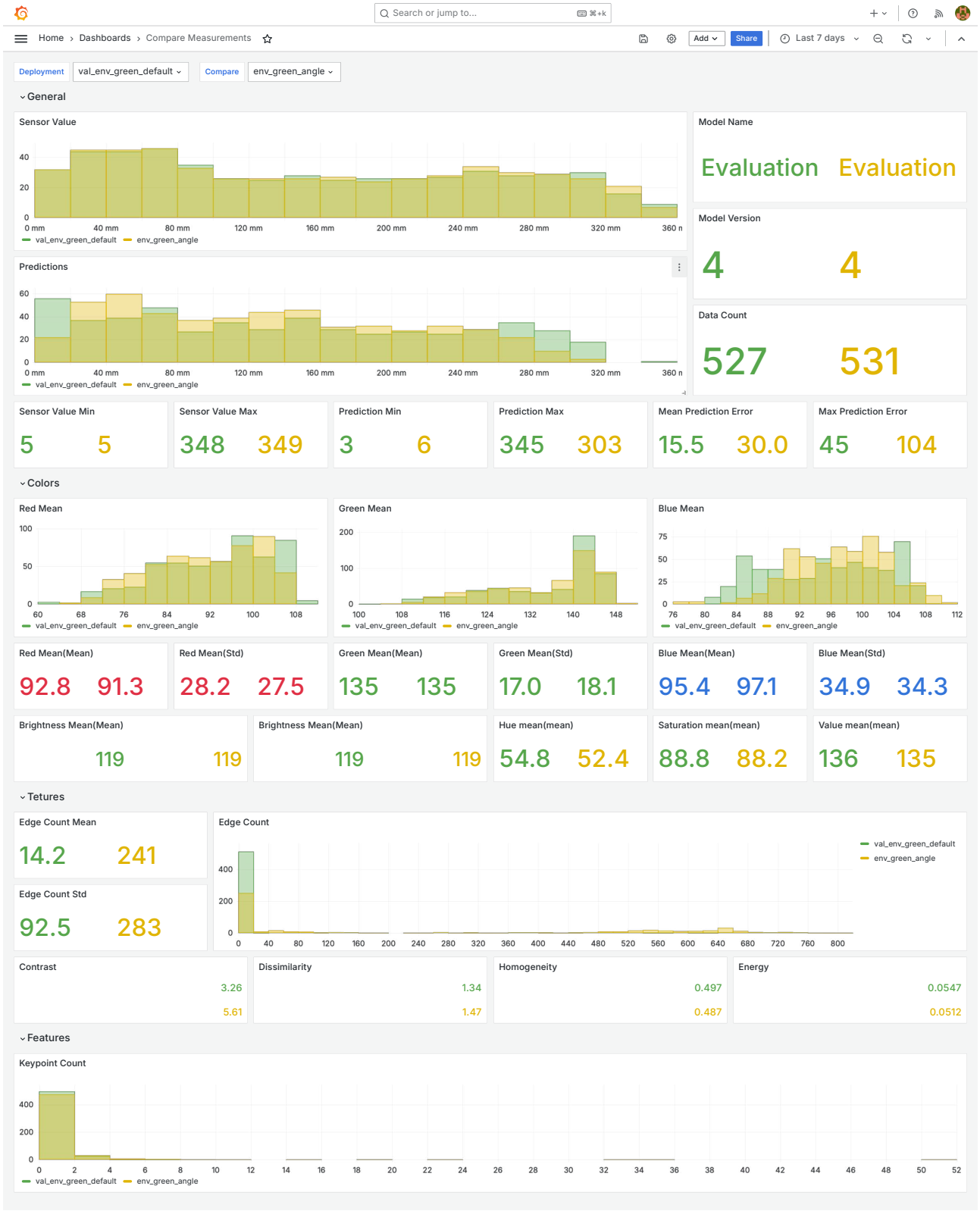



Abbildung C.11.: Messung mit verändertem Winkel in grüner Umgebung, vor Neutraining

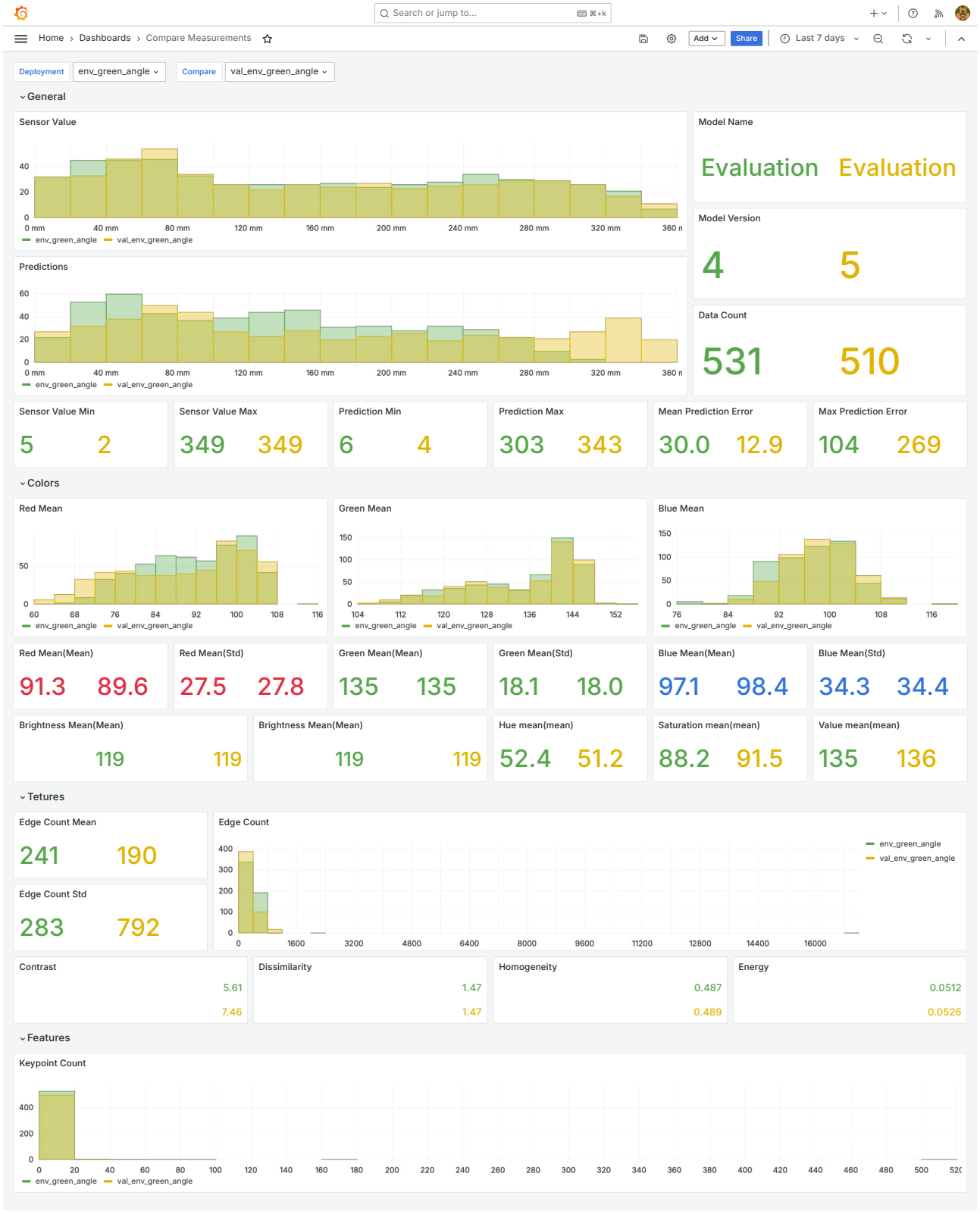



Abbildung C.12.: Messung mit verändertem Winkel in grüner Umgebung, nach Neutraining

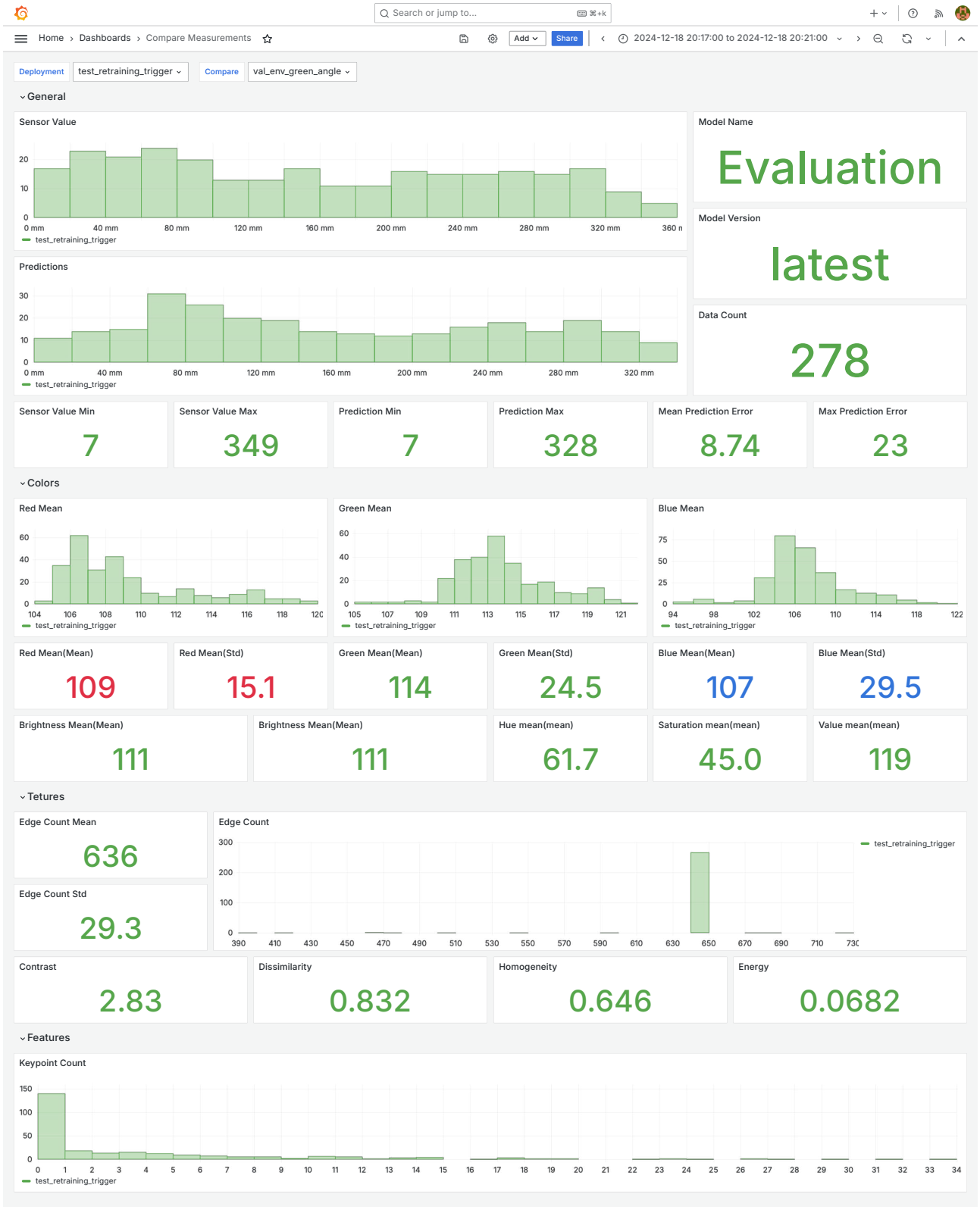



Abbildung C.13.: Messung zur automatisierten Modellanpassung, vor Umgebungsänderung

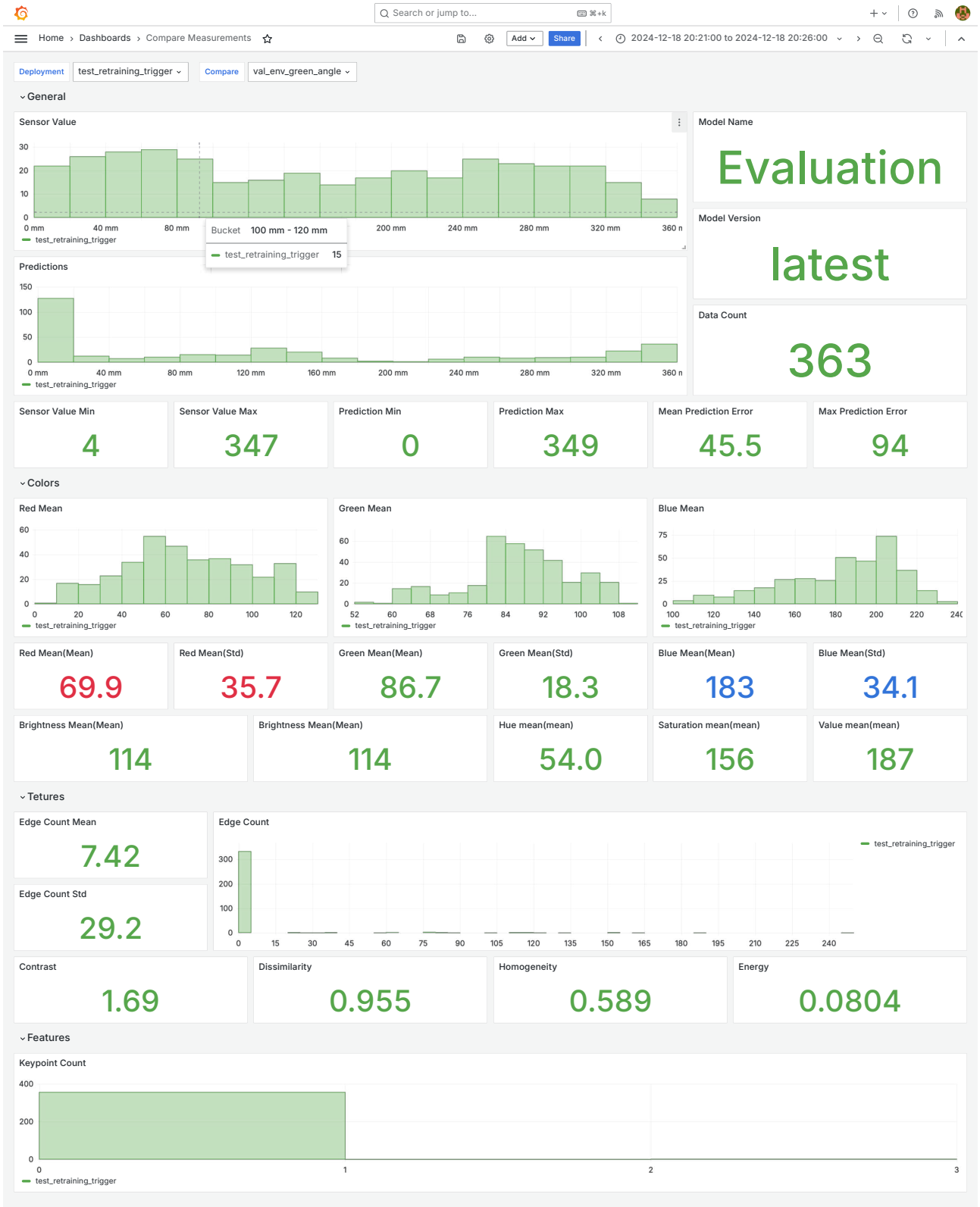



Abbildung C.14.: Messung zur automatisierten Modellanpassung, nach Umgebungsänderung

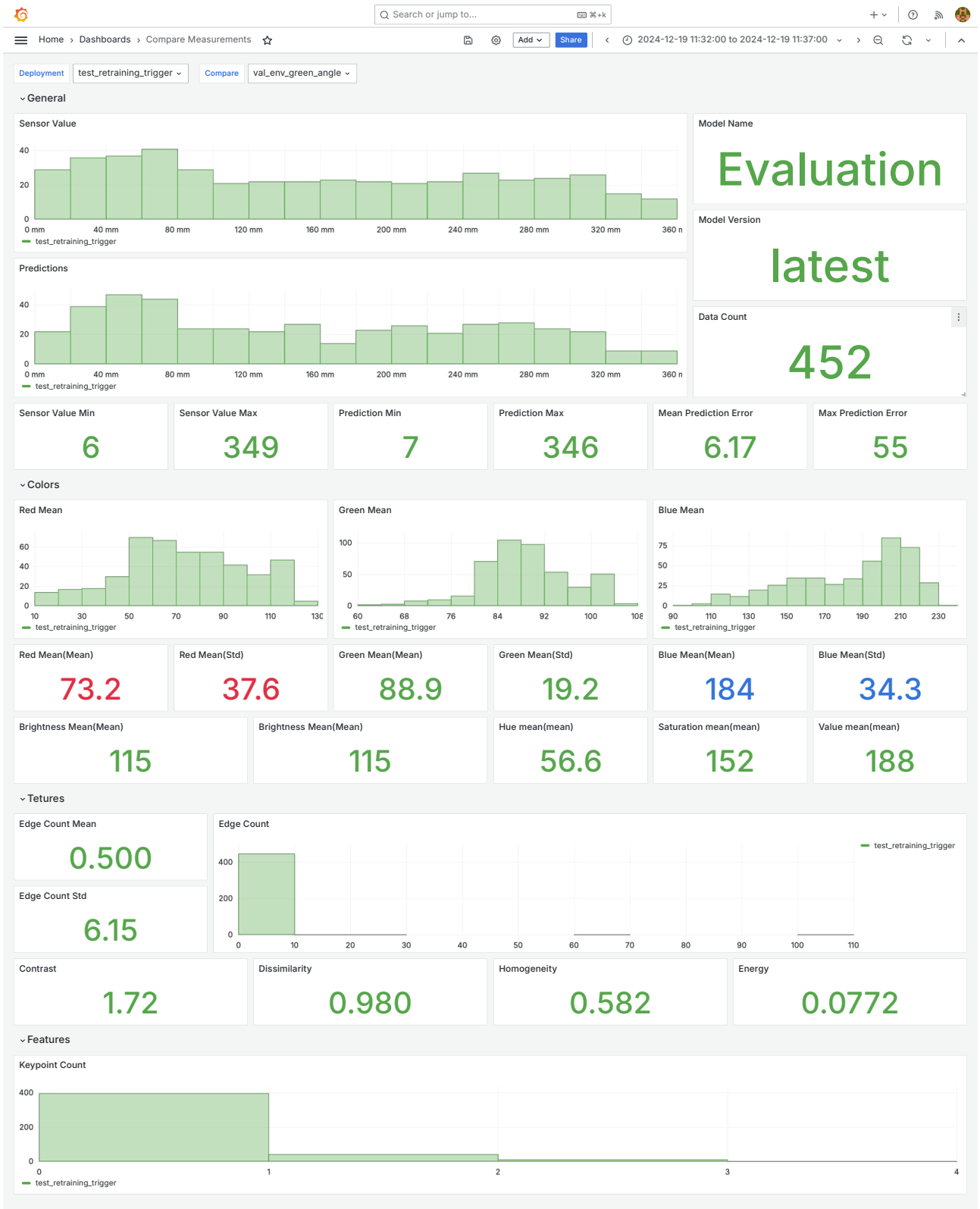



Abbildung C.15.: Messung zur automatisierten Modellanpassung, nach Neutraining

## C.3. Fokus auf bestimmte Datenpunkte

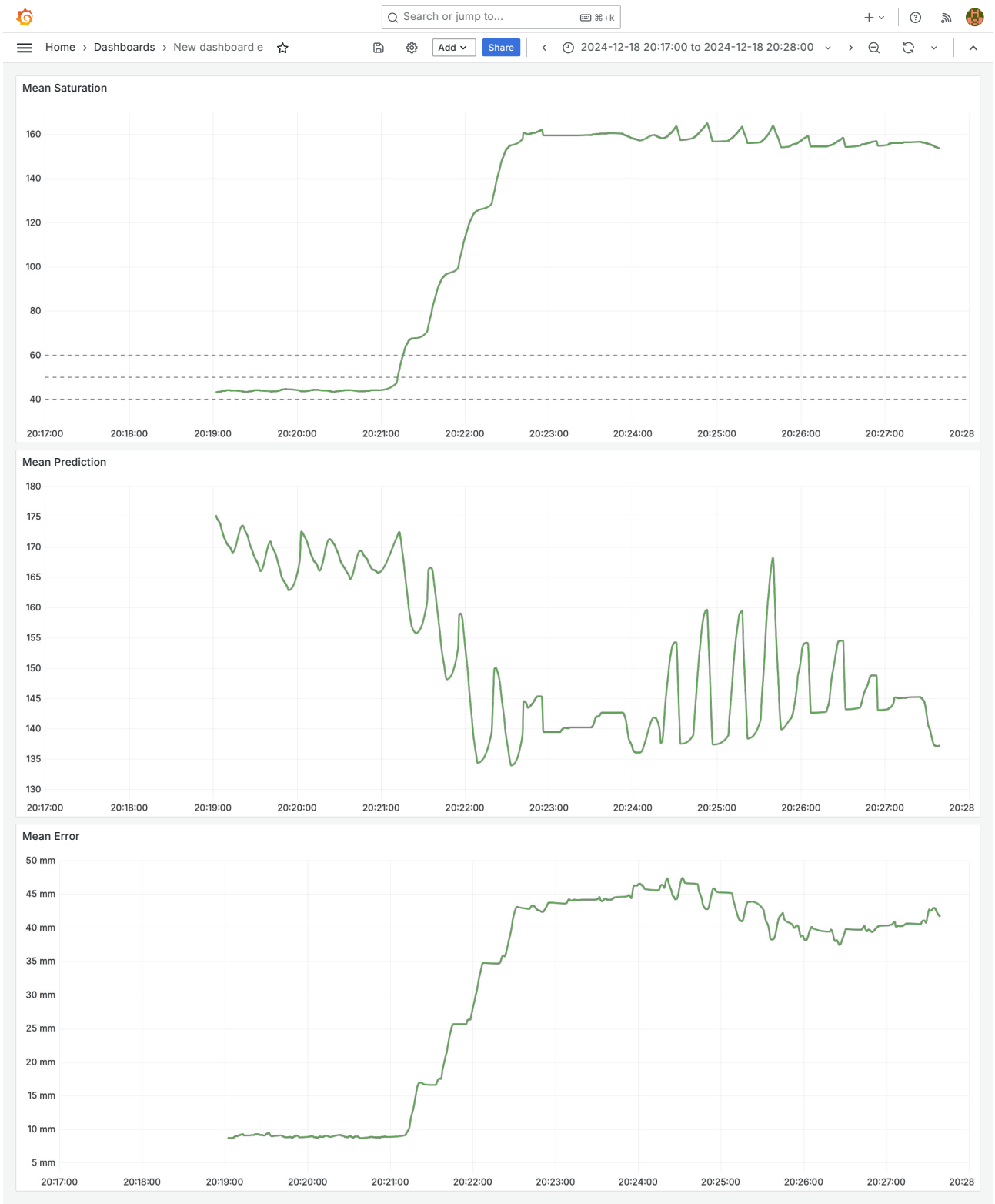


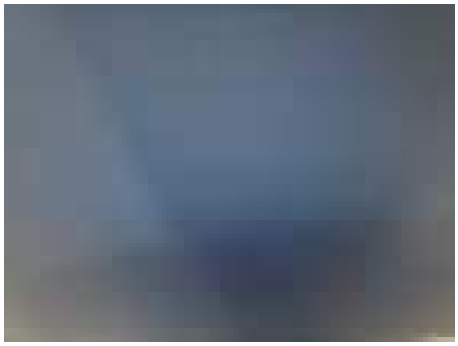

Abbildung C.16.: Zeitliche Veränderung von Sättigung, Vorhersage und Error durch Drift

## C.4. Einblick in die Trainingsdaten

env_beige_default, 235 mm

env_beige_distant, 41 mm

env_beige_default, 323 mm



env_beige_default, 8 mm

env_green_default, 167 mm

env_green_angle, 149 mm

test_retraining_trigger, 149 mm

test_retraining_trigger, 302 mm

Abbildung C.17.: Auswahl an Trainingsdaten mit Herkunft und Labels

# C.5. Auszüge von Daten in Systemen und Diensten

## 2: Get the data ready

In this step, the data will be imported and preprocessed.

### Download Dataset

The dataset has to be imported using the submodule `model/import_data.py`.

```python
from create_dataset import create_dataset
from helpers.s3 import enable_local_dev as s3_enable_local_dev
from helpers.influx import enable_local_dev as influx_enable_local_dev

# Enable local development (use local path instead of internal docker path)
s3_enable_local_dev()
influx_enable_local_dev()

measurements = ['env_beige_default']
img_size = (None, 75, 100, 3)
dataset_id = "5e1530ab-ba18-42fa-8897-590f6ea833e5"
images, labels, uids = create_dataset(measurements, img_size)
max_input_value = 350
```

Python

### Inspect dataset

```python
print("Number of images: ", images.shape[0])
print("Labels from ", np.min(labels), " to ", np.max(labels), "\n")

def plot_image(image, label):
    plt.imshow(image)
    plt.title(label)
    plt.show()

# Plot first image
plot_image(images[0], f"distance: {labels[0]}")
```

Python

```
Number of images:  528
Labels from  1  to  249
```

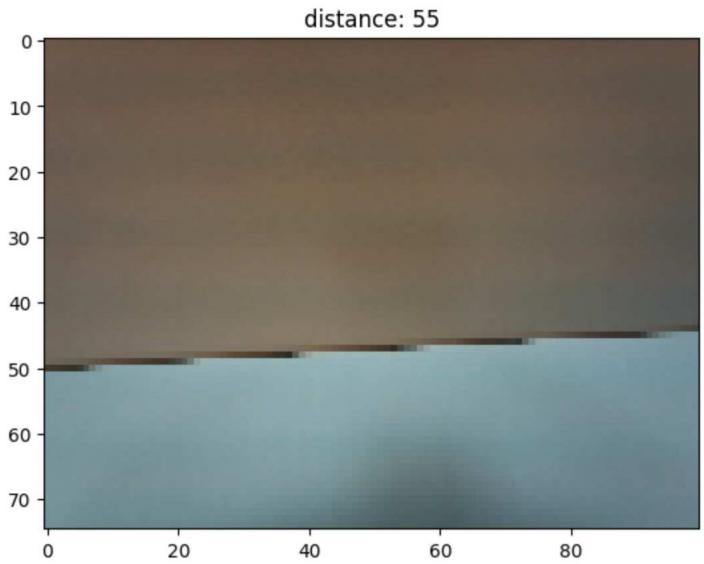



Abbildung C.18.: Auszug aus Jupiter-Notebook zur Modellentwicklung

# Config-UI MLOps-Research

## Clients

| Name | UID | Actions |
|---|---|---|
| car_lab _blue | d1c03ea4-b770-4675-87d5-9cdd9386beb8 | Delete |
| New Client | | Create |

## Deployments

| Name | UID | Model | Version | Records | Status | Actions |
|---|---|---|---|---|---|---|
| env_beige_default | 972f0aea-ddd0-461d-ac6c-44331a7980df | None | - | 528 | Inactive | Set Active |
| val_env_beige_default | 7854b44e-cc75-46cd-b9ab-e4ff24fd6102 | Evaluation | 1 | 521 | Inactive | Set Active |
| env_beige_distant | 43e63f79-022a-4b7c-b7a6-90398c00a123 | Evaluation | 1 | 532 | Inactive | Set Active |
| val_env_beige_distant | 0fa3806d-b5d7-425d-a68c-0bfe7d4e0a41 | Evaluation | 2 | 517 | Inactive | Set Active |
| val_env_beige_default+distant | 19dba8d4-8338-4112-b6f7-14e79a7ecf26 | Evaluation | 3 | 521 | Inactive | Set Active |
| env_green_default | 56a27e02-9e08-4c01-9300-36625d2272c2 | Evaluation | 3 | 523 | Inactive | Set Active |
| val_env_green_default | 5066aec4-3a40-43c7-be0e-676a8b169b41 | Evaluation | 4 | 527 | Inactive | Set Active |
| env_green_angle | aa38b0dd-4a31-4731-9fb9-e33b07af72c3 | Evaluation | 4 | 531 | Inactive | Set Active |
| val_env_green_angle | 90598d22-34d0-441e-a3ba-4433f40baf7e | Evaluation | 5 | 510 | Inactive | Set Active |
| val_env_green_angle_multi | b27880f3-1d3d-4b2e-a736-db27274e897e | Evaluation | 5 | 624 | Inactive | Set Active |
| test_retraining_trigger | 090dac92-80af-4a47-bf16-99f81e32d4f7 | Evaluation | latest | 2990 | Active | Set Active |
| New Deployment | | Model Name | 1 | | | Create |

Abbildung C.19.: Darstellung der Messreihen und Geräte in der Config-UI


Abbildung C.20.: Darstellung der Container und Volumes in der Docker-Extension

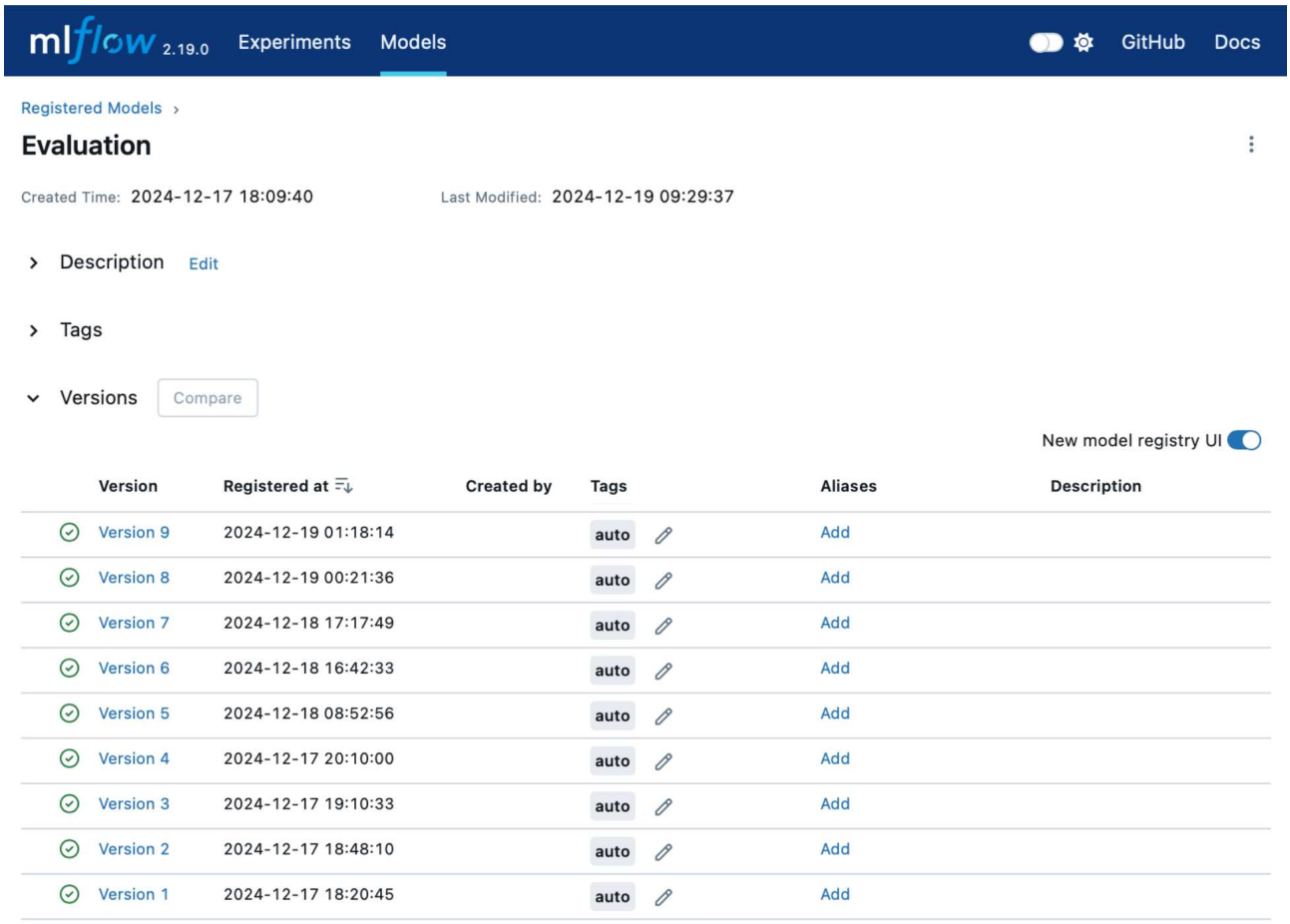



Abbildung C.21.: Darstellung des Modells mit Versionen im MLFlow-Modellregister

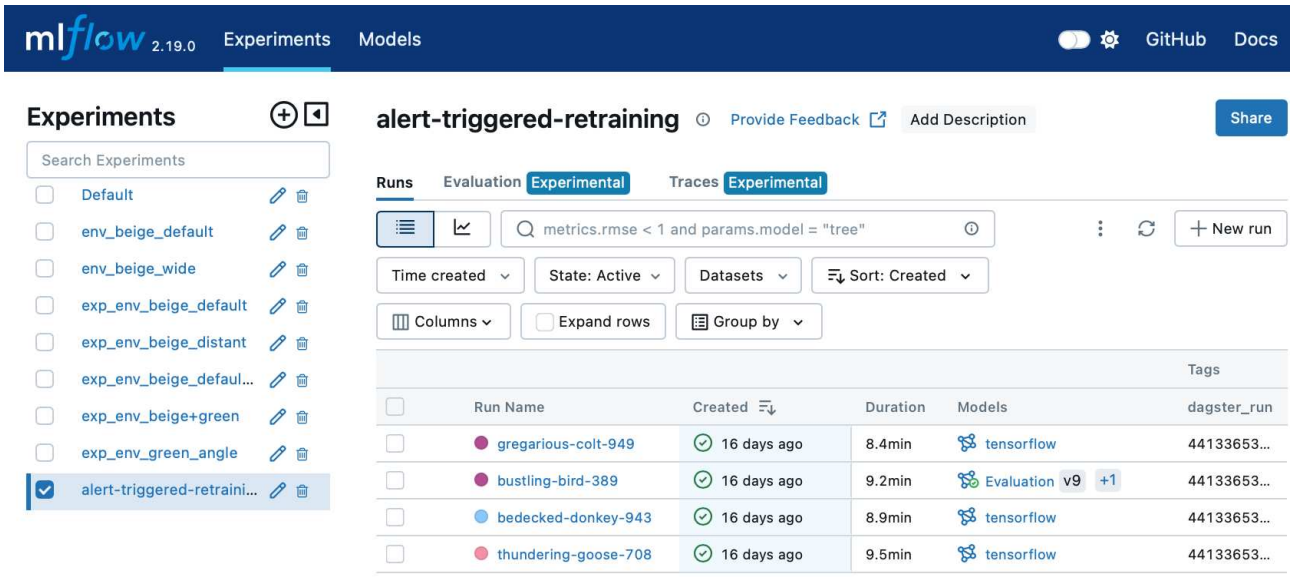



Abbildung C.22.: Darstellung der Experimente in der MLFlow-UI

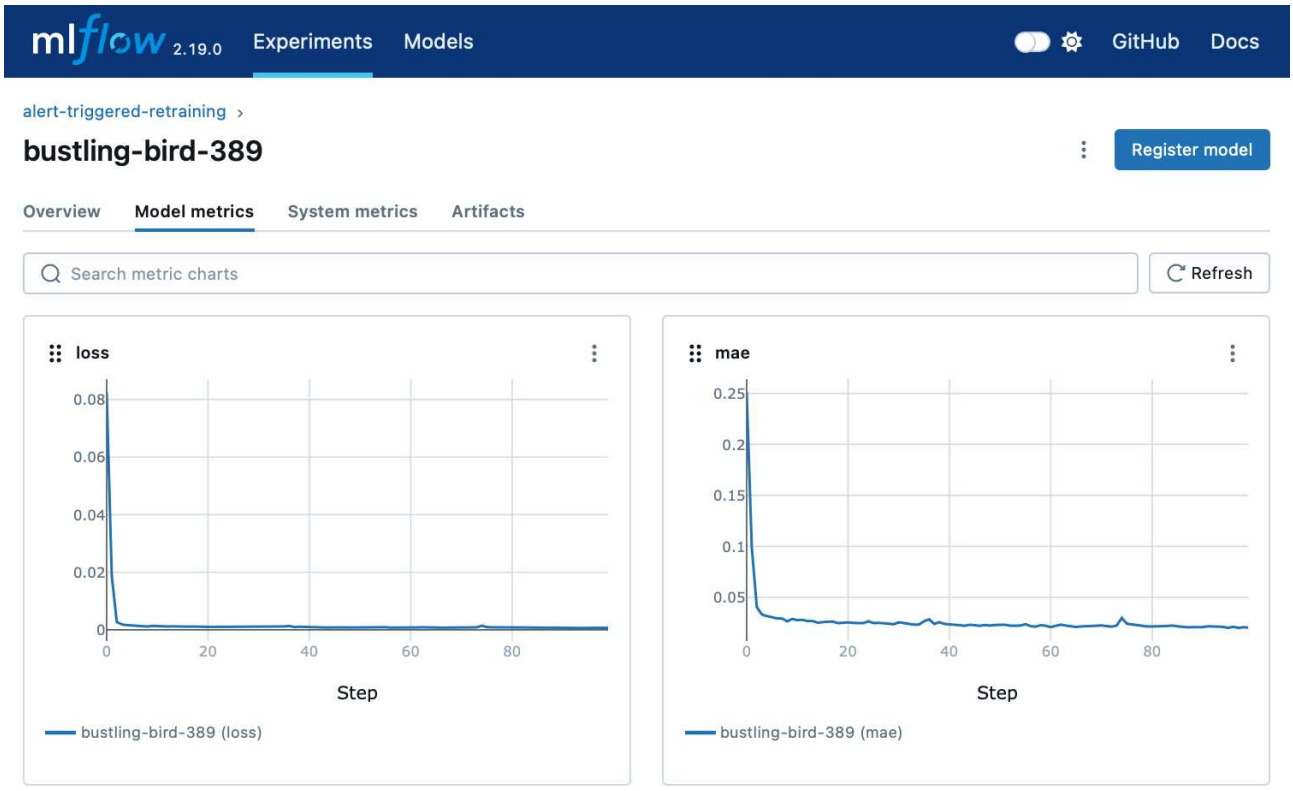



Abbildung C.23.: Darstellung der Trainingshistorie in der MLFlow-UI

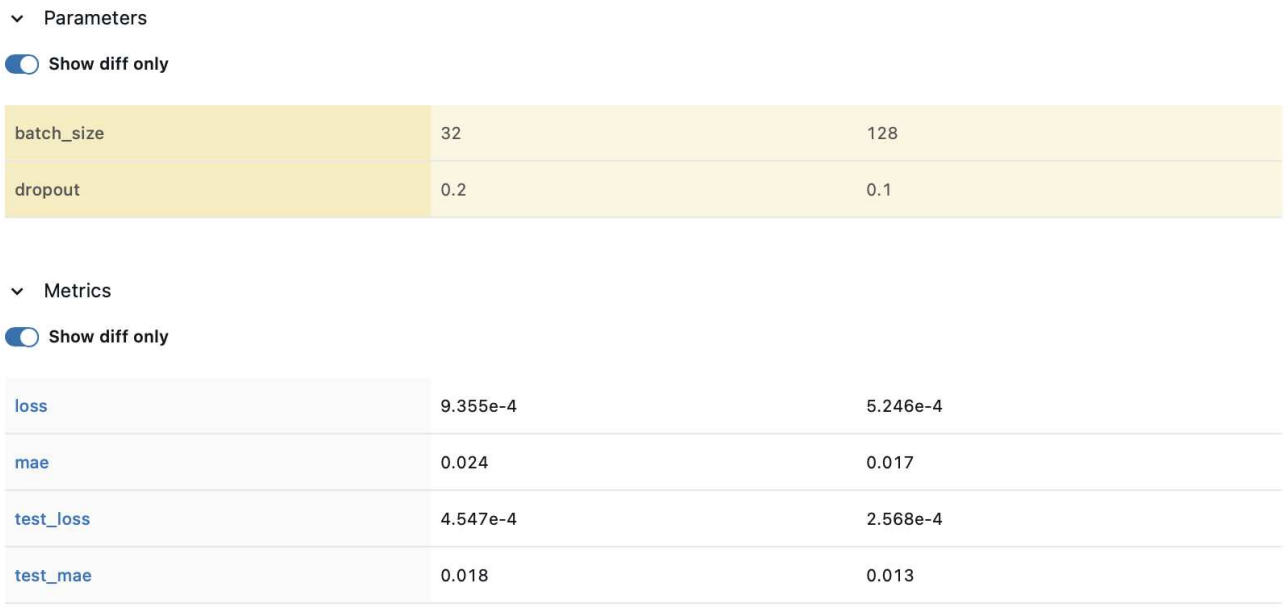



Abbildung C.24.: Vergleich von zwei Runs in der MLFlow-UI

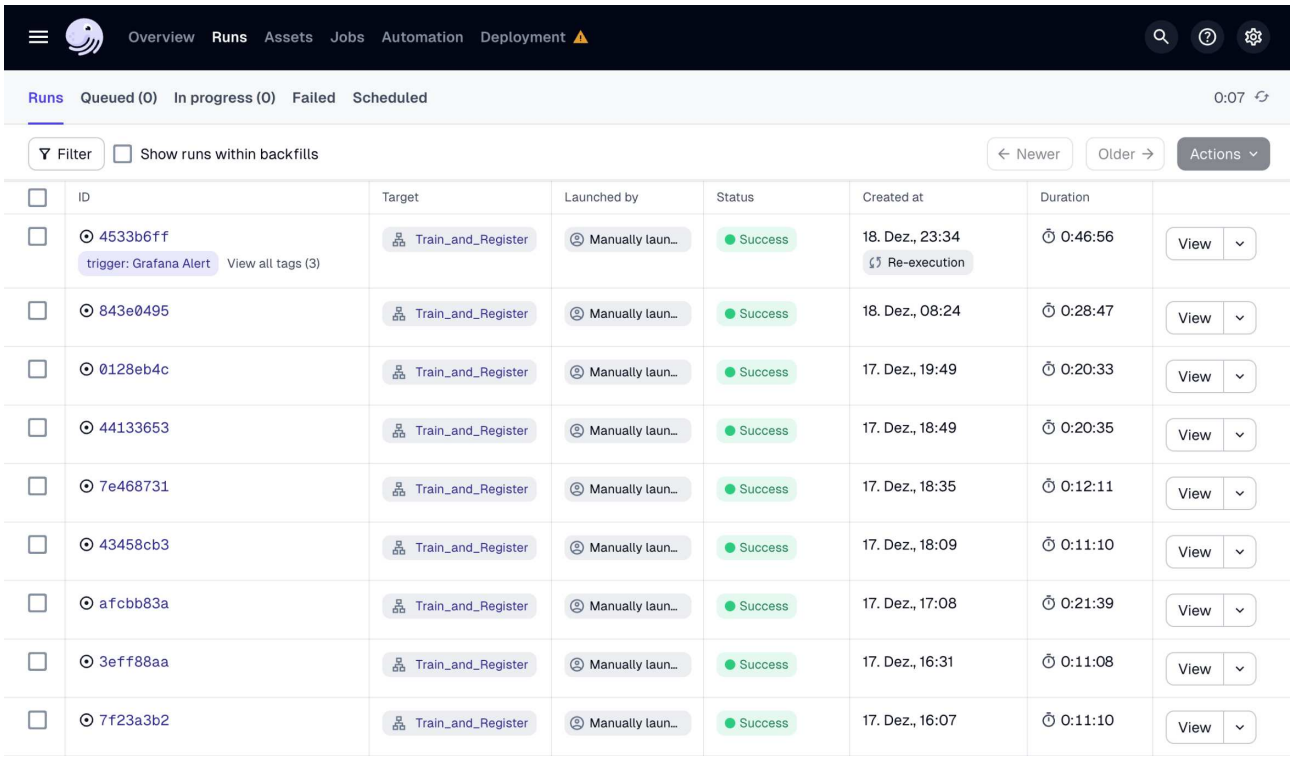



Abbildung C.25.: Darstellung der Dagster Runs in der Dagster-UI

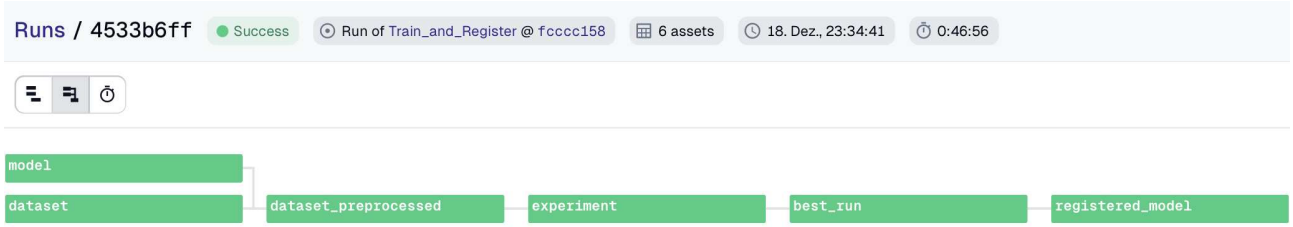



Abbildung C.26.: Darstellung der Assets eines Runs in der Dagster-UI

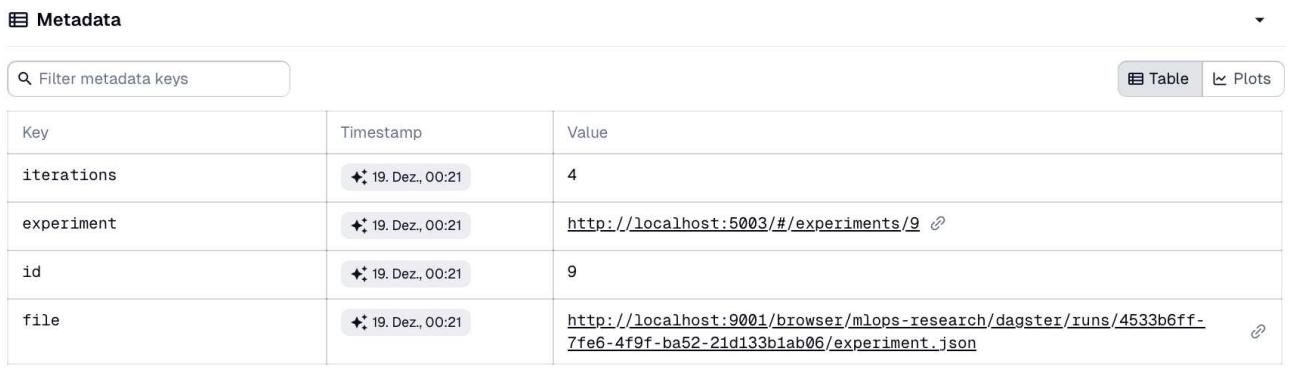



Abbildung C.27.: Metadaten des Experiment-Assets in der Dagster-UI

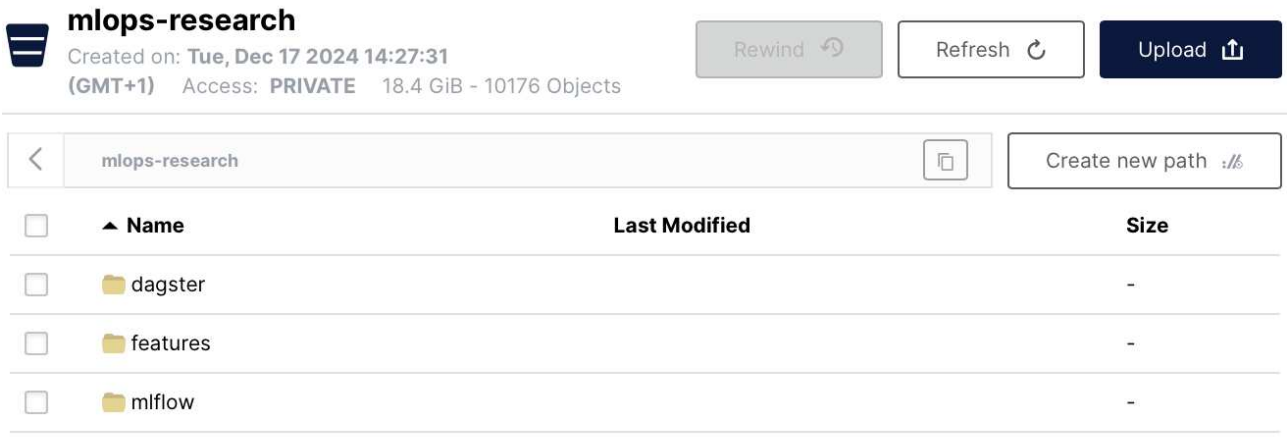



Abbildung C.28.: Getrennte Ordner für Services im MinIO-Bucket

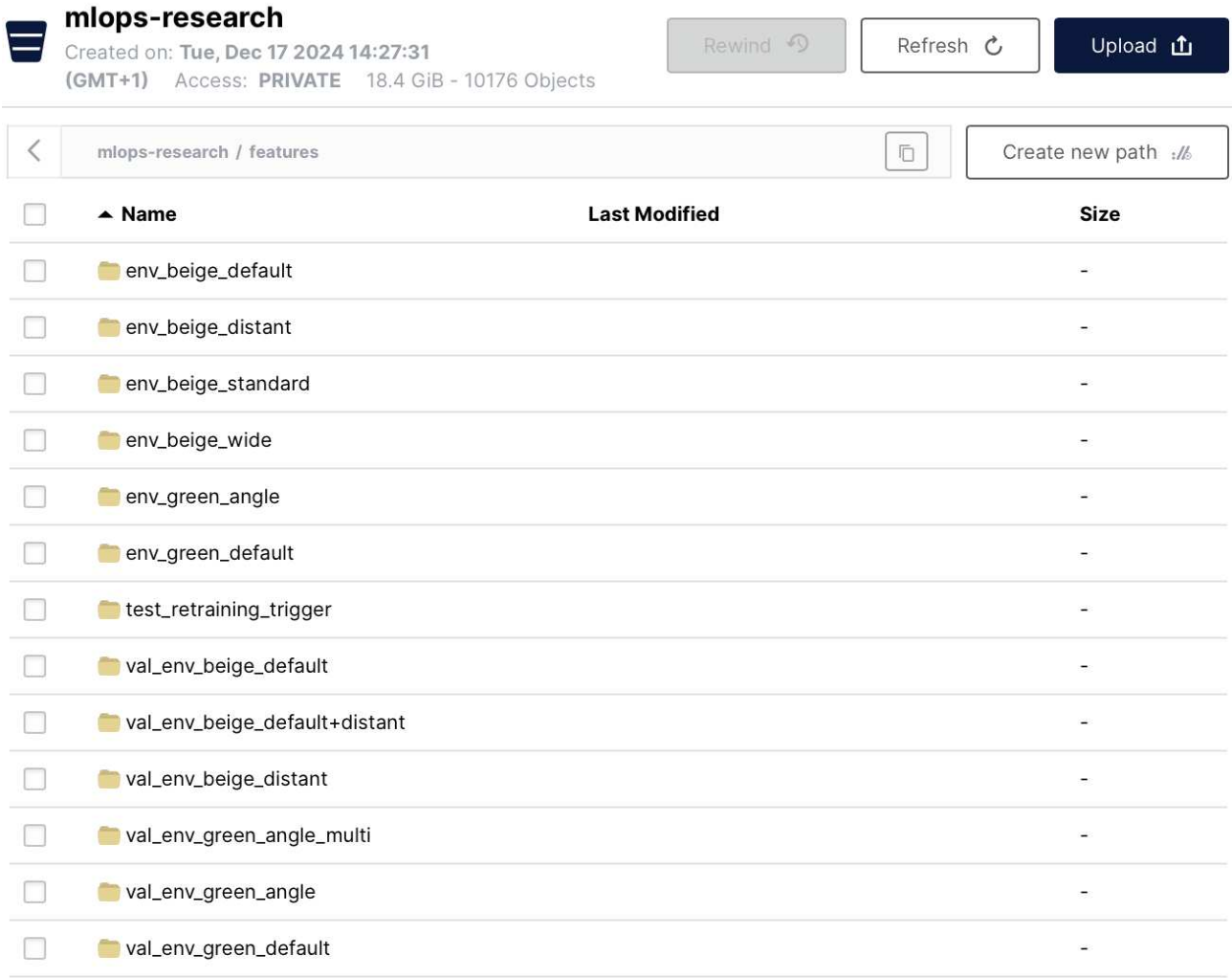



Abbildung C.29.: Ablage der gesammelten Bilddaten im MinIO-Bucket

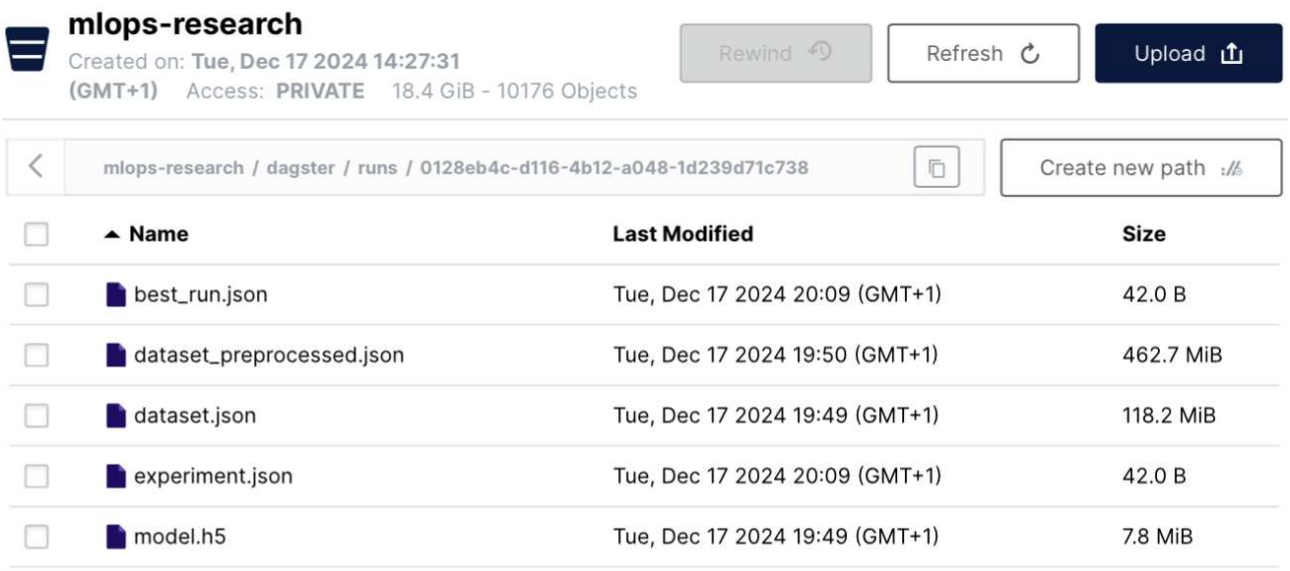



Abbildung C.30.: Ablage der Dagster-Run Metadaten im MinIO-Bucket


Abbildung C.31.: Ablage der MLFlow-Modellartefakte im MinIO-Bucket

# D. Projektplanung

1. **Themenauswahl und Einarbeitung**
   - Einarbeitung in die Grundlagen von MLOps, Einordnung und Eingrenzung des Themas sowie Festlegen des Forschungsziels
   - Dauer: 1 Woche
2. **Literaturrecherche und Stand der Technik**
   - Recherche, Analyse und Bewertung aktueller Literatur zu MLOps und verwandten Themen; Identifikation der MLOps-Prinzipien und -Komponenten als Basis für die eigene Entwicklung; praxisorientierter Entwurf der Systemarchitektur anhand der theoretischen Erkenntnisse
   - Dauer: 3 Wochen
   - Abhängig von: Themenauswahl und Einarbeitung
3. **Implementierung der Softwarelösung (Proof of Concept)**
   - Implementierung der identifizierten Prinzipien und Komponenten von MLOps im Gesamtsystem als Grundlage für die Evaluation; systematisches Entwickeln, Testen und Dokumentieren einzelner Dienste und Schnittstellen
   - Dauer: 5 Wochen
   - Abhängig von: Literaturrecherche und Stand der Technik
4. **Entwicklung des praktischen Anwendungsbeispiels**
   - Konkretisierung des Anwendungsfalls, Entwicklung des Miniaturfahrzeugs mit Edge-Software und Planung der Versuchsreihen zur Evaluation
   - Dauer: 1 Woche
   - Abhängig von: Literaturrecherche und Stand der Technik
5. **Durchführung der praktischen Evaluation**
   - Aufbau der Umgebung, Durchführung der Versuchsreihen und Auswertung der Ergebnisse; paralleles korrigieren von aufgetretenen Fehlern und Optimieren der Softwarelösung
   - Abhängig von: Implementierung der Softwarelösung, Entwicklung des praktischen Anwendungsbeispiels
   - Dauer: 1 Woche

6. **Dokumentation der Ergebnisse**
    - Zusammenführen von Recherche, Implementierung und Evaluation in der schriftlichen Ausarbeitung und Anfertigen von Abbildungen zur Veranschaulichung der Ergebnisse
    - Abhängig von: Durchführung der praktischen Evaluation
    - Dauer: 4 Wochen
7. **Korrektur und Fertigstellung der Arbeit**
    - Hinzufügen von übrigen Kapiteln sowie Einleitung, Zusammenfassung und Abstract; Überprüfung der Arbeit auf formale und inhaltliche Korrektheit sowie Verständnis in Rücksprache mit Betreuer und Korrekturlesern.
    - Dauer: 1 Woche
    - Abhängig von: Dokumentation der Ergebnisse
8. **Finalisierung der Softwarelösung**
    - Erweitern der Softwarelösung um zusätzliche Funktionen; bearbeiten offener „Todos“ und umfangreiche Dokumentation für die erfolgreiche Anwendung des entwickelten Systems in weiteren Projekten
    - Dauer: 2 Wochen
    - Abhängig von: Korrektur und Fertigstellung der Arbeit
9. **Vorbereitung von Präsentation und Materialien**
    - Terminieren der Präsentation, Erstellen von Folien, Vorbereitung von Vortrag und Demos sowie Anfertigen eines Posters
    - Dauer: 1 Woche
    - Abhängig von: Korrektur und Fertigstellung der Arbeit

Abbildung D.1 zeigt den erstellten Zeitplan als Gantt-Diagramm mit farblicher Kennzeichnung der verschiedenen Tätigkeiten.

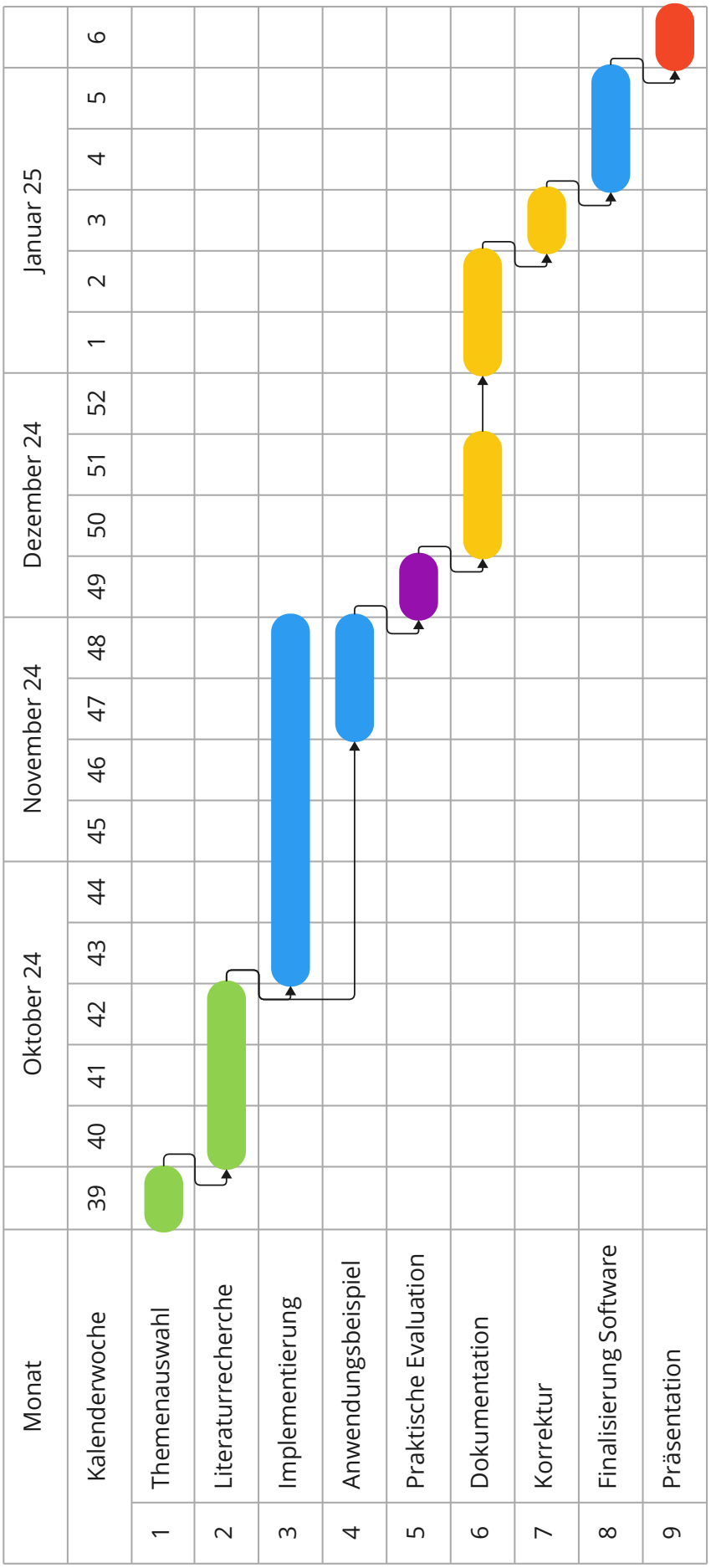



Abbildung D.1.: Projektplanung als Gantt-Diagramm

# Literaturverzeichnis


[1] McKinsey & Company. *The State of AI in early 2024: Gen AI adoption spikes and starts to generate value*. Mai 2024. URL: https://www.mckinsey.com/capabilities/quantumblack/our-insights/the-state-of-ai# (besucht am 06. 08. 2024).

[2] Rackspace Technology. *Global AI Report 2024*. 2024. URL: https://www.rackspace.com/sites/default/files/2024-03/2024-Global-AI-Report-Rackspace-FAIR.pdf (besucht am 06. 08. 2024).

[3] Gartner Inc. *Gartner Says Nearly Half of CIOs Are Planning to Deploy Artificial Intelligence*. Feb. 2018. URL: https://www.gartner.com/en/newsroom/press-releases/2018-02-13-gartner-says-nearly-half-of-cios-are-planning-to-deploy-artificial-intelligence (besucht am 06. 08. 2024).

[4] Iguazio Ltd. *Automating CI/CD for Machine Learning*. URL: https://go.iguazio.com/automating-ci/cd-for-machine-learning (besucht am 07. 08. 2024).

[5] D. Sculley u. a. *Hidden Technical Debt in Machine Learning Systems*. 2015.

[6] Dayne Sorvisto, Hrsg. *MLOps Lifecycle Toolkit: A Software Engineering Roadmap for Designing, Deploying, and Scaling Stochastic Systems*. 2023.

[7] appliedAI Institute for Europe, UnternehmerTUM und IPAI. *The MLOps Workbook: A Guided Online Course for Getting Started with MLOps*. 2023. URL: https://www.appliedai-institute.de/en/free-online-courses/the-mlops-workbook/ (besucht am 17. 11. 2024).

[8] Tesla Inc. *Replacing Ultrasonic Sensors with Tesla Vision*. URL: https://www.tesla.com/support/transitioning-tesla-vision (besucht am 13. 11. 2024).

[9] Rahul Y und Binoy B Nair. „Camera-Based Object Detection, Identification and Distance Estimation“. In: *2018 2nd International Conference on Micro-Electronics and Telecommunication Engineering (ICMETE)*. Sep. 2018. DOI: 10.1109/ICMETE.2018.00052. URL: https://ieeexplore.ieee.org/document/8713664 (besucht am 13. 11. 2024).

[10] Bundesministerium für Digitales und Verkehr. *Neue Fahrzeugsicherheitssysteme*. 28. Feb. 2024. URL: https://bmdv.bund.de/SharedDocs/DE/Artikel/StV/Strassenverkehr/neue-fahrzeugsicherheitssysteme.html (besucht am 13. 11. 2024).

[11] Nicolas Caballero. *Brief Analysis Of Tesla Withdrawing Ultrasonic Sensors From Its EVs*. 19. Okt. 2022. URL: https://www.torquenews.com/15475/brief-analysis-tesla-withdrawing-ultrasonic-sensors-its-evs (besucht am 13. 11. 2024).

[12] Red Hat Inc. *What is an IDE?* URL: https://www.redhat.com/en/topics/middleware/what-is-ide (besucht am 13. 11. 2024).

[13] Jupyter Team. *Project Jupyter Documentation*. URL: https://docs.jupyter.org/en/latest/ (besucht am 13. 11. 2024).

[14] Software Freedom Conservancy Inc. *Getting Started - About Version Control*. URL: https://git-scm.com/book/ms/v2/Getting-Started-About-Version-Control (besucht am 13. 11. 2024).

[15] Stack Overflow. *Stack Overflow Developer Survey 2022*. URL: https://survey.stackoverflow.co/2022/#section-version-control-version-control-systems (besucht am 13. 11. 2024).

[16] Microsoft Corporation. *Flake8 extension for VSCode*. URL: https://marketplace.visualstudio.com/items?itemName=ms-python.flake8 (besucht am 14. 11. 2024).

[17] Robert Cecil Martin. *TheBowlingGameKata*. Juni 2005. URL: http://butunclebob.com/ArticleS.UncleBob.TheBowlingGameKata (besucht am 28. 01. 2025).

[18] Ashwin Pajankar. *Python Unit Test Automation: Practical Techniques for Python Developers and Testers*. 2017.

[19] Python Packaging Community. *Python Packaging User Guide*. URL: https://packaging.python.org/en/latest/tutorials/installing-packages/ (besucht am 14. 11. 2024).

[20] Nihal Shetty. *Digital Engineering Transformation Through DevOps*. URL: https://www.bosch-softwaretechnologies.com/en/explore-and-experience/digital-engineering-transformation-through-devops/ (besucht am 14. 11. 2024).

[21] GitLab Inc. *What is CI/CD?* URL: https://about.gitlab.com/topics/ci-cd/ (besucht am 14. 11. 2024).

[22] Eyer AS. *Grafana for DevOps: A Practical Guide*. URL: https://eyer.ai/blog/grafana-for-devops-a-practical-guide/ (besucht am 14. 11. 2024).

[23] Ian Goodfellow, Yoshua Bengio und Aaron Courville. *Deep Learning*. https://deeplearningbook.org. MIT Press, 2016.

[24] Michael A. Nielsen. *Neural Networks and Deep Learning*. https://neuralnetworksanddeeplearning.com/. Determination Press, 2015.

[25] Bettina Belousov. *Warum Python perfekt für maschinelles Lernen ist: Eine tiefgreifende Analyse*. 17. Nov. 2023. URL: https://saracus.com/synvert-saracus-blog/warum-python-perfekt-fuer-maschinelles-lernen-ist-eine-tiefgreifende-analyse/ (besucht am 18. 11. 2024).

[26] Stack Overflow. *Stack Overflow Developer Survey 2024*. URL: https://survey.stackoverflow.co/2024/technology/#1-other-frameworks-and-libraries (besucht am 13. 11. 2024).

[27] Walker Rowe und Jonathan Johnson. *Top Machine Learning Frameworks To Use*. URL: https://www.bmc.com/blogs/machine-learning-ai-frameworks/ (besucht am 14. 11. 2024).

[28] Sridhar Alla und Suman Kalyan Adari. *Beginning MLOps with MLFlow*. Apress, 2021.

[29] Dominik Kreuzberger, Niklas Kühl und Sebastian Hirschl. *Machine Learning Operations (MLOps): Overview, Definition, and Architecture*. 2023. URL: https://ieeexplore.ieee.org/document/10081336.

[30] Mihail Eric. *MLOps Is a Mess But That's to be Expected*. März 2022. URL: https://www.mihaileric.com/posts/mlops-is-a-mess/ (besucht am 28. 11. 2024).

[31] Bitmotec GmbH. *InfluxDB – Einführung in die Open-Source-Zeitreihendatenbank*. URL: https://www.bitmotec.com/blog/influxdb-einfuehrung-in-die-open-source-zeitreihendatenbank/ (besucht am 21. 11. 2024).

[32] Charles Mahler. *Getting Started with InfluxDB and Grafana*. 20. Dez. 2022. URL: https://www.influxdata.com/blog/getting-started-influxdb-grafana/ (besucht am 21. 11. 2024).

[33] Dagster Inc. *The what and why of Dagster*. URL: https://docs.dagster.io/getting-started/what-why-dagster (besucht am 27. 11. 2024).

[34] Patrick Soschinski und Tom Scholz. *Lessons learned: Was wir in einem Jahr ML Orchestrierung mit Dagster gelernt haben*. 12. Sep. 2024. URL: https://www.codecentric.de/wissens-hub/blog/lessons-learned-was-wir-in-einem-jahr-ml-orchestrierung-mit-dagster-gelernt-haben (besucht am 27. 11. 2024).

[35] Flask Community. *Flask Quickstart*. URL: https://flask.palletsprojects.com/en/stable/quickstart/ (besucht am 27. 11. 2024).

[36] Mohan Reddy. *Ground Truth Gold - Intelligent data labeling and annotation*. März 2018. URL: https://medium.com/hivedata/ground-truth-gold-intelligent-data-labeling-and-annotation-632f63d9662f (besucht am 05. 01. 2025).

[37] Kieran Woodward u. a. *LabelSens: enabling real-time sensor data labelling at the point of collection using an artificial intelligence-based approach*. Juni 2020. URL: https://link.springer.com/article/10.1007/s00779-020-01427-x (besucht am 06. 01. 2025).

[38] Patrick Bangert u. a. *Active Learning Performance in Labeling Radiology Images Is 90% Effective*. Okt. 2021. URL: https://www.frontiersin.org/journals/radiology/articles/10.3389/fradi.2021.748968/full (besucht am 05. 01. 2025).

[39] Pooyan Jamshidi u. a. *Microservices: The Journey So Far and Challenges Ahead*. 2012. URL: https://ieeexplore.ieee.org/abstract/document/8354433.

[40] Docker Community. *What is Docker?* URL: https://docs.docker.com/get-started/docker-overview/ (besucht am 19. 11. 2024).

[41] Docker Community. *Docker Compose*. URL: https://docs.docker.com/compose/ (besucht am 19. 11. 2024).

[42] Kubernetes Community. *Kubernetes Overview*. URL: https://kubernetes.io/docs/concepts/overview/ (besucht am 19. 11. 2024).

[43] Google LCC. *Was ist Cloud-Computing?* URL: https://cloud.google.com/learn/what-is-cloud-computing (besucht am 02. 10. 2024).

[44] Subhendu Nayak. *SageMaker vs Azure ML vs Google AI Platform: A Comprehensive Comparison*. 13. Sep. 2024. URL: https://www.cloudoptimo.com/blog/sagemaker-vs-azure-ml-vs-google-ai-platform-a-comprehensive-comparison/ (besucht am 20. 11. 2024).

[45] Dr. Harald Dreher. *Vor- und Nachteile des ERP-Cloud Computing*. 15. Feb. 2023. URL: https://www.dreher-consulting.com/de/blog/cloud-computing/ (besucht am 21. 11. 2024).

[46] Roman Burdiuzha. *The Power of AIOps: Transforming IT Operations*. Nov. 2023. URL: https://gartsolutions.medium.com/the-power-of-aiops-transforming-it-operations-0ef2bf8a4040 (besucht am 22. 01. 2025).

[47] IBM. *What is AIOps?* URL: https://www.ibm.com/think/topics/aiops (besucht am 22. 01. 2025).

[48] LinkedIn Community u. a. *How can you deploy machine learning models on edge devices?* URL: https://www.linkedin.com/advice/0/how-can-you-deploy-machine-learning-models-dbhxc (besucht am 23. 01. 2025).

Während der Erstellung dieser Arbeit wurde ChatGPT (OpenAI) unterstützend verwendet. Alle inhaltlichen Aussagen und Ergebnisse wurden eigenständig überprüft und verantwortet.

# Abbildungsverzeichnis

Die Abbildungen sind mit der Software Miro erstellt, aus den grafischen Oberflächen des entwickelten Systems exportiert oder selbst fotografiert. Dargestellte Logos vom Diensten und Tools sind geistiges Eigentum der zugehörigen Unternehmen. Enthalten die Abbildungen sonstige Grafiken von Dritten, sind diese entsprechend gekennzeichnet.