Technische Hochschule Deggendorf
Faculty Applied Information Technology
Studiengang Bachelor of Cyber-Security

Projekt "WeedDetector"
Projektübergabe

Vorgelegt von:

Christof Renner (22301943)
Manuel Friedl (1236626)

Prüfungsleitung:

Prof. Dr. Holger Jehle

Datum: 05. Juli 2025

Contents

1 Einführung und Projektübersicht

1.1 Motivation und Zielsetzung

Der **WeedDetector** ist ein innovatives Softwaresystem, das entwickelt wurde, um in landwirtschaftlichen Umgebungen automatisiert Unkräuter zu erkennen und zu klassifizieren. Die Hauptmotivation hinter diesem Projekt liegt in der zunehmenden Notwendigkeit nachhaltiger landwirtschaftlicher Praktiken und der Reduzierung des Einsatzes von Herbiziden.

Dieses Projekt entstand aus folgenden konkreten Anforderungen:

- **Präzisionslandwirtschaft:** Durch die genaue Identifikation von Unkraut können Landwirte gezielt eingreifen, anstatt flächendeckend Herbizide einzusetzen.
- **Umweltschutz:** Reduzierung des Chemikalieneinsatzes durch punktgenaue Behandlung von Unkrautflächen.
- **Automatisierung:** Integration mit Robotersystemen zur vollautomatischen Unkrautbekämpfung.
- **Kosteneinsparung:** Verringerung des Ressourcenverbrauchs und effizientere Arbeitsabläufe.

1.2 Systemarchitektur

Der WeedDetector folgt dem Model-View-Controller (MVC) Architekturmuster, um eine saubere Trennung der Verantwortlichkeiten zu gewährleisten und die Wartbarkeit zu verbessern.

1.3 Technologiestack

Der WeedDetector nutzt einen modernen Technologiestack, der für Bildverarbeitung und maschinelles Lernen optimiert ist:

- **Programmiersprache:** Python 3.13 als Hauptsprache für die gesamte Anwendung
- **Computer Vision:**
  - OpenCV 4.11.0.86 für Bildverarbeitung und Kameraintegration
  - Ultralytics YOLOv8 8.3.155 für Objekterkennung und -klassifikation
- **Benutzeroberfläche:**
  - Tkinter für die grafische Benutzeroberfläche
  - Pillow 11.2.1 für erweiterte Bildverarbeitung in der GUI

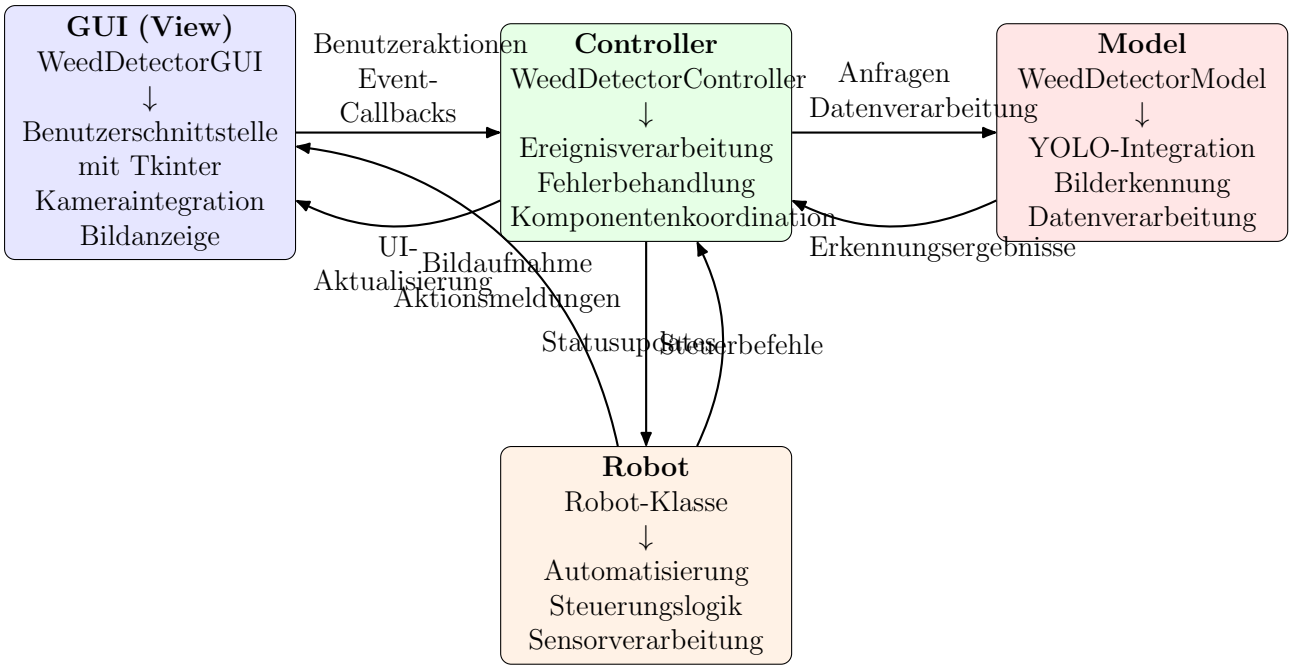

Figure 1.1: Detaillierte Architektur des WeedDetector-Systems

- **Containerisierung:** Docker für einheitliche Entwicklungs- und Produktionsumgebungen
- **Kontinuierliche Integration:** GitHub Actions für automatisierte Tests und Builds

2 Installation und Konfiguration

2.1 Systemvoraussetzungen

Für den erfolgreichen Betrieb des WeedDetector-Systems werden folgende Mindestsystemvoraussetzungen empfohlen:

- **Hardware:**
  - CPU: Quad-Core Prozessor, 2.5 GHz oder höher
  - RAM: Mindestens 8 GB (16 GB empfohlen)
  - Grafikkarte: CUDA-fähige GPU mit mindestens 4 GB VRAM für optimale Leistung
  - Festplattenspeicher: Mindestens 2 GB freier Speicherplatz
  - Kamera: USB-Webcam oder integrierte Kamera für Live-Erkennung
- **Software:**
  - Betriebssystem: Linux (Ubuntu 22.04 LTS empfohlen), Windows 10/11 oder macOS
  - Docker: Version 24.0 oder höher
  - X11-Server (für GUI-Anzeige aus dem Container)

2.2 Installationsanleitung

Die Installation des WeedDetector-Systems erfolgt primär über Docker, um eine konsistente Umgebung zu gewährleisten und Abhängigkeitsprobleme zu vermeiden.

2.2.1 Installation mittels Docker

1. **Docker installieren:** Falls noch nicht geschehen, installieren Sie Docker gemäß der offiziellen Anleitung für Ihr Betriebssystem (`https://docs.docker.com/get-docker/`).
2. **Repository klonen:**

```
1    git clone https://github.com/SS25-SoftwareEngineering-
        WeedDetector.git
2    cd SS25-SoftwareEngineering-WeedDetector
```

3. **Docker-Image erstellen:**

```
1    docker build -t weed-detector .
```

4. **Anwendung starten:**

```
docker run --device /dev/video0:/dev/video0 --net=host -e
    DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix weed-
    detector
```

2.2.2 Manuelle Installation (ohne Docker)

Für Entwicklungs- oder Testzwecke kann das System auch manuell installiert werden:

1. **Python-Umgebung einrichten:**

```
python -m venv venv
source venv/bin/activate  # Unter Windows: venv\Scripts\
    activate
```

2. **Abhängigkeiten installieren:**

```
pip install -r requirements.txt
```

3. **Anwendung starten:**

```
python app/main.py
```

2.3 Konfiguration und Anpassung

Der WeedDetector bietet verschiedene Konfigurationsmöglichkeiten, um das System an spezifische Anforderungen anzupassen:

2.3.1 Modellkonfiguration

Die Standardkonfiguration verwendet ein vortrainiertes YOLOv8-Modell. Das System sucht automatisch nach trainierten Modellen in folgenden Pfaden:

- `runs/detect_train/weights/best.pt`
- `data/weights/best.pt`
- `data/models/best.pt`
- `models/best.pt`

Wenn kein benutzerdefiniertes Modell gefunden wird, wird auf das Standard-YOLOv8-Modell zurückgegriffen.

2.3.2 Umgebungsvariablen

Folgende Umgebungsvariablen können beim Start des Docker-Containers konfiguriert werden:

- `QT_X11_NO_MITSHM=1`: Verhindert Probleme mit X11-Shared-Memory
- `QT_DEBUG_PLUGINS=1`: Aktiviert Debug-Ausgaben für Qt-Plugins
- `OPENCV_VIDEOIO_PRIORITY_V4L2=0`: Priorität für Video4Linux2-Backend
- `OPENCV_VIDEOIO_DEBUG=1`: Aktiviert Debug-Informationen für OpenCV Video-I/O

3 Milestones & User Stories

In diesem Abschnitt werden die wichtigsten **Meilensteine** und **User Stories** des Projekts Weed Detector dokumentiert. Sie dienen dazu, den Projektfortschritt nachvollziehbar zu machen und zeigen auf, wie die Anforderungen schrittweise umgesetzt wurden.

Die User Stories orientieren sich an den Bedürfnissen der Nutzer und definieren die Funktionen aus Anwendersicht, während die Milestones die wesentlichen Etappen im Entwicklungsprozess markieren, um die Struktur und Planung des Projekts übersichtlich darzustellen.

3.1 Milestones

Folgende Meilensteine wurden am Anfang des Projektes festgelegt:

1. **Anforderungsanalyse & Architektur (05.05. - 12.05.25):** Festlegen des technischen Konzepts und der Architektur
2. **Prototyp Klassifikation & Koordinatenbestimmung (12.05. - 25.05.25):** Passenden Datensatz (Bilder, WeedAI) suchen oder erstellen, Bilder in ein einheitliches Format bringen, labeln und optional durch Data Augmentation erweitern.
3. **Prototyp Navigation & Integration (25.05. - 08.06.25):** Implementierung der Bildvorverarbeitung, Klassifikation und Koordinatenbestimmung sowie erste Integration in ROS.
4. **Test, Validierung & Performance-Optimierung (08.06. - 22.06.25):** Erstellung und Durchführung von Unit-, Integrations- und Performance-Tests
5. **Dokumentation & Präsentation (22.06. - 07.07-25):** Dokumentation (Ziele, Methodik, Ergebnisse) erstellen, Code kommentieren, Screenshots einfügen, Präsentation vorbereiten und am 07.07.25 präsentieren.

3.2 User Stories

Nachfolgend werden die User Stories aufgeführt, die zu den jeweiligen Milestones definiert wurden, um die Anforderungen aus Nutzersicht klar darzustellen und den Bezug zum Entwicklungsfortschritt sichtbar zu machen.

- **M1:**
  - Als *Entwickler* möchte ich Anforderungen und Ziele definieren, um eine klare Projektvision zu haben.

  - Als *Entwickler* möchte ich die passenden Tools und Bibliotheken auswählen, um den Entwicklungsprozess effizient zu gestalten.

- **M2:**
  - Als *Landwirt/Gärtner* möchte ich, dass der Weeds Detector typische Unkräuter erkennt, um mein Feld effizient zu analysieren.
  - Als *Entwickler* möchte ich ein ML-Modell trainieren, das Unkraut in Bildern erkennt, um den Erkennungsprozess zu automatisieren.
  - Als *Entwickler* möchte ich die Koordinaten der Unkrautpositionen berechnen, um die Lokalisierung zu ermöglichen.
  - ALs *User* möchte ich, dass das System visuell anzeigt, wo Unkraut erkannt wurde, damit ich gezielt eingreifen kann.

- **M3:**
  - Als *Landwirt/Gärtner* möchte ich, dass der Roboter zielgerichtet zum Unkraut fährt, um manuelles Eingreifen zu reduzieren.
  - Als *Entwickler* möchte ich erkannte Koordinaten an das Navigationssystem übergeben, damit gezielt darauf reagiert werden kann.
  - Als *Entwickler* möchte ich Navigation und Klassifikation integrieren, um ein durchgängiges System zu schaffen.

- **M4:**
  - Als *Nutzer* möchte ich wissen, wie zuverlässig die Vorhersagen sind, um Vertrauen ins System zu haben.
  - Als *Nutzer* möchte ich die Anwendung einfach und selbsterklärend bedienen können.
  - Als *Entwickler* möchte ich das Modell mit neuen Bildern testen, um die Genauigkeit zu validieren.

- **M5:**
  - Als *Stakeholder* möchte ich verstehen, wie das System funktioniert, um seine Relevanz zu beurteilen.
  - Als *Entwickler* möchte ich meinen Code und Ergebnisse dokumentieren, um das Projekt verständlich und nachvollziehbar zu machen.
  - Als *Entwickler* möchte ich eine Präsentation vorbereiten, um mein Projekt überzeugend vorstellen zu können

4 Scope Changes & Blocking Issues

4.1 Scope Changes

In diesem Abschnitt werden die **Änderungen am ursprünglichen Projektumfang (Scope Changes)** dokumentiert, die im Verlauf der Entwicklung notwendig wurden. Diese Anpassungen entstanden durch neue Anforderungen, technische Herausforderungen oder Optimierungsmöglichkeiten und zeigen, wie das Projekt flexibel an veränderte Bedingungen angepasst wurde.

- **Meilenstein 2:** Festlegung, dass der Docker-Container nur unter Linux läuft, um Kompatibilitätsprobleme mit der Tkinter-Library unter Windows zu vermeiden.
- **Meilenstein 3:** Ergänzung eines UI end-to-end Redesign & Refactors, um eine seamless UI/UX und zeitgemäße Ästhetik gemäß Best Practices sicherzustellen.

4.2 Blocking Issues

Im Verlauf des Projekts traten keine direkten **Blocking Issues** auf, die den Fortschritt maßgeblich verzögert oder die Entwicklung gestoppt hätten. Alle auftretenden technischen oder organisatorischen Herausforderungen konnten zeitnah identifiziert und gelöst werden, wodurch ein kontinuierlicher Arbeitsfluss im Projekt gewährleistet war.

5 Modelltraining und -anpassung

5.1 Vorbereitung des Trainingsdatensatzes

Die Qualität des Erkennungsmodells hängt maßgeblich von der Qualität und Vielfalt des Trainingsdatensatzes ab.

5.1.1 Datenannotation

Die Annotation der Trainingsdaten kann mit folgenden Tools durchgeführt werden:

- **Roboflow:** Online-Plattform mit intuitiver Benutzeroberfläche
- **LabelImg:** Lokales Open-Source-Tool für YOLO-Annotationen
- **CVAT:** Leistungsstarkes webbasiertes Annotationstool

5.1.2 Datenaugmentierung

Zur Verbesserung der Modellrobustheit werden folgende Augmentierungstechniken empfohlen:

- Horizontale und vertikale Spiegelung
- Rotation ($\pm 15^\circ$ bis $\pm 90^\circ$)
- Zufällige Zuschnitte (0-20%)
- Helligkeits- und Kontrastvariationen ($\pm$10-20%)
- Sättigungsanpassungen ($\pm$20%)

5.2 Durchführung des Trainings

Der Trainingsprozess kann sowohl lokal als auch in der Cloud durchgeführt werden.

5.2.1 Training starten

Starten Sie das Training mit dem folgenden Befehl:

```
python train.py --data data/data.yaml --epochs 50 --img 640 --batch 16
```

Trainingsparameter können je nach verfügbarer Hardware angepasst werden:

- `--epochs`: Anzahl der Trainingsdurchläufe (empfohlen: 50-300)
- `--img`: Bildgröße für das Training (empfohlen: 640)
- `--batch`: Batch-Größe (an verfügbaren GPU-Speicher anpassen)

5.3 Modellbewertung und -optimierung

Nach Abschluss des Trainings sollte das Modell bewertet und optimiert werden.

5.3.1 Bewertungsmetriken

Die wichtigsten Metriken zur Bewertung des Modells sind:

- **mAP (mean Average Precision):** Gesamtgenauigkeit über alle Klassen
- **Precision:** Verhältnis korrekter Erkennungen zu allen Erkennungen
- **Recall:** Verhältnis korrekter Erkennungen zu allen tatsächlichen Objekten
- **F1-Score:** Harmonisches Mittel aus Precision und Recall

5.3.2 Feinabstimmung des Modells

Bei unzureichender Leistung können folgende Optimierungsschritte durchgeführt werden:

1. **Daten verbessern:** Mehr Trainingsbilder, bessere Annotation, zusätzliche Augmentierung
2. **Hyperparameter anpassen:** Lernrate, Epochenzahl, Batch-Größe
3. **Modellarchitektur ändern:** Größeres/kleineres Basismodell (YOLOv8n, YOLOv8s, YOLOv8m, YOLOv8l, YOLOv8x)
4. **Transfer Learning:** Training auf vortrainiertem Modell für verwandte Domänen

5.4 Fehlerbehandlung und Logging

Das System implementiert eine umfassende Fehlerbehandlung, um Robustheit zu gewährleisten:

- **Exception Handling:** Alle kritischen Operationen sind mit spezifischen Exception-Handlern umgeben.
- **Fehlerprotokolle:** Fehler werden sowohl in der Konsole als auch in der GUI angezeigt.
- **Fallback-Mechanismen:** Bei Problemen mit benutzerdefinierten Modellen erfolgt ein Fallback auf das Standardmodell.
- **Robuste Kameraintegration:** Sichere Behandlung von Kamerazugriffsfehlern und Bildverarbeitungsproblemen.

6 Test, Validierung & Perfromance-Optimierung

6.1 Unit-, Integration- und Systemtests

6.1.1 Unit-Tests

Für die Weed Detector Anwendung wurden umfangreiche Unittests entwickelt, um die Funktionalität und Stabilität der wichtigsten Komponenten sicherzustellen. Die Tests decken insbesondere die grafische Benutzeroberfläche (GUI), das Modell zur Unkrauterkennung sowie die Controller-Logik ab. Dabei werden sowohl typische Anwendungsfälle als auch Fehler- und Randfälle überprüft. Die GUI-Tests simulieren Benutzerinteraktionen wie das Auswählen und Anzeigen von Bildern, das Starten und Stoppen der Kamera sowie das Anzeigen von Fehlermeldungen. Für das Modell werden unter anderem das Laden und Verarbeiten von Bildern, die Fehlerbehandlung bei ungültigen Dateien und die Formatierung der Erkennungsergebnisse getestet. Die Controller-Tests prüfen das Zusammenspiel zwischen GUI und Modell, einschließlich der Fehlerbehandlung und der Steuerung des Roboters. Durch den Einsatz von Mocking werden externe Abhängigkeiten isoliert, sodass die Tests reproduzierbar und unabhängig von der Hardware ausgeführt werden können. Die Unittests tragen maßgeblich zur Qualitätssicherung und Wartbarkeit des Projekts bei.

Listing 6.1: Beispiel Unit-Test

```
1     @patch('tkinter.messagebox.showerror')
2     def test_show_error_box(self, mock_showerror):
3         """Test if error box is shown correctly."""
4         self.gui.show_error_box("Fehlertext")
5         mock_showerror.assert_called_with("Error", "Fehlertext")
6         content = self.gui.results_text.get("1.0", tk.END)
7         self.assertIn("Error: Fehlertext", content)
```

6.1.2 Integrations-Tests

Zur Sicherstellung des fehlerfreien Zusammenspiels aller Hauptkomponenten wurden für die Weed Detector Anwendung umfassende Integrationstests implementiert. Diese Tests überprüfen insbesondere die Interaktion zwischen GUI, Modell und Controller im Gesamtsystem. Es werden typische Abläufe wie das Laden und Verarbeiten von Bildern, die Anzeige von Ergebnissen, das Starten und Stoppen des Roboters sowie die Fehlerbehandlung bei fehlerhaften Eingaben getestet. Dabei wird sowohl das Verhalten bei korrekten als auch bei fehlerhaften Abläufen geprüft, um die Robustheit der Anwendung zu gewährleisten. Durch den Einsatz von Mocks für GUI-Elemente und Modellfunktionen können die Integrationstests unabhängig von externer Hardware und realen Bilddaten

ausgeführt werden. Die Integrationstests tragen somit entscheidend dazu bei, die Zuverlässigkeit und Stabilität der gesamten Anwendung sicherzustellen.

Listing 6.2: Beispiel Integrations-Test

```
    def test_model_detect_weeds_with_real_image(self):
        """Test: mnodel detects weeds and returns results."""
        image_path = os.path.join(os.path.dirname(__file__), "..", "
            unkraut1.jpg")
        image_path = os.path.abspath(image_path)
        if not os.path.exists(image_path):
            self.skipTest("unkraut1.jpg␣not␣found")
        image = cv2.imread(image_path)
        processed, result = self.model.detect_weeds(image)
        self.assertIsNotNone(processed)
        self.assertIsInstance(result, str)
```

6.1.3 Systemtest

Für die Weed Detector Anwendung wurde ein Systemtest implementiert, der den vollständigen Ablauf der Unkrauterkennung unter realistischen Bedingungen simuliert. Dabei werden alle Hauptkomponenten – Modell, GUI und Controller – gemeinsam in einer Testumgebung ausgeführt. Der Test bildet einen typischen Nutzer-Workflow ab: Ein reales Bild wird geladen, durch das Modell verarbeitet und das Ergebnis anschließend in der grafischen Oberfläche angezeigt. Abschließend wird überprüft, ob die Erkennungsergebnisse korrekt im GUI dargestellt werden. Dieser End-to-End-Test stellt sicher, dass die Integration aller Komponenten im Gesamtsystem funktioniert und die Anwendung aus Sicht eines Anwenders zuverlässig arbeitet. Der Systemtest trägt somit wesentlich zur Qualitätssicherung und zur Validierung der zentralen Funktionalität der Anwendung bei.

Listing 6.3: Code-Snippet Systemtest

```
    ...
    def test_full_detection_flow(self):
        """Simulate user uploading an image and verify detection and GUI
            update."""
        # Simulate user selects image
        image = self.model.load_image(self.image_path)
        processed, result = self.model.detect_weeds(image)
        # Display in GUI
        self.gui.display_image(processed)
        self.gui.update_results(result)
        # Check that results are shown in the GUI
        gui_content = self.gui.results_text.get("1.0", "end")
        self.assertIn("weed", gui_content.lower())
        self.assertGreater(len(gui_content.strip()), 0)
    ...
```

6.2 Line Coverage

Zur Sicherstellung der Codequalität wurde die Line-Coverage des Projekts gemessen, um die Abdeckung der implementierten Unit- und Integrationstests zu überprüfen. Der Weed

Detector erreichte dabei eine **Total Coverage Score von 85%**, was eine hohe Testabdeckung widerspiegelt und zeigt, dass der Großteil des Codes durch Tests abgesichert ist.

Listing 6.4: Line Coverage

```
PS C:\Repos\SS25-SoftwareEngineering-WeedDetector> python -m coverage report
 Name                 Stmts   Miss  Cover
 ---------------------------------------
 app\controller.py       55      6    89%
 app\gui.py             231     32    86%
 app\main.py             10      1    90%
 app\model.py           112     19    83%
 app\robot.py            45      9    80%
 ---------------------------------------
 TOTAL                  453     67    85%
```

6.3 Linting

Zur Sicherstellung einer sauberen und einheitlichen Codebasis wurde im Projekt Pylint für Linting eingesetzt. Der gesamte Code des Weed Detectors wurde damit auf Syntaxfehler, Stilkonformität nach PEP8 und potenzielle Probleme geprüft.

Durch regelmäßiges Linting konnte die Codequalität verbessert, die Lesbarkeit erhöht und die Wartbarkeit des Projekts sichergestellt werden.

Listing 6.5: Pylint Score

```
 ----------------------------------
 Your code has been rated at 8.97/10
```

7 Wartung und Weiterentwicklung

7.1 Bekannte Einschränkungen

Das aktuelle System hat folgende bekannte Einschränkungen:

- **Rechenleistung:** Die Echtzeiterkennung benötigt ausreichende CPU/GPU-Ressourcen.
- **Lichtempfindlichkeit:** Die Erkennungsgenauigkeit variiert je nach Lichtverhältnissen.
- **Modellspezifität:** Das Modell funktioniert am besten für die Unkrautarten, auf die es trainiert wurde.
- **GUI-Beschränkungen:** Die Tkinter-Oberfläche hat eingeschränkte Anpassungsmöglichkeiten.

7.2 Erweiterungsmöglichkeiten

Folgende Erweiterungen sind für zukünftige Versionen geplant:

1. **Erweiterte Roboterintegration:** Unterstützung für komplexere Roboterhardware und -steuerung.
2. **Cloud-Integration:** Anbindung an Cloud-Dienste für Modelltraining und Datenspeicherung.
3. **Mehrsprachige Unterstützung:** Internationalisierung der Benutzeroberfläche.
4. **Mobile App:** Entwicklung einer begleitenden mobilen Anwendung für Felddiagnosen.
5. **Kartenintegration:** Geo-Tagging und Kartierung erkannter Unkrautbereiche.
6. **Verbesserte Bildvorverarbeitung:** Automatische Bildkorrektur für verschiedene Lichtverhältnisse.

7.3 Wartungsrichtlinien

Zur langfristigen Wartung der Anwendung sollten folgende Richtlinien beachtet werden:

- **Regelmäßige Updates:** Aktualisierung der Abhängigkeiten, insbesondere von OpenCV und YOLOv8.
- **Modellneuevaluierung:** Periodische Neubewertung und ggf. Neutraining des Erkennungsmodells.

- **Codeüberprüfung:** Regelmäßige statische Codeanalyse mit Bandit, Snyk und Pylint.
- **Dokumentationsaktualisierung:** Aktualisierung der Dokumentation bei Änderungen am Code oder an Funktionen.
- **Testabdeckung:** Aufrechterhaltung und Erweiterung der automatisierten Testabdeckung.

8 Fazit und Ausblick

8.1 Projektergebnisse

Der WeedDetector repräsentiert eine erfolgreiche Integration moderner Computer-Vision-Technologien in landwirtschaftliche Anwendungen. Das System bietet:

- Eine benutzerfreundliche Oberfläche für die Unkrauterkennung
- Ein anpassbares und erweiterbares YOLO-basiertes Erkennungsmodell
- Eine modulare, wartbare Architektur nach dem MVC-Muster
- Eine vollständige Containerisierung für einfache Bereitstellung

8.2 Zukünftige Forschungsrichtungen

Basierend auf den Erfahrungen aus diesem Projekt ergeben sich folgende zukünftige Forschungs- und Entwicklungsrichtungen:

1. **Multi-Spektralanalyse:** Integration von Infrarot- und Multispektraldaten für verbesserte Erkennung.
2. **Drohnenintegration:** Anpassung des Systems für den Einsatz auf landwirtschaftlichen Drohnen.
3. **Federated Learning:** Verteiltes Modelltraining zwischen verschiedenen landwirtschaftlichen Betrieben.
4. **Präzisionssprühmechanismen:** Entwicklung präziser Mechanismen zur gezielten Behandlung einzelner Unkrautpflanzen.
5. **Langzeitüberwachung:** Systeme zur kontinuierlichen Überwachung und Analyse der Unkrautentwicklung über Wachstumsperioden hinweg.

8.3 Abschließende Bewertung

Der WeedDetector stellt einen bedeutenden Schritt in Richtung nachhaltiger, präziser Landwirtschaft dar. Die entwickelte Lösung bietet nicht nur praktische Vorteile für Landwirte, sondern trägt auch zu umweltfreundlicheren landwirtschaftlichen Praktiken bei.

Die modulare, erweiterbare Architektur des Systems ermöglicht zukünftige Anpassungen an spezifische landwirtschaftliche Anforderungen und die Integration neuer Technologien, sobald diese verfügbar werden.