Technical & Functional Design

Automation of Solarpanel Detection

Delvin Bacho & Navid Gharapanjeh

Saxion University of Applied Sciences
MSc Software Engineering

Supervisors:
Pieter Zeilstra & Deepak Tunuguntla

Abstract

The Solarpanel Detection System addresses the challenge of identifying rooftop solar installations across the Dutch housing stock. Our solution automates the retrieval of aerial imagery, the continuous training of object-detection models, and batch inference workflows. Key contributions include:

- Automated batch scraping of aerial images for Dutch houses.
- A continuous training pipeline for data ingestion, data preprocessing, model training, validation, and deployment.
- Batch detection of solarpanels with results formatted for integration into the NOWATT project dataset.

April 20, 2025

Contents

1 Introduction

This document outlines the technical and functional design of the project. The technical design will begin in Chapter 3: Solution Strategy.

1.1 Background and Context View

The NOWATT project aims to accelerate the energy transition and mitigate grid congestion by promoting energy-neutral neighborhoods. This initiative integrates multiple stakeholders, including residents, SMEs, and government agencies, to optimize energy efficiency through AI-driven solutions.
A key aspect of this project is leveraging Machine Learning to categorize homes and residents, enabling personalized sustainability recommendations and district-level planning. One of the major factors influencing a home's energy efficiency rating is the presence of solarpanels, as they significantly impact a building's energy label score.
By 2030, all rental properties in the Netherlands must have an energy label of at least D. Housing associations and landlords will need to take sustainability measures to comply with these regulations. Nijhuis Bouw, an organization specializing in sustainable housing solutions, is particularly interested in automating the assessment of energy efficiency to streamline this transition.

1.2 Problem Statement

The NOWATT project [1] aims to efficiently evaluate and label the energy performance of residential buildings based on specific house attributes. One key attribute crucial to this evaluation is whether a house is equipped with solarpanels. Currently, the identification of solarpanels on houses in the Netherlands is performed manually, a process that is both resource-intensive and time-consuming. To address this challenge, the objective of this project is to support NOWATT by developing an automated detection system capable of determining the presence of solarpanels on houses within the Netherlands. This automation will significantly reduce manual effort, enhance accuracy, and allow rapid scalability. In that NOWATT project, there are at the moment thousands of records, where each record shows a house, and the columns stands for housing attributes. Our goal is to be able to fill in the column, if a house has solarpanels or not.
To achieve this objective, the automated system will take as input either a Dutch city code or a list of unique house IDs and subsequently detect solarpanels on these houses by analyzing relevant imagery data. The specific goals of the project include:

1. **Continuous Training Pipeline**: Establishing a robust machine learning lifecycle that enables the model to be regularly updated with new data, ensuring ongoing accuracy and adaptability over time.

2. **Dutch Houses Collection Process**: Automating the process of retrieving house images from publicly available sources in the Netherlands. Each image must clearly depict exactly one individual house, and each house must have a unique House ID assigned for consistent tracking and identification.

3. **Inference/Detection Process**: Employing the trained model to automatically identify solarpanels on newly acquired imagery, explicitly linking each detection result to the corresponding unique House ID for traceability and subsequent evaluation purposes.

2 Functional Design

This section presents the functional design of the project, including the technical and operational constraints, the key stakeholders, and the overarching project vision. It also outlines the primary business goals and the corresponding functional requirements that the system must fulfill.

2.1 Constraints

Technical Constraints

- Software-Tools should be free or open source. (Client & Developer)
- Our own machine is not ideal, since we train on images (limited computing power without a compute server). For properly training a model, we would need a proper server for computing. (Developer)

Operational Constraints

- Deadline: April 20, 2025.
- Training data is provided by the project and is publicly available. Inference data (Dutch houses) is not available and must be collected, resulting in different formats and regions (Germany vs. Netherlands).
- This project only detects solarpanels on roofs, and does not detect vertical solarpanels on the façades of houses! At the moment of 2025, roof solarpanels are much popular, but in the future the trend might shift.

2.2 Stakeholders

Since our project is a subset of the NOWATT Project, and our results are expected only to be used for the NOWATT Project, the actual client for this project is Selin Colakhasanoglu (NOWATT representatitve). From now on, we will use Selins name or the NOWATT Project, when we refere to the client or end user of our project.

- **Developer Team**: Delvin Bacho & Navid Gharapanjeh
- **Project Lead & NOWATT Representative**: Selin Colakhasanoglu
- **NOWATTs Client Organization**: Nijhuis Bouw (Roel Prinsen)
- **Funding Agency**: Taskforce for Applied Research SIA

2.3 Vision

FOR the NOWATT Project
WHO need an automated and scalable way to assess and improve building energy efficiency
THE Solar Panel Detection System
THAT automatically detects solarpanels from aerial and satellite imagery to support energy label predictions
UNLIKE manual surveys and outdated labeling methods,
OUR PRODUCT offers a fully automated, ML-driven training pipeline.

2.4 Handover Documents

Since project inception, we have collected:

- **Research Paper & Satellite Imagery Dataset**: VHR imagery dataset from Germany with labeled residential solarpanels.
- **Python Script to Split Data**: `split_trainTest.py` divides satellite images into training and test sets.
- **Household Data for Energy Labels**: CSVs from Enschede:
  - `merged_3dbag_EPonline_Enschede.csv`: This file contains detailed building information. Eventually, an additional column indicating the presence of solar panels (as a boolean or binary value) should be added. However, we are not responsible for integrating this data into the dataset. Our task is to deliver the detection results linked to the corresponding house IDs, so they can be easily mapped to this file.

2.5 Business Goals and Functional Requirements

The NOWATT project leverages computer vision for solarpanel detection, links results to BAG IDs, and implements a continuous MLOps Training pipeline. Below are our primary goals and their associated requirements.

Goal 1: Continuous Training Pipeline

In order to keep model performance high as new imagery and installation patterns emerge,
As the developer team,
We want a fully automated training pipeline that can be triggered on fresh data, evaluates each new model version, and promotes the best one to production without manual intervention.

ID	Pipeline Stage	Requirement Description	MoSCoW	Finished
R-01	Data Ingestion	Store raw images and labels in an object-storage bucket	Must	Yes
R-02	Data Pre-processing	Split each new dataset into train/validation/test subsets (80/10/ 10) as soon as it is ingested.	Must	Yes
R-03	Model Training	Execute an automated training job that uses the current training set and logs all hyper-parameters and metrics.	Must	Yes
R-04	Model Validation	Evaluate the trained model on the validation set; compute mAP_{50}, $\text{mAPmAP}_{50\text{-}95}$, precision, and recall.	Must	Yes
R-05	Model Deployment	Use a tool to compare validation metrics across runs and mark the best-performing model as the *candidate for production.*	Must	Yes
R-06	Model Deployment	Promote the selected candidate to production and store the model artefact in object storage for inference use.	Must	Partly
R-07	Data Versioning	Version control the training data and use an object storage as the remote for it	Could	Yes

Goal 2: Dutch Houses Collection Process

In order to supply the detection model with up-to-date, house-level aerial images from across the Netherlands,
As Selin,
We want an automated collection service that retrieves imagery from public sources, guarantees each image shows exactly one house, and tags it with the correct `pid/vid`.

ID	Requirement Description	MoSCoW	Finished
R-08	Provide a REST endpoint that accepts a list of `VIDs` and stores imagery for those specific houses in an object storage.	Should	Yes
R-09	Provide a REST endpoint that accepts a Dutch city code and stores imagery for every house in that city in an object storage.	Could	Yes
R-10	Ensure every stored image is saved together with its corresponding `pid` and `vid`.	Should	Yes
R-11	Ensure every stored image depicts exactly one house.	Could	Partly

Goal 3: Inference/Detection Process

In order to populate the NOWATT dataset with reliable solarpanel information at scale,
As the developer team,
We want an inference workflow that runs the latest model on newly collected images, links every prediction to its unique house ID, and returns confidence-scored results ready for downstream energy-labelling.

ID	Requirement Description	MoSCoW	Finished
R-12	Orchestrate inference runs with an Orchestrator tool that loads the latest production detection model and processes new images in an object storage.	Must	Yes
R-13	Save an annotated copy of each processed image (bounding boxes + confidence scores) in a dedicated bucket.	Must	Yes
R-14	Export a results file containing `pid`, `vid`, image URL, and highest confidence per image in CSV format.	Must	Yes
R-15	Automatically map detection results into the NOWATT projects dataset	Could	No

2.6 Scenarios

For the above functional requirements with a Must and Should Tag, we will specify some scenarios, to make the requirement more clear, if needed.

R-02 — Split Dataset
Given a versioned dataset exists in object storage,
When the *Data Pre-processing* stage runs,
Then the data are shuffled once and split 80% train, 10% validation, 10% test, with the split recorded for reproducibility.

R-03 — Automated Training Job
Given the current training subset and default hyper-parameters,
When the *Model Training* stage is launched,
Then a training job shall execute to completion and log all hyper-parameters, epoch metrics, and artefacts.

R-04 — Model Evaluation
Given a freshly trained weight file,
When the *Model Validation* stage is executed,
Then the system shall compute mAP_{50}, $\mathrm{mAP}_{50\text{-}95}$, precision, and recall on the validation subset and store the results.

R-05 — Candidate Selection
Given metrics for the latest and previous runs,
When the *Model Deployment* stage compares validation results,
Then the model with the highest primary metric (mAP_{50}) shall be flagged as the new production candidate.

R-06 — Promote Model
Given a production candidate is flagged,
When the promotion step is approved (automatic or manual),
Then the candidate weights are copied to the "production" location in object storage and the model registry is updated.

R-08 — VID Endpoint
Given a client sends a REST request containing a list of `vid` identifiers,

When the *Collection Service* receives the request,
Then the service shall download one image per identifier and store it in object storage.

R-10 — Metadata Tagging
Given an image has been stored,
When the storage transaction completes,
Then the image metadata shall include its `pid`, `vid`, and acquisition source.

R-12 — Run Inference DAG
Given new images exist in the inference input bucket,
When the orchestrator triggers the latest production model,
Then each image shall be processed and its detections returned.

R-13 — Save Annotated Copy
Given inference produces bounding boxes and confidence scores,
When post-processing completes,
Then an annotated PNG/JPEG is saved to the dedicated “annotated” bucket.

R-14 — Export Results File
Given a batch of images has been processed,
When the batch job finishes,
Then a CSV file containing `pid`, `vid`, image URL, and highest confidence is written to object storage for downstream import.

3 Solution Strategy

In this section, we outline a structured strategy specifically designed to achieve the project goals described in the introduction, which include:

1. Continuous Training Pipeline
2. Dutch Houses Collection Process
3. Inference/Detection Process

We will systematically discuss each goal, detailing our strategic approach, explaining each step, and evaluating multiple technologies to select appropriate tools for the corresponding tasks.

3.1 Continuous Training Pipeline

The primary objective of the Continuous Training Pipeline is to automate the process of training, validating, and deploying an object-detection model capable of identifying solarpanels in aerial imagery. Fig. 1 illustrates a generic machine learning training pipeline, which serves as the architectural foundation for our project. Note that Data Validation, Model Tuning, Model Analysis, and Model Feedback **are not** within the scope of this project and are therefore indicated as crossed out in the figure. How this pipeline works at runtime is illustrated in chapter 5.2.

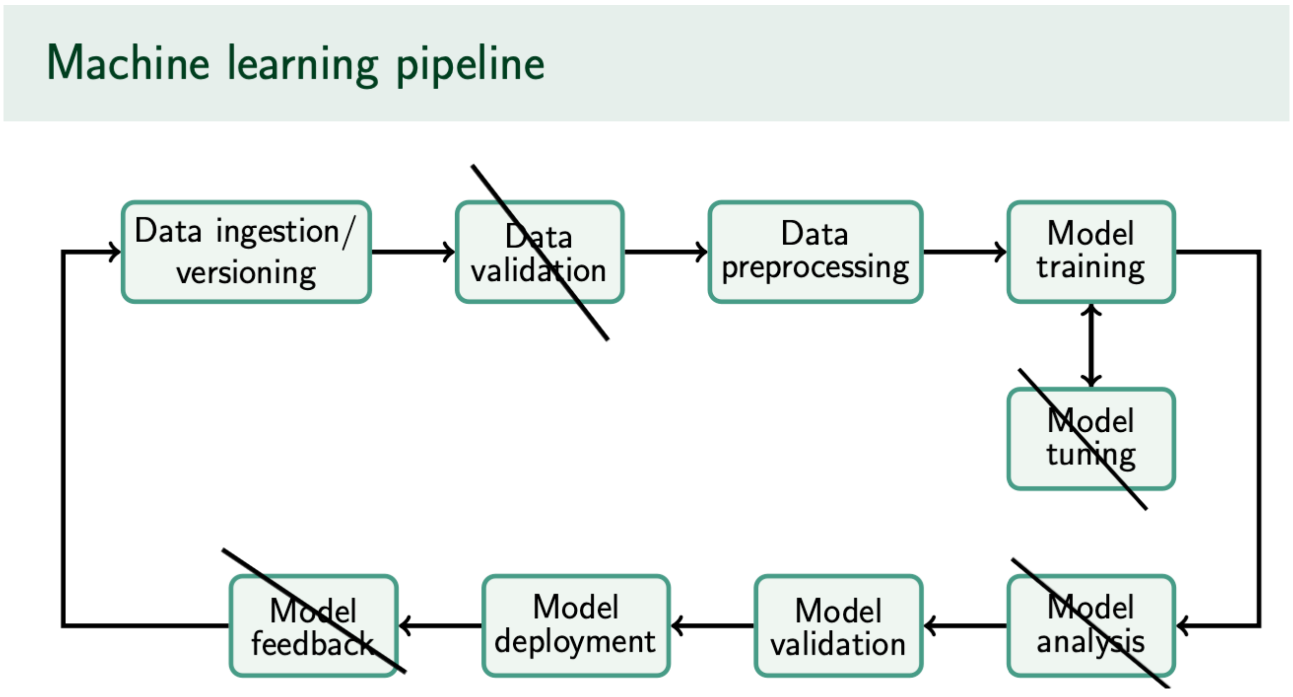

Figure 1: Machine Learning Training Pipeline

The pipeline consists of the following iterative steps, each detailed with specific considerations and chosen technologies. We selected Python as the programming language due to its widespread adoption, comprehensive ecosystem for data science, and robust libraries for computer vision tasks, such as Ultralytics and OpenCV [2]. Apache Airflow is used as an orchestrator to automate and manage all pipeline steps, with the rationale for Airflow and other selected tools elaborated further in section 2.4. On how and where AIrflow is used excatly you can see in the Runtime Diagrams in chapter 5.2 and 5.3 as well as in 4.2.3, where the DAGs we have are listed and explained.

1. **Data Ingestion and Versioning**

   This step involves ingesting, storing, and versioning labeled training images. We initially utilized an open-source, labeled aerial house dataset from southern Germany [3], enabling a rapid project start due to the tight 10-week timeline. An example of such an aerial image is shown in Fig. 2.

   The labeling information is stored separately in a '.txt' file using YOLO (You Only Look Once) format, which includes precise coordinates indicating solarpanel locations within the images. Further details on YOLO are discussed in Step 3 (Model Training).

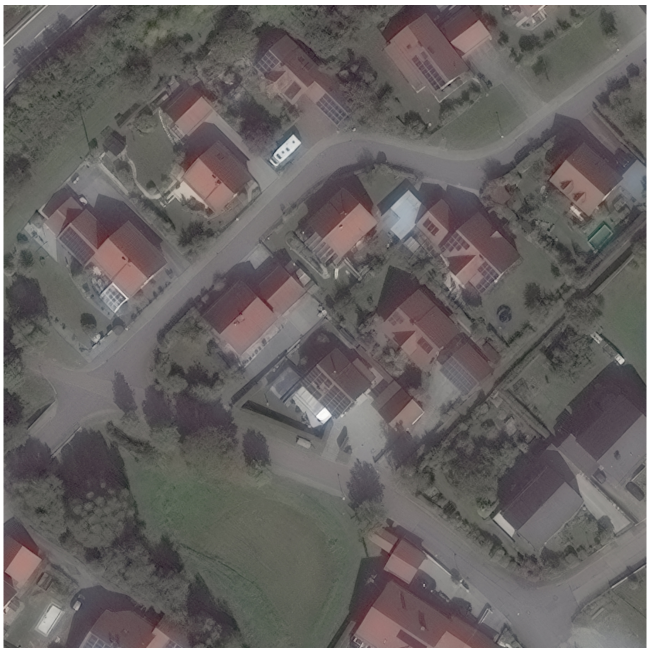

Figure 2: Example aerial imagery from southern Germany

   In the future, our own pipeline for automatically collecting Dutch house images could serve as an additional data source. However, due to project constraints, we did not label or utilize these newly collected images during the current project phase.

   We implemented MinIO as an object storage solution and Data Version Control (DVC) for versioning, connecting MinIO as a remote storage backend. Although implemented, DVC provided limited practical benefit due to the static nature of our initial dataset.

   Technology needed:

   - **Object Storage:** MinIO

- **Data Versioning Tool:** Data Version Control (DVC)

2. **Data Preprocessing**

   The original German dataset required minimal preprocessing, as it was already labeled and structured appropriately. Thus, preprocessing was limited to splitting the dataset into training, test, and validation subsets with an 80/10/10 ratio, ensuring effective model training and evaluation.

   Future Dutch house images, collected via our automated process, will also require minimal preprocessing, as images will already be standardized to contain exactly one house per image. Further details about this collection approach are outlined in chapter 2.2.

   Technology needed:

   - No additional specific technologies required for this step.

3. **Model Training**

   A supervised learning model is trained to detect solarpanels using the labeled imagery data. Due to our dataset's YOLO-specific labeling, we selected YOLOv8 as our object detection model. YOLOv8 provides several advantages for our project, including state-of-the-art accuracy, rapid inference speed, and efficient resource utilization, making it ideal for detecting small objects such as solarpanels in batches of high-resolution aerial imagery. Given the potential scale of hundreds of thousands of images, efficient batch processing capabilities are essential for practical deployment.

   Technology needed:

   - **Detection Model:** YOLOv8
   - **Experiment Tracking Tool:** MLflow, to log training parameters and facilitate reproducibility.

4. **Model Validation**

   Validation is conducted to evaluate the model's performance using key metrics, including mean Average Precision (mAP50 and mAP50-95), precision, recall, and validation losses (bounding box loss, classification loss, and distribution focal loss). These metrics offer comprehensive insight into model performance, particularly in object detection scenarios.

   MLflow captures these metrics during each training run, enabling automated comparison and selection of the optimal model version based on performance metrics. This ensures systematic and traceable model versioning and deployment.

   Technology needed:

   - **Experiment Tracking Tool:** MLflow
   - **Relational Database:** PostgreSQL, utilized by MLflow to store metadata, parameters, metrics, and model version details.

5. **Model Deployment**

   Based on model validation outcomes, MLflow automatically identifies the best-performing model for deployment, storing the selected model artifact in MinIO. This streamlined deployment approach simplifies version management, facilitates reproducibility, and ensures immediate availability of high-quality models for inference tasks.

   Technology needed:

   - **Experiment Tracking Tool:** MLflow
   - **Object Storage:** MinIO, for securely storing and retrieving model artifacts.

3.2 Dutch Houses Collection Process

This step addresses the lack of available aerial imagery for Dutch houses, which is crucial both for inference (detecting solarpanels on houses) and potentially for future model training. Hence, the data collection process plays an integral role across multiple phases of this project.
The key requirements of this data collection are:

- Each image must contain exactly one house to ensure automated inference results can be correctly mapped.
- Images must be stored along with unique house identifiers (`pid` and `vid`) to ensure accurate mapping and integration into the existing NOWATT project dataset, which contains detailed energy attributes for each house.

To fulfill these requirements, we utilize multiple publicly accessible external APIs. Details regarding the specific services and how they interact are explained comprehensively in *Section 3 (System Scope and Context)*, and runtime behavior is detailed further in *Section 5.1*.
This collection process will be accessed via two primary usage options:

- **City-based collection**: Input a city name to retrieve images of all houses within that city, ideal for generating future training datasets.
- **ID-based collection**: Input a list of house IDs (`pid/vid`) to fetch images of specific houses, essential for immediate integration into existing datasets.

We will implement these collection methods via an API Gateway developed using FastAPI, allowing clients to easily interact with the system through REST requests or via FastAPI's built-in interactive Swagger UI.

Technologies needed:

- **Python REST API Framework:** FastAPI
- **Object Storage:** MinIO

3.3 Inference/Detection Process

The inference phase involves applying the trained YOLO model to detect solarpanels on collected images. Given the project's emphasis on pipeline and process construction rather than model optimization, we utilize existing images stored in MinIO, and the inference is orchestrated through Airflow DAGs.
Processed images will be annotated with bounding boxes indicating solarpanel detections and associated confidence scores (ranging from 0 to 1, where values closer to 1 represent higher detection confidence). Annotated images are stored separately in MinIO for visual verification. Additionally, inference results—including house identifiers (`pid`, `vid`), image URLs, and the highest detection confidence per image—are saved in a CSV file. This format facilitates straightforward integration into the NOWATT project dataset. Confidence thresholds (e.g., above 0.8) can subsequently be applied to classify panel presence definitively as "yes" or "no," depending on project needs.

Technologies needed:

- **Orchestration Tool:** Apache Airflow (automation via DAGs)
- **Object Storage:** MinIO
- **Detection Model:** Trained YOLOv8 model

In Fig. 3 you can see an dutch housing image which ran through the inference process.

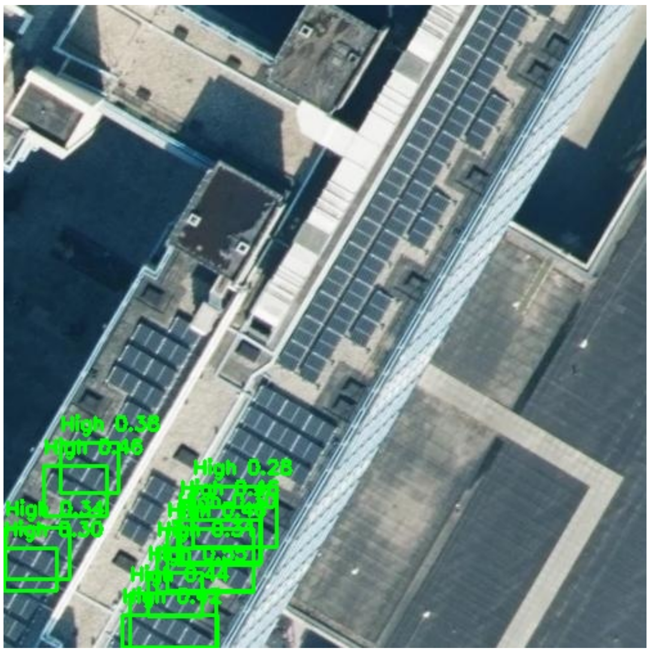

Figure 3: Example for Inference

3.4 Technology Selection and Rationale

Based on the strategies outlined above, we require the following technologies, each carefully chosen for their suitability to our project's constraints and goals.

Programming Language

The project's core language requires robust support for data science workflows and computer vision tasks. **Python** emerges as the optimal choice due to its extensive ecosystem (e.g., OpenCV, Ultralytics) and strong community support [2]. Alternatives like R (statistical analysis) and Julia (performance-focused) were considered but ultimately ruled out due to Python's superior versatility, broad adoption, and production readiness.

REST API Framework

FastAPI was selected over Flask and Django REST Framework for its built-in automatic OpenAPI documentation, intuitive Swagger UI, and ease of use for non-technical clients [4]. Additionally, as only a lightweight solution with minimal endpoints is required, the complexity and overhead of Flask or Django would be unnecessary.

Object Storage

Our object storage requirements are straightforward: store and retrieve images and model artifacts reliably, cost-effectively, and compatibly with the S3 protocol. Several open-source solutions were evaluated, such as MinIO [5], Deuxfleurs [6], SeaweedFS [7], Ceph [8]
We selected **MinIO** primarily due to the team's familiarity, simplicity of deployment, and full alignment with our technical requirements.

Experiment Tracking

Based on a comprehensive evaluation of open-source ML tracking tools by neptune.ai [9], the viable options included MLflow, Aim, Sacred and Omniboard, and TensorBoard. However, **MLflow** was uniquely suited as a comprehensive MLOps solution providing not only experiment tracking but also model deployment capabilities through MLflow Models and a centralized Model Registry.

Relational Database

PostgreSQL was selected as our relational database due to explicit recommendations by Apache Airflow (for DAG metadata storage) and MLflow (for experiment tracking and metadata persistence) [10], [11]. PostgreSQL's widespread industry acceptance further supports this choice.

Orchestration Tool

Following DataCamp's evaluation of workflow orchestration alternatives [12], several alternatives (Prefect, Dagster, Mage AI, Kedro, Luigi) were considered. However, **Apache Airflow** was chosen for its established DAG-based approach, robust monitoring interface, and the team's prior expertise—prioritizing rapid development and reliability over exploring functionally similar tools.

Containerization

Docker was selected to ensure deployment consistency, isolation, and reproducibility across development, testing, and production environments. Docker's widespread adoption simplifies containerized deployments and maintains industry-standard compatibility.

Deployment Platform

Given the project's short-term and demonstrational nature, several deployment options were evaluated:

- Saxion's provided AWS account (managed cloud)
- Alternative cloud providers (Azure, GCP)
- On-premises hosting

Saxion's AWS account was ultimately selected because it provides:

- Zero-cost institutional access
- Managed infrastructure with minimal administrative overhead
- Sufficient resources for effective demonstration and short-term use

This approach satisfies deployment requirements without introducing unnecessary complexity or cost.'

4 System Scope and Context

This section provides an **overview of the system landscape**, showing **who interacts with our system** and **how it fits into the environment**. It includes high-level context diagrams, focusing on the **Solarpanel Detection System**.

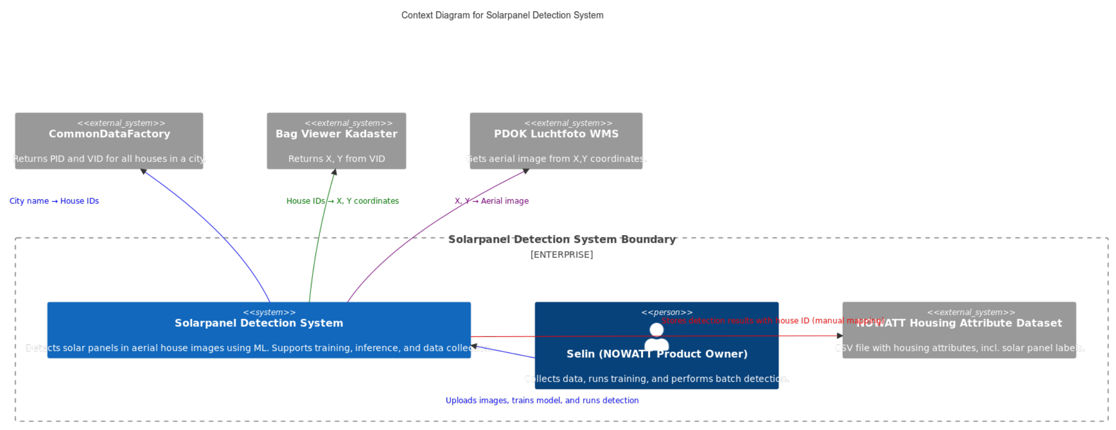
Context Diagram for Solarpanel Detection System
<<external_system>>
CommonDataFactory
Returns PID and VID for all houses in a city.
<<external_system>>
Bag Viewer Kadaster
Returns X, Y from VID
<<external_system>>
PDOK Luchtfoto WMS
Gets aerial image from X,Y coordinates.
City name → House IDs
House IDs → X, Y coordinates
X, Y → Aerial image
Solarpanel Detection System Boundary
[ENTERPRISE]
<<system>>
Solarpanel Detection System
Detects solar panels in aerial house images using ML. Supports training, inference, and data collection.
<<person>>
Selin (NOWATT Product Owner)
Collects data, runs training, and performs batch detection.
Stores detection results with house ID (manual mapping)
<<external_system>>
NOWATT Housing Attribute Dataset
CSV file with housing attributes, incl. solar panel labels.
Uploads images, trains model, and runs detection

Figure 4: C4 Context Diagram

The **Solarpanel Detection System** includes all the three project goals, so Training Pipeline, Inference and Data Collection.

Key Stakeholders and External Systems

Stakeholders

- **Selin (NOWATT Product Owner)**: Can collect housing images, run trainings and batch detect solarpanels on images.

External Services

- **CommonDataFactory**: Receives a **city name** and returns a **list of addresses**.
- **Bag Viewer Kadaster**: Takes an **address** and provides the corresponding **X, Y co-ordinates**.
- **PDOK Luchtfoto WMS**: Takes **X, Y coordinates** and returns the **aerial image** of a house.
- **NOWATT Housing Attribute Dataset**: Currently an CSV file (owned by Selin, the product owner of NOWATT Project) that will stores attributes for houses like if house has a solarpanel or not. Our System does actually not send data to the data storage for results. We store our detection results along with the house id, so our results can be mapped to this dataset with a short python script. This would need to be done by Selin.

5 Building Block View

This section outlines the overall structure of the Solarpanel Detection System, showing the main containers and their interactions.

5.1 Container View

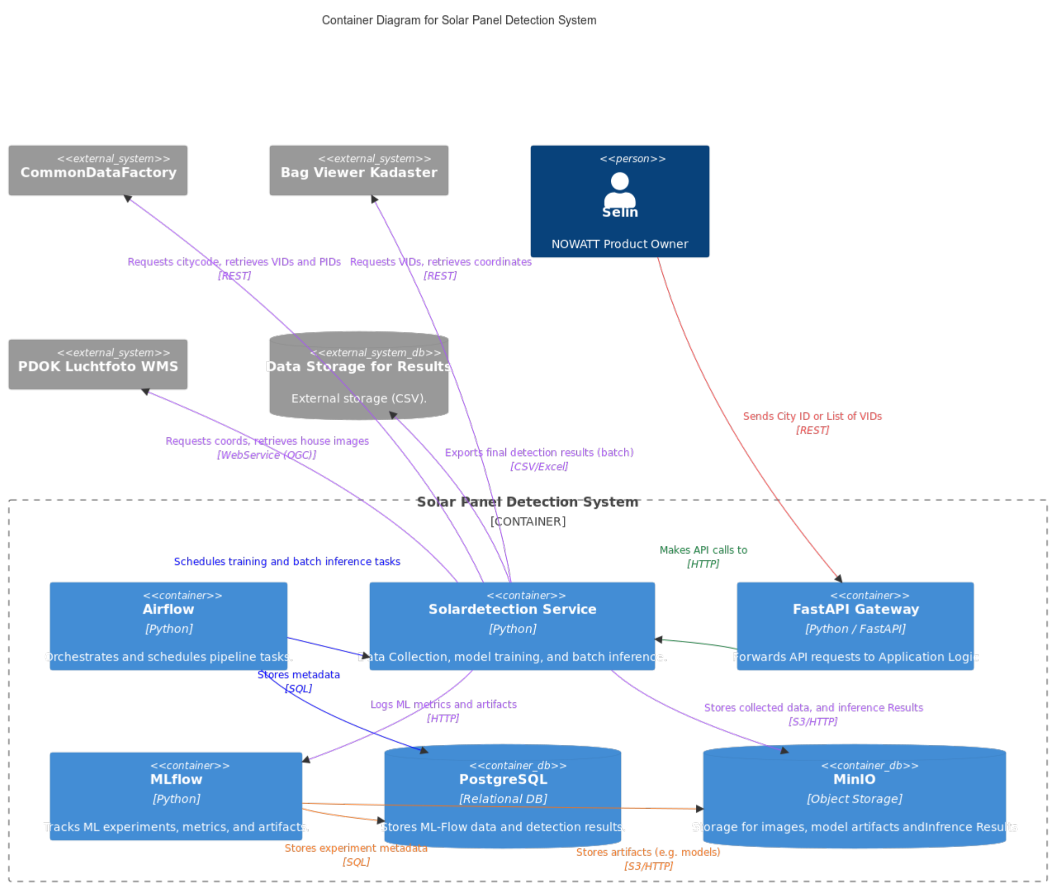
Container Diagram for Solar Panel Detection System
<<external_system>>
CommonDataFactory
<<external_system>>
Bag Viewer Kadaster
<<person>>
Selin
NOWATT Product Owner
Requests citycode, retrieves VIDs and PIDs
[REST]
Requests VIDs, retrieves coordinates
[REST]
<<external_system>>
PDOK Luchtfoto WMS
<<external_system_db>>
Data Storage for Results
External storage (CSV).
Sends City ID or List of VIDs
[REST]
Requests coords, retrieves house images
[WebService (OGC)]
Exports final detection results (batch)
[CSV/Excel]
Solar Panel Detection System
[CONTAINER]
Schedules training and batch inference tasks
Makes API calls to
[HTTP]
<<container>>
Airflow
[Python]
Orchestrates and schedules pipeline tasks
<<container>>
Solardetection Service
[Python]
Data Collection, model training, and batch inference.
<<container>>
FastAPI Gateway
[Python / FastAPI]
Forwards API requests to Application Logic
Stores metadata
[SQL]
Logs ML metrics and artifacts
[HTTP]
Stores collected data, and inference Results
[S3/HTTP]
<<container>>
MLflow
[Python]
Tracks ML experiments, metrics, and artifacts
<<container_db>>
PostgreSQL
[Relational DB]
Stores ML-Flow data and detection results
<<container_db>>
MinIO
[Object Storage]
Storage for images, model artifacts andInfrence Results
Stores experiment metadata
[SQL]
Stores artifacts (e.g. models)
[S3/HTTP]

Figure 5: C4 Container Diagram

The diagram above shows the internal structure of the Solarpanel Detection System and how it interacts with external systems and users:

- **Airflow** orchestrates the entire pipeline by scheduling data-collection, training, and batch-inference jobs.
- **Solardetection Service** handles house-ID lookups and imagery scraping, trains the ML model, and runs batch inference.

- **FastAPI Gateway** exposes a REST endpoint for clients (e.g. Selin) and forwards image-upload and inference requests to the Solardetection Service.
- **MLflow** tracks experiments—logging parameters and metrics to PostgreSQL and storing model artifacts to MinIO.
- **PostgreSQL** holds MLflow and Airflow metadata.
- **MinIO** provides object storage for raw images, trained model files, and batch-inference outputs.

5.2 Component View

This section provides a component-level view of three key containers: the **Solardetection Service**, **MLflow**, and **Airflow**. Other containers, such as external services, are not covered here, as they are less central to our architectural overview.

5.2.1 Solardetection Service Container

Figure 6 illustrates the internal components of the **Solardetection Service** container, structured into three distinct pipelines, each managing a specific aspect of the data workflow:

1. **Continuous Training Pipeline**: Manages the complete lifecycle of model training upon arrival of new training data, tracks model performance and metadata using MLflow, and deploys updated models.
2. **Inference Process**: Handles processing of newly uploaded images using the latest deployed model and stores detection results in MinIO.
3. **Data Collection Process**: Automates the retrieval of house IDs, coordinates, and aerial images for Dutch houses, tagging each image with a unique House ID for future inference or training purposes.

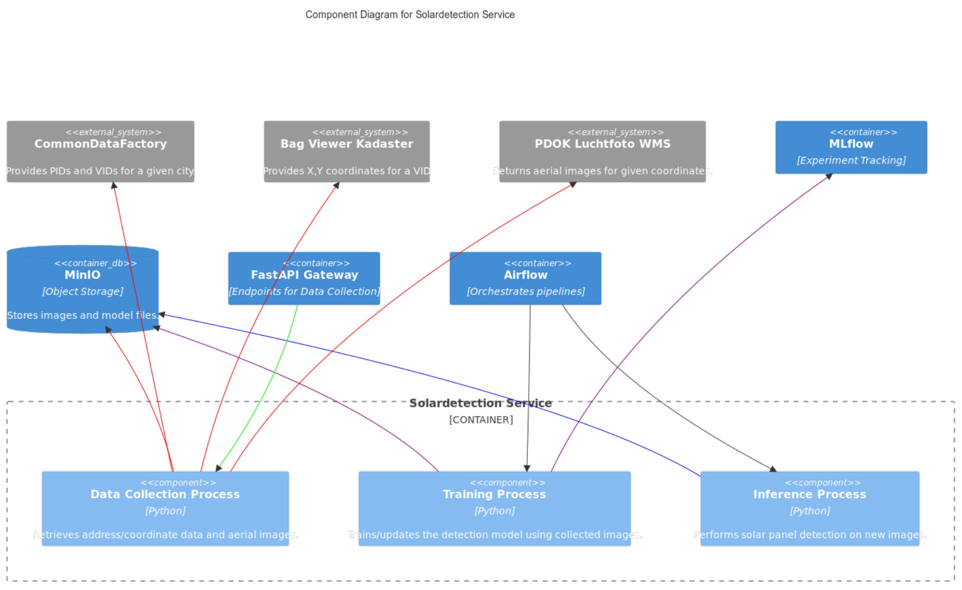

Figure 6: C4 Component Diagram – Solardetection Service

5.2.2 MLflow Container

Within this project, we specifically use **MLflow Tracking**, which provides the following functionality:

MLflow Tracking is an API and UI for logging parameters, code versions, metrics, and artifacts generated during machine learning experiments. It enables teams to visualize, compare, and analyze results across multiple model training runs.

A component diagram of MLflow is provided in Figure 7.

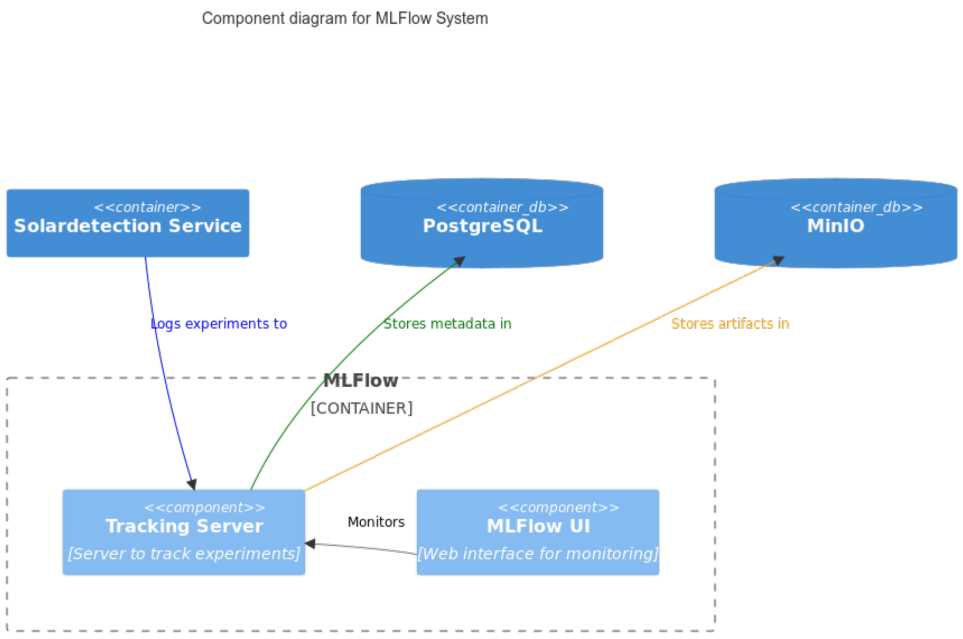

Figure 7: C4 Component Diagram – MLflow

5.2.3 Airflow Container

Apache Airflow orchestrates and schedules the execution of our solar panel detection workflows. Its key components are illustrated in Figure 8.

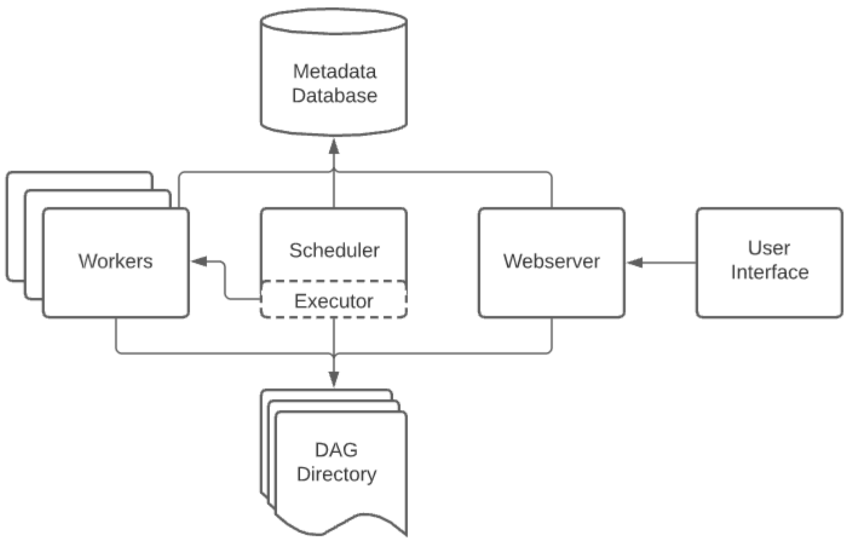

Figure 8: Component Diagram – Airflow.
Source: `https://www.astronomer.io/docs/learn/airflow-components/`

Most important things are:

1. **DAG Files**: Define the sequence and dependencies of tasks.
2. **Workers**: Execute Python scripts upon trigger by the scheduler.

We will now show you the DAG Files we have and their purpose. The Apache Airflow pipeline is structured around three primary DAG files, each serving a distinct purpose in our workflow:

1. **Model Deployment DAG (`split_traintest_solar_panel`)**:

   This DAG manages the deployment of trained YOLO models to the inference service. Its main responsibilities include:

   - Checking the MLflow Model Registry for newly registered models marked as "Production".
   - Downloading the latest production-ready model artifact from MLflow.
   - Converting the PyTorch model to ONNX format for optimized inference performance.
   - Deploying the ONNX model to the designated service directory.
   - Automatically restarting the inference service to apply the new model immediately.

   The DAG is scheduled to run daily, ensuring timely updates whenever new models reach production status.

2. **Batch Inference DAG (`batch_detection`)**:

   This DAG facilitates automated batch inference using the latest deployed YOLO model. Specifically, it:

   - Retrieves aerial images from MinIO object storage.
   - Runs inference to detect solar panels, annotating each image with bounding boxes and confidence scores.
   - Stores annotated images back in MinIO.
   - Updates a CSV manifest in MinIO, capturing the detection results, house identifiers, and confidence levels.

   This process runs hourly, ensuring frequent and automated inference processing on new image batches.

3. **Model Training DAG (`train_yolo_solar_panel_detection`)**:

   This DAG automates the entire model training workflow for YOLO-based solar panel detection. The key steps involved are:

   - Validating and downloading training data from MinIO, ensuring the dataset's integrity.
   - Training the YOLO model, with parameters logged to MLflow for comprehensive experiment tracking.
   - Evaluating the trained model against a validation dataset, recording essential metrics such as precision, recall, and mean Average Precision (mAP).
   - Registering the trained model in the MLflow Model Registry, promoting models to either Staging or Production based on performance metrics.

   This DAG is triggered manually due to the computational intensity and dataset size considerations associated with model training.

Each DAG clearly defines the sequence and dependencies of tasks, executed reliably by Airflow workers upon scheduling or manual triggers.

6 Runtime View

This section describes the runtime interactions of the three main pipelines within the **Solardetection Service** container, using dynamic runtime diagrams.

6.1 Data Collection Process

Figure 9 illustrates the runtime workflow for the Data Collection Process.
We have implemented two endpoints for initiating data collection:

- `/trigger_collection_by_city`: The NOWATT product owner can provide a city code to collect aerial images for all houses within that city. This approach executes all steps depicted in the diagram.
- `/trigger_collection_by_vids`: Alternatively, the product owner can provide a specific list of house IDs (VIDs) to directly fetch images for those houses. In this case, the system skips the second step of retrieving house IDs from the Common Data Factory.

The user will get nothing back. This is implemented as endoints with FastAPI, so technical and non technical users can run the data collection. Non Technical users can use the nice Swagger Web-UI.

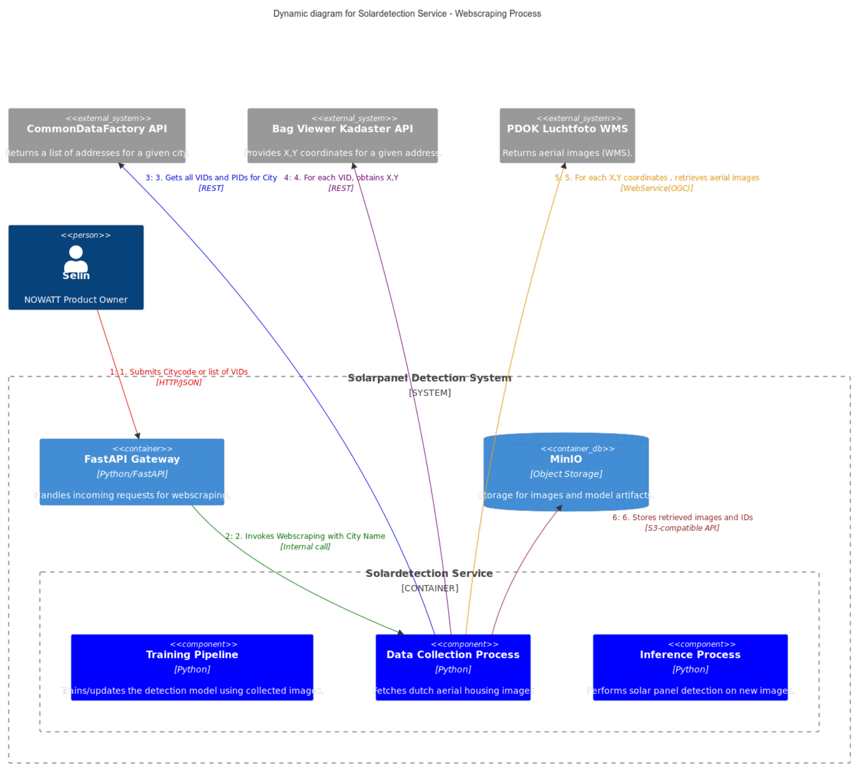
Dynamic diagram for Solardetection Service - Webscraping Process
<<external_system>>
CommonDataFactory API
Returns a list of addresses for a given city.
<<external_system>>
Bag Viewer Kadaster API
Provides X,Y coordinates for a given address.
<<external_system>>
PDOK Luchtfoto WMS
Returns aerial images (WMS).
3: 3. Gets all VIDs and PIDs for City
[REST]
4: 4. For each VID, obtains X,Y
[REST]
5: 5. For each X,Y coordinates , retrieves aerial images
[WebService(OGC)]
<<person>>
Selin
NOWATT Product Owner
1: 1. Submits Citycode or list of VIDs
[HTTP/JSON]
Solarpanel Detection System
[SYSTEM]
<<container>>
FastAPI Gateway
[Python/FastAPI]
Handles incoming requests for webscraping.
<<container_db>>
MinIO
[Object Storage]
Storage for images and model artifacts.
6: 6. Stores retrieved images and IDs
[S3-compatible API]
2: 2. Invokes Webscraping with City Name
[Internal call]
Solardetection Service
[CONTAINER]
<<component>>
Training Pipeline
[Python]
Trains/updates the detection model using collected images.
<<component>>
Data Collection Process
[Python]
Fetches dutch aerial housing images
<<component>>
Inference Process
[Python]
Performs solar panel detection on new images.

Figure 9: C4 Dynamic Diagram – Data Collection Process

6.2 Training Pipeline

The runtime sequence for the Continuous Training Pipeline (Fig. 10) is as follows:

1. **Data upload** – The data owner places new, labelled images in the training bucket of MinIO.
2. **DAG trigger** – Airflow's *Training DAG* periodically checks for new training data and starts the Training process inside the Solardetection Service.
3. **Model training** – The Trainingprocess pulls the images from MinIO, trains the YOLO model, and streams metrics and parameters to MLflow.
4. **Artefact & metadata storage** –
   - MLflow registers the *best.pt* model in its Model Registry and stores the file back in MinIO.
   - MLflow also writes run metadata (run ID, metrics, model URI, stage) to PostgreSQL.

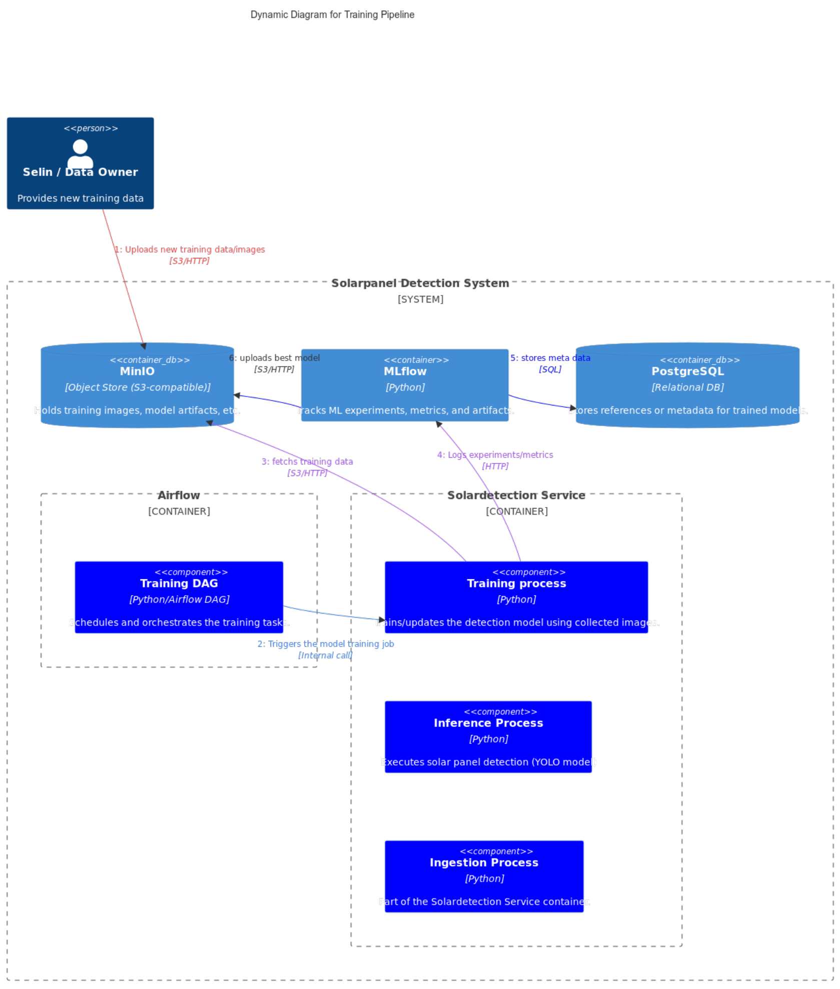
Dynamic Diagram for Training Pipeline
<<person>>
Selin / Data Owner
Provides new training data
1: Uploads new training data/images
[S3/HTTP]
Solarpanel Detection System
[SYSTEM]
<<container_db>>
MinIO
[Object Store (S3-compatible)]
Holds training images, model artifacts, etc.
6: uploads best model
[S3/HTTP]
<<container>>
MLflow
[Python]
Tracks ML experiments, metrics, and artifacts.
5: stores meta data
[SQL]
<<container_db>>
PostgreSQL
[Relational DB]
Stores references or metadata for trained models.
3: fetchs training data
[S3/HTTP]
4: Logs experiments/metrics
[HTTP]
Airflow
[CONTAINER]
Solardetection Service
[CONTAINER]
<<component>>
Training DAG
[Python/Airflow DAG]
Schedules and orchestrates the training tasks.
<<component>>
Training process
[Python]
Trains/updates the detection model using collected images.
2: Triggers the model training job
[Internal call]
<<component>>
Inference Process
[Python]
Executes solar panel detection (YOLO model)
<<component>>
Ingestion Process
[Python]
Part of the Solardetection Service container

Figure 10: C4 Dynamic Diagram – Training Pipeline

6.3 Inference Process

Figure 11 demonstrates the runtime workflow for the Inference Process:

1. Inference images are already collected during Data Collection process and are ready for Inference.
2. The Airflow DAG is either triggered manually or runs on a scheduled interval.
3. Detection is performed on all house images using the YOLOv8 model; labeled images are then moved to a separate folder.
4. Detection confidence scores along with the corresponding House IDs are stored in a CSV file in MinIO

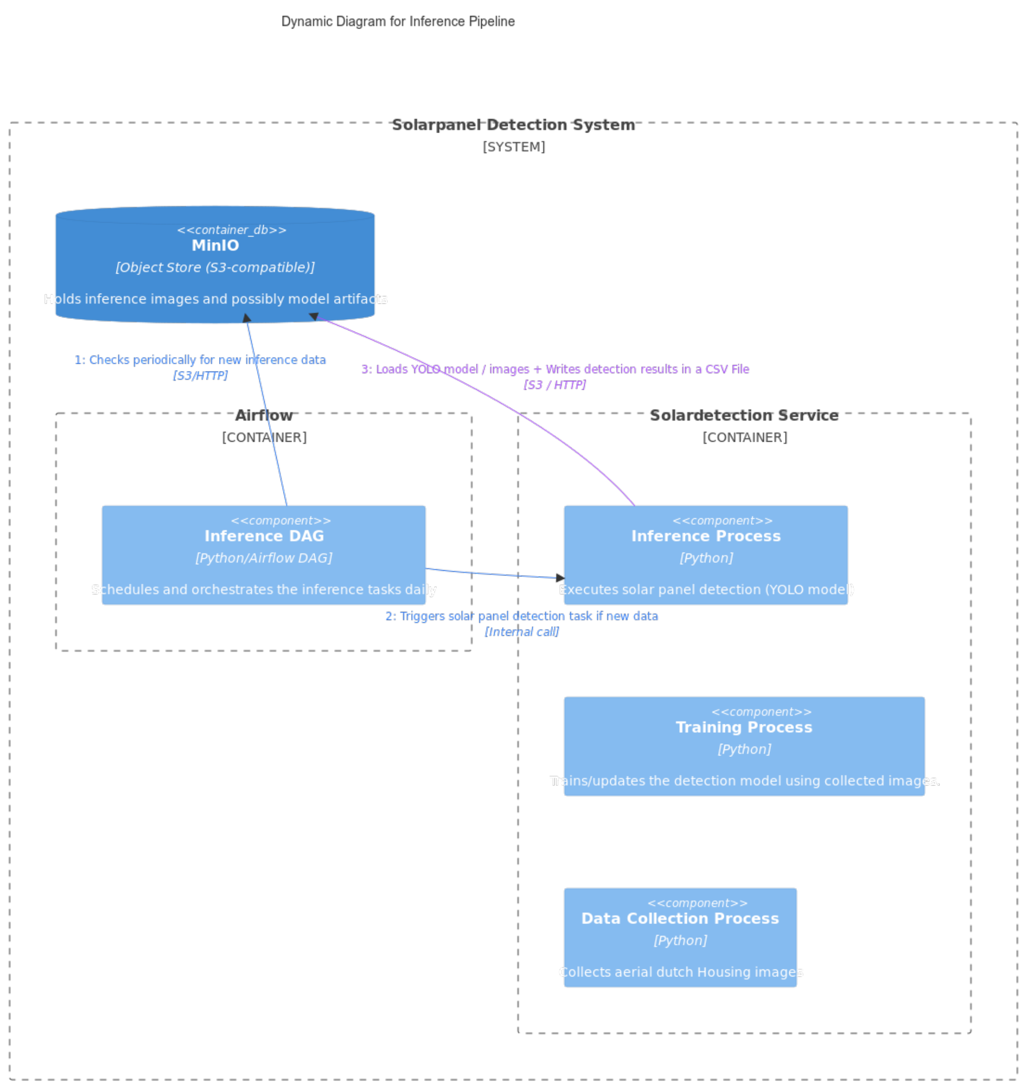
Dynamic Diagram for Inference Pipeline
Solarpanel Detection System
[SYSTEM]
<<container_db>>
MinIO
[Object Store (S3-compatible)]
Holds inference images and possibly model artifacts
1: Checks periodically for new inference data
[S3/HTTP]
3: Loads YOLO model / images + Writes detection results in a CSV File
[S3 / HTTP]
Airflow
[CONTAINER]
Solardetection Service
[CONTAINER]
<<component>>
Inference DAG
[Python/Airflow DAG]
Schedules and orchestrates the inference tasks daily
<<component>>
Inference Process
[Python]
Executes solar panel detection (YOLO model)
2: Triggers solar panel detection task if new data
[Internal call]
<<component>>
Training Process
[Python]
Trains/updates the detection model using collected images.
<<component>>
Data Collection Process
[Python]
Collects aerial dutch Housing images

Figure 11: C4 Dynamic Diagram – Inference Process

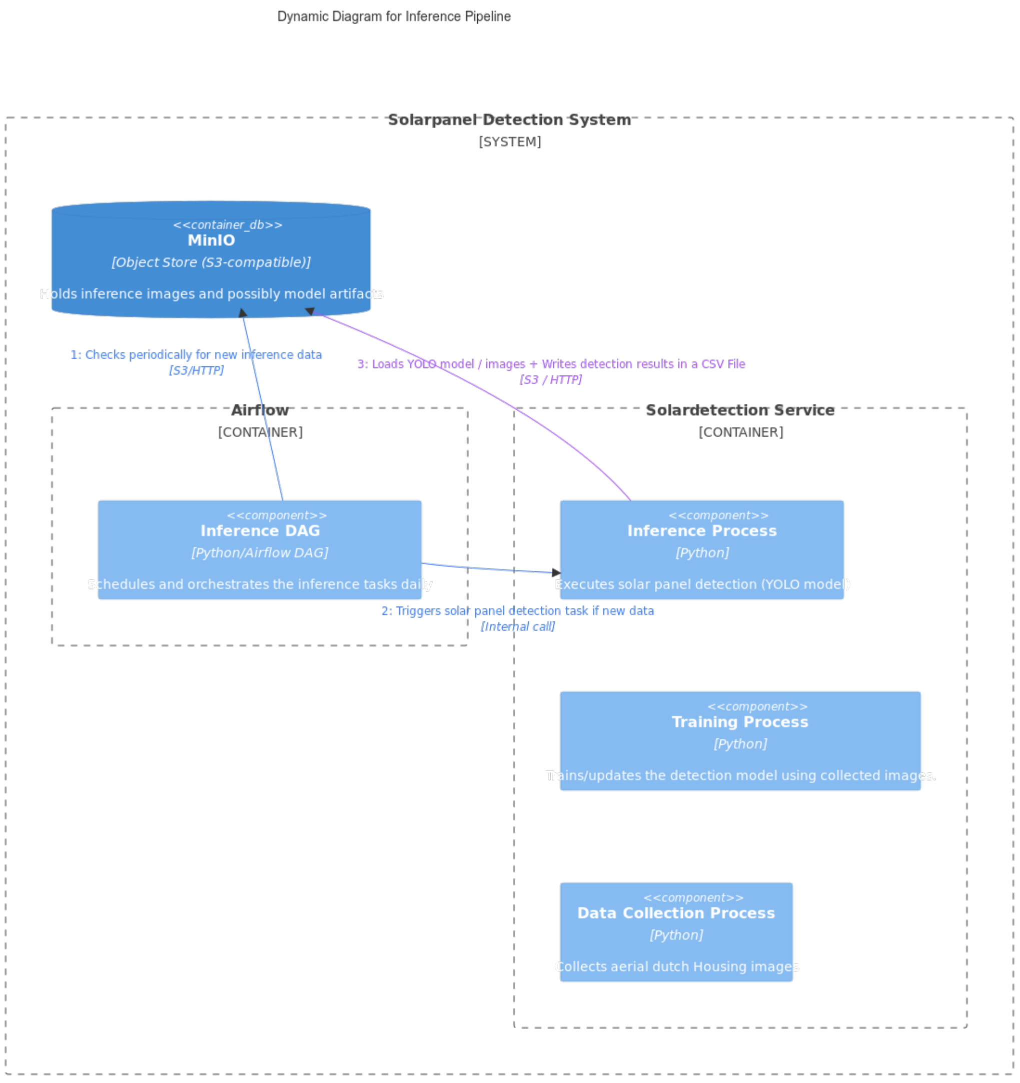
Dynamic Diagram for Inference Pipeline
Solarpanel Detection System
[SYSTEM]
<<container_db>>
MinIO
[Object Store (S3-compatible)]
Holds inference images and possibly model artifacts
1: Checks periodically for new inference data
[S3/HTTP]
3: Loads YOLO model / images + Writes detection results in a CSV File
[S3 / HTTP]
Airflow
[CONTAINER]
Solardetection Service
[CONTAINER]
<<component>>
Inference DAG
[Python/Airflow DAG]
Schedules and orchestrates the inference tasks daily
<<component>>
Inference Process
[Python]
Executes solar panel detection (YOLO model)
2: Triggers solar panel detection task if new data
[Internal call]
<<component>>
Training Process
[Python]
Trains/updates the detection model using collected images.
<<component>>
Data Collection Process
[Python]
Collects aerial dutch Housing images

Figure 12: C4 Dynamic Diagram – Inference Process

7 Deployment and Continuous Deployment Strategy

The Solar Panel Detection project is deployed using Saxion's AWS account with Docker containers running on EC2 instances. Deployment is automated via a GitLab CI/CD pipeline, ensuring consistent and scalable delivery of services.

7.1 Deployment Architecture

The system architecture consists of multiple containerized microservices deployed on AWS EC2 instances, including:

- Solar panel detection service
- FastAPI gateway
- PostgreSQL database
- MinIO object storage (S3-compatible)
- MLflow for machine learning experiment tracking
- Airflow for workflow orchestration

7.2 Continuous Deployment Pipeline

The CD pipeline is implemented through GitLab CI/CD with the following stages:

1. **Build Stage**: Constructs Docker images for all services using `docker-compose.build.yml`
2. **Deploy Stage**: Automatically deploys to EC2 instances when changes are merged to the main branch

7.3 AWS Infrastructure

The deployment leverages several AWS services:

- **EC2 Instances**: Hosts the containerized application stack
- **IAM Roles**: Provides necessary permissions for service communication
- **Security Groups**: Controls network access between services
- **SSH Key-based Authentication**: Secures the GitLab-to-EC2 deployment process

7.4 Deployment Workflow

When code is merged to the main branch, the following automated process occurs:

1. GitLab CI/CD pipeline is triggered
2. Docker images are built and tagged with commit SHA
3. SSH connection is established to the EC2 instance
4. Latest code is pulled from the repository
5. Docker Compose brings down existing services
6. Updated containers are built and deployed with `docker-compose up -d -build`

7.5 Configuration Management

Environment-specific configurations are managed through:

- Environment variables in the `.env` file for local settings
- GitLab CI/CD variables for sensitive credentials
- Docker Compose environment files for service communication

This deployment strategy ensures consistency across environments, facilitates rapid iteration, and maintains high availability of the solar panel detection system.

This deployment strategy ensures consistency across environments, facilitates rapid iteration, and maintains high availability of the solar panel detection system.

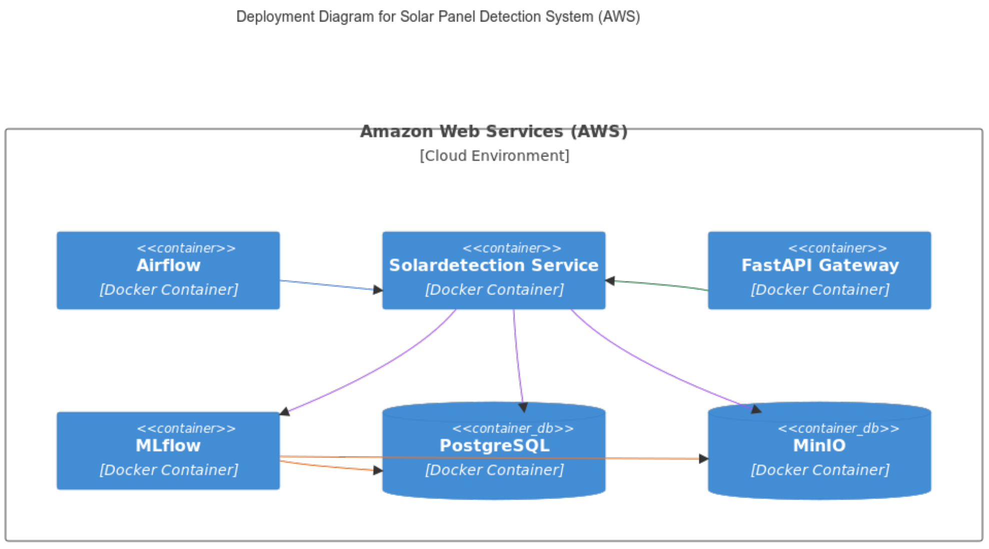

Figure 13: C4 Deployment Diagram

8 Open Problems

Several unresolved technical challenges remain in the current implementation, primarily related to dataset consistency and image retrieval accuracy. These issues present clear opportunities for future development and refinement.

8.1 Dataset Consistency and Model Accuracy

The initial object detection model has been trained using a publicly available, pre-labeled aerial imagery dataset from Germany. However, the inference phase targets images of Dutch houses. This regional mismatch in the training and inference datasets may significantly affect model accuracy due to differences in architectural styles, roof characteristics, solarpanel types, and geographical features. In Fig. 14 you can see that the pure black solarpanels are not detected. Maybe because the initial dataset is too old and these design of solarpanels is newer.
A future enhancement could involve labeling Dutch house imagery collected through our automated data ingestion process. By incorporating these localized datasets into model retraining cycles, we could substantially improve detection accuracy specific to Dutch housing.

8.2 Dynamic Bounding Box Calculation for Image Fetching

When fetching aerial imagery for individual houses, there is a critical requirement to ensure each image contains exactly one house. Currently, a fixed bounding box size and resolution are used, which does not adequately handle varying house sizes and could inadvertently include neighboring houses in a single image. In Fig. 14 you can see that there are multiple houses on that image, and there is a detection, on a house, which was probably not to be meant for that picture/house ID.
A more effective solution involves dynamically calculating the bounding box size based on actual house dimensions. Such dimensions (e.g., floor plans or house area in square meters) can potentially be retrieved from external APIs like *Kadaster BAG Viewer* or *CommonDatafactory*. Instead of a static bounding box calculation, as exemplified in the following simplified snippet:

```
bbox = f"{x - offset},{y - offset},{x + offset},{y + offset}"
width, height = 500, 500
```

we propose a dynamic calculation approach, roughly outlined as:

```
bbox = calculate_bbox(x, y, house_dimensions)
width, height = calculate_image_resolution(house_dimensions)
```

Implementing such a dynamic bounding box solution would significantly improve the precision of collected images, subsequently enhancing both inference accuracy and dataset quality for potential retraining of the detection model.
These unresolved technical aspects provide clear directions for future iterations, aiming for a more robust and accurate solarpanel detection pipeline.

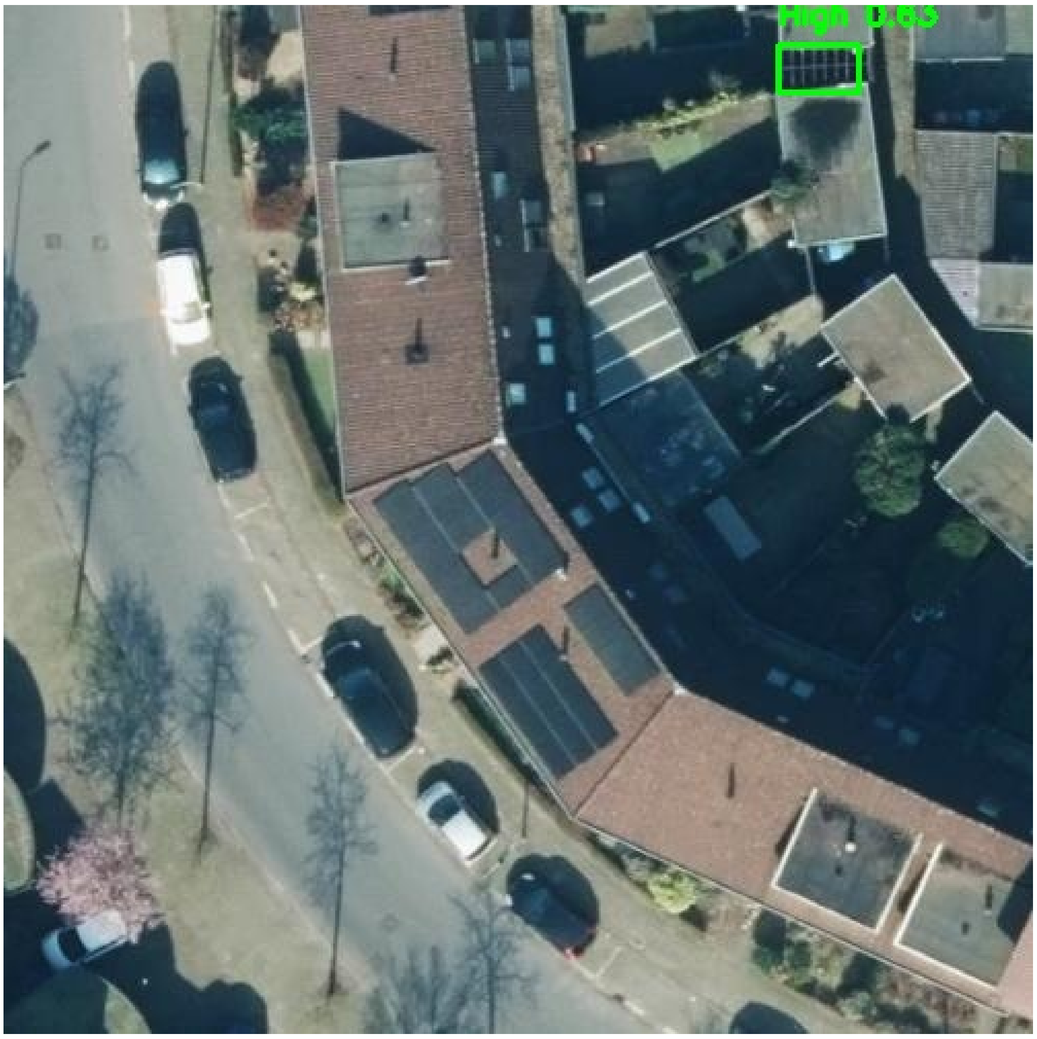

Figure 14: Open Problems

References

[1] S. U. of Applied Sciences. “Nowatt project - ambient intelligence.” (2024), [Online]. Available: `https://www.saxion.nl/onderzoek/smart-industry/ambient-intelligence/nowatt` (visited on 04/18/2025).

[2] S. Overflow. “Developer survey 2024.” (2024), [Online]. Available: `https://survey.stackoverflow.co/2024/technology#most-popular-technologies-language-prof` (visited on 04/18/2025).

[3] C. N. C.
bibinitperiod F. Pacifici. “A solar panel dataset of very high resolution satellite imagery to support the sustainable development goals.” (2023), [Online]. Available: `https://www.nature.com/articles/s41597-023-02539-8` (visited on 04/18/2025).

[4] Tiangolo. “Fastapi official documentation.” (2025), [Online]. Available: `https://fastapi.tiangolo.com/` (visited on 04/18/2025).

[5] I. MinIO. “Minio: High performance object storage.” (2025), [Online]. Available: `https://min.io/` (visited on 04/18/2025).

[6] D. Cooperative. “Garage distributed object storage.” (2025), [Online]. Available: `https://garagehq.deuxfleurs.fr/` (visited on 04/18/2025).

[7] C. Lu and Contributors. “Seaweedfs: Simple and highly scalable distributed file system.” (2025), [Online]. Available: `https://github.com/seaweedfs/seaweedfs` (visited on 04/18/2025).

[8] T. C. Authors. “Ceph: The future of storage.” (2025), [Online]. Available: `https://ceph.io/en/` (visited on 04/18/2025).

[9] Neptune.ai. “13 best tools for ml experiment tracking and management in 2025.” (2025), [Online]. Available: `https://neptune.ai/blog/best-ml-experiment-tracking-tools` (visited on 04/18/2025).

[10] T. A. S. Foundation. “Set up a database backend.” (2025), [Online]. Available: `https://airflow.apache.org/docs/apache-airflow/stable/howto/set-up-database.html` (visited on 04/18/2025).

[11] M. Project. “Mlflow.” (2025), [Online]. Available: `https://mlflow.org/docs/latest/tracking/` (visited on 04/18/2025).

[12] D. inc. “Top 5 airflow alternatives for data orchestration.” (2025), [Online]. Available: `https://www.datacamp.com/blog/airflow-alternatives` (visited on 04/18/2025).