



## Project Showcase

Submission for Apple Developer Institute for AIML

# AI-Driven PPE Monitoring System for Achieving a Safer Workplace

Safety Compliance Monitoring (SCM)

Portfolio by Kemal Dwi Hedy Muhammad







## Problem Statement

Safety Compliance Monitoring

# 64% increase

in the fatality rate was caused by workplace accidents that occurred **between 2020 and 2022**.

<https://goodstats.id>

# Over 38k more

workplace accidents were recorded in the first semester of **2024 compared to 2023**.

<https://satudata.kemnaker.go.id>

# Up to 85%

of workplace accidents are caused by an **improper or negligent use of personal protective equipment (PPE)**.

<https://journal.universitaspahlawan.ac.id/index.php/prepotif/article/view/442/360>



## What if...

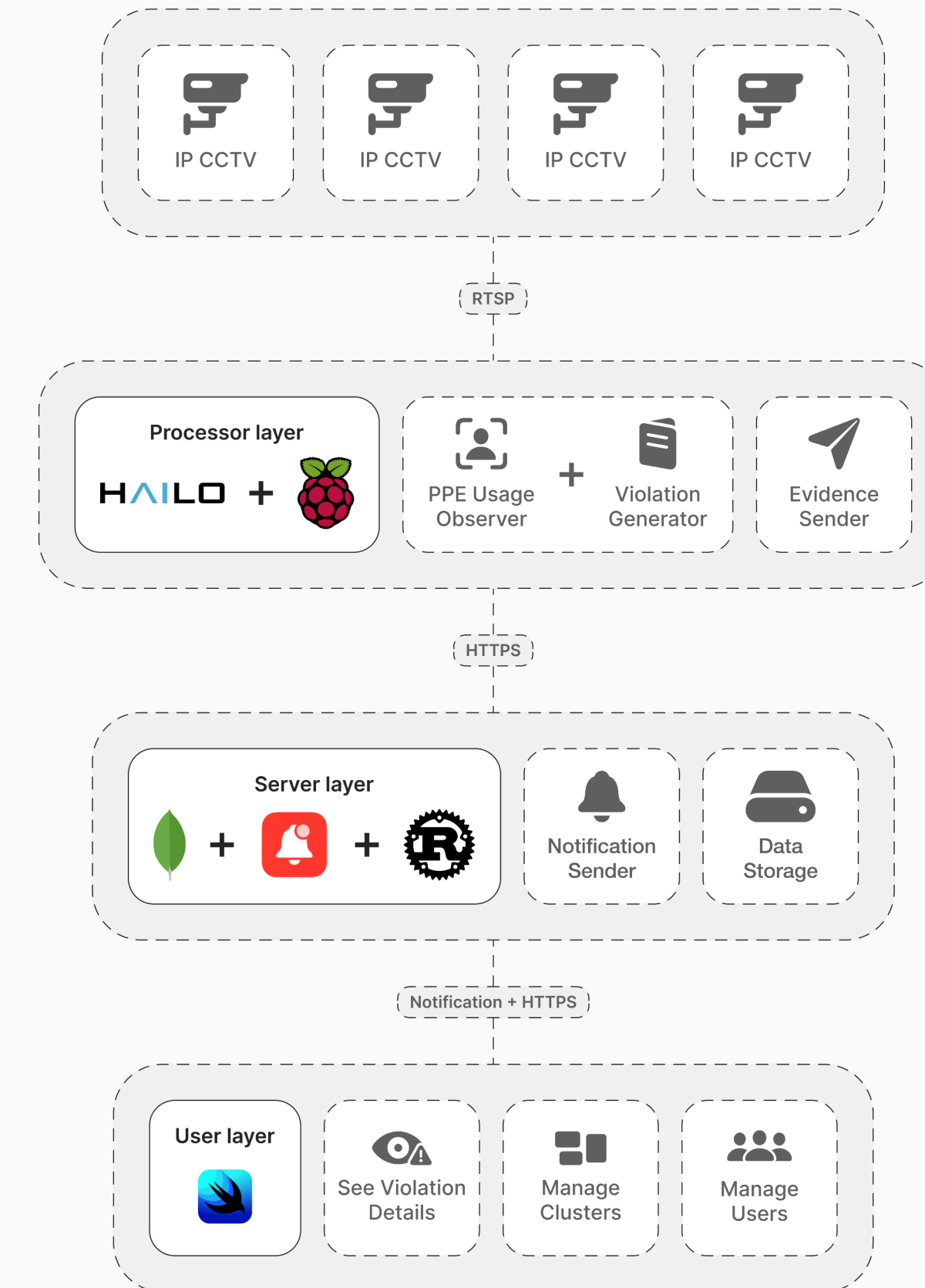
there's a system that **notifies safety supervisors** whenever a **violation towards PPE usage is present**?

### Integration of IP CCTV, AI, and Edge Devices

By leveraging **IP CCTVs** with **AI computer vision** model on an **edge device** such as **Raspberry Pi, SCM** can achieve an **automated PPE detection**, and **edge processing** without server bottleneck.

### Real-time Alert System

Supervisors receive **immediate mobile notifications** **when violations occur**, such as: the type of violation (missing hard hat, safety vest, etc.), details, and snapshot or evidence of the violation.



Distributed edge computing with multi-camera processing (4 cameras per Raspberry Pi + Hailo-8L). Violations are detected locally and communicated via HTTPS for real-time iOS notifications.

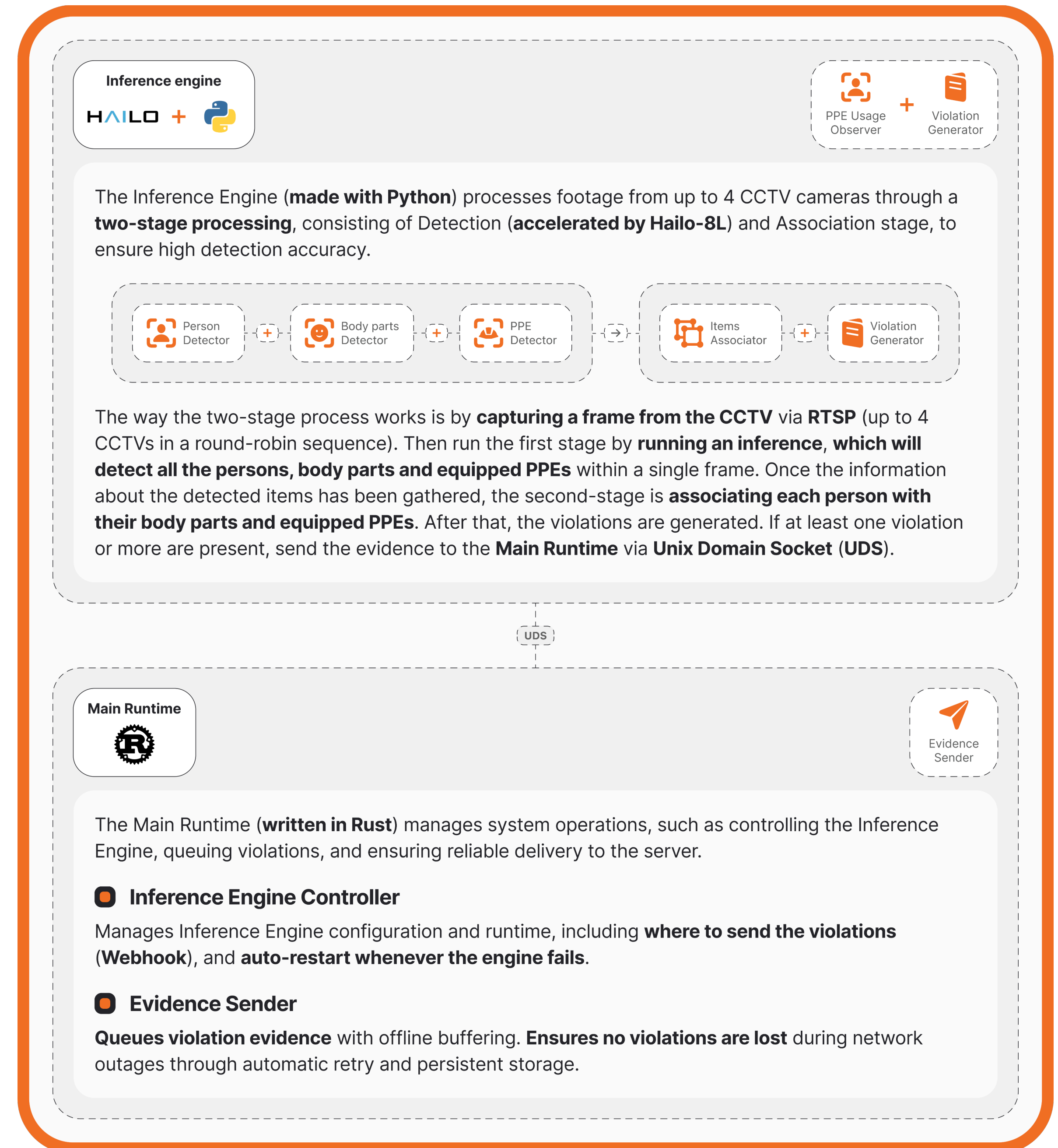




# Processor Layer

Hybrid Python+Rust Architecture for Edge AI

The processor layer consists of two important parts: the **Inference Engine**, which handles the **AI inference** processing up to **4 camera feeds simultaneously**, and the **Main Runtime** which **manages system reliability**, such as configuration, offline buffering, and server communication. **These components communicate via Unix Domain Socket (UDS)**, which is a communication protocol for programs within the same host. This separation provides **fault isolation** (Inference Engine crashes don't affect the Main Runtime) **while enabling multi-camera processing**. This architecture runs on **Raspberry Pi 5** with **Hailo-8L AI accelerator** (13 TOPS), enabling **real-time edge processing**.





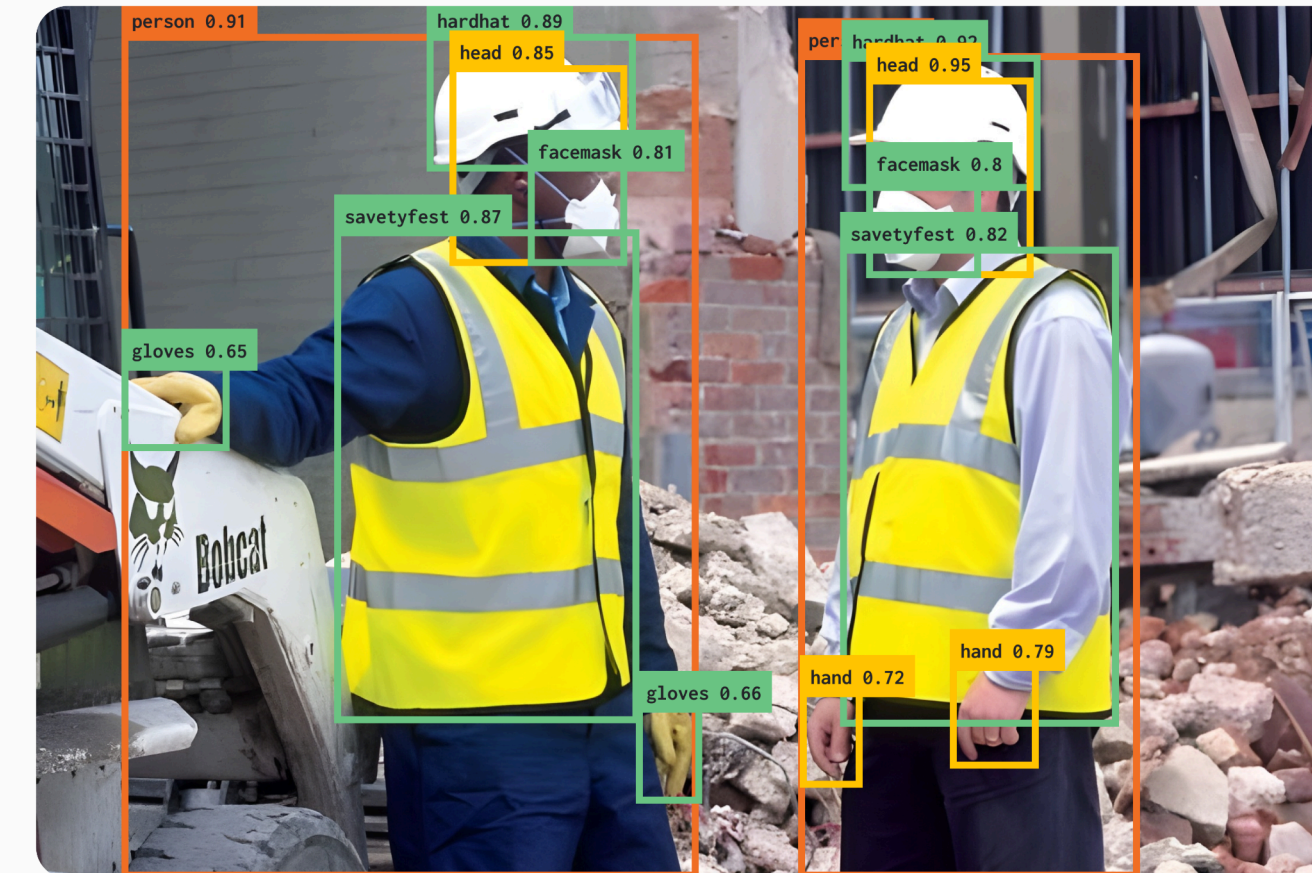


# Two-stage Processing

## The Heart of Compliance Monitoring

This is the most crucial part of the entire system. It determines **whether someone complies with PPE usage or not**. This process consists of two stages, the first one is the **detection**, where an **input frame is received and then inferred by an AI model** (I will cover about it later) to detect persons, body parts, and PPEs within the frame. The second stage is the **association**, where all the **detected body parts and PPEs are associated with each person** by checking if the item's bounding box is within the person's bounding box. Once the evidences are gathered, **it is sent to the Main Runtime via UDS**. The process **repeats as long as the Inference Engine is running**.

### Detection Stage



The detection stage is where the **Hailo-8L** does its job of **running an inference with an AI model** that's capable of **detecting every occurrence of 8 different classes** within a single frame. The available classes are:

- person
- hand
- safetyvest
- gloves
- head
- face
- hardhat
- facemask

### Association Stage — 1 / 2



In this stage, the extracted detection information such as class name, bounding box coordinates, and confidence will be passed through another two associators:

#### Belonging Associator

Within this associator, **each body part and PPE, will be assigned to a single person** based on the containment of the item's bounding box. If the item's bounding box is within the person's bounding box, then the item belongs to the person.

#### Violation Associator

Once body parts and PPE are associated with each person, this associator **determines which violations exist based on compliance rules** (explained on the next slide).





### Association Stage — 2 / 2



#### Belonging Associator



Output of Detection Stage

Associated Body Parts and PPEs Based on Containment

The diagram above illustrates how the Belonging Associator works. The left image shows the output from the Detection Stage, where all items detected but not yet organized. The Belonging Associator then **iterates through each detected body part and PPE item to check if it belongs to one of the detected persons**. The checking is done by verifying if **each corner of the item's bounding box is within the person's bounding box**. The right image illustrates the result, where each person is now associated with their body parts and PPE. Notice the **person with the blue bounding box** has all items **marked in blue**, while the **person with the pink bounding box** has all items **marked in pink**. The snippet on the right shows the logic behind containment checking.

```
def is_contained(
    inner_bbox: Dict,
    outer_bbox: Dict,
    margin: float = 0.05
) -> bool:
    width = outer_bbox.xmax() - outer_bbox.xmin()
    height = outer_bbox.ymax() - outer_bbox.ymin()

    expanded_xmin = outer_bbox.xmin() - margin * width
    expanded_ymin = outer_bbox.ymin() - margin * height
    expanded_xmax = outer_bbox.xmax() + margin * width
    expanded_ymax = outer_bbox.ymax() + margin * height

    return (
        inner_bbox.xmin() >= expanded_xmin
        and inner_bbox.ymin() >= expanded_ymin
        and inner_bbox.xmax() <= expanded_xmax
        and inner_bbox.ymax() <= expanded_ymax
    )
```

Containment Checking Snippet

#### Violation Associator

After the Belonging Associator assigns items to persons, the Violation Associator determines compliance status on each detected person based on these visibility-aware rules:



##### Glove Detection

- Hand + no glove → **violation** (missing)
- Hand + glove → **violation** (improper\*)
- No hand + glove → **compliant** (covered)
- No hand + no glove → **skip**



##### Hardhat Detection

- Head + no hardhat → **violation** (missing)
- Head + hardhat → **compliant**
- No head + hardhat → **skip**
- No head + no hardhat → **skip**



##### Facemask Detection

- Face + no facemask → **violation** (missing)
- Face + facemask → **violation** (improper\*)
- No face + facemask → **compliant** (covered)
- No face + no facemask → **skip**



##### Safetyvest Detection

- No safetyvest → **violation** (missing)
- Safetyvest → **compliant**
- Person too small (< 5% frame's width) → **skip**

\*Improper: PPE is visible but the body part it should cover is still visible, indicating incorrect usage.





## Creating the **AI Model**

### Finding the Right Dataset

Building an accurate PPE detection model for the **two-stage** processing required training on a **well-annotated dataset** and **good-quality images**. My initial thoughts were to search an existing dataset on **Roboflow Universe**, which **revealed a frustrating reality**: most PPE datasets that I could find were either **poorly annotated** (loose bounding boxes that don't fit objects tightly) or **used low-quality, pre-processed images** that would **degrade even further** during training augmentation (see figures on the right). After getting frustrated with searching the dataset on *Roboflow Universe*, I decided to do a simple Google search, where I came upon the **SH17 dataset**.



poorly and loosely annotated image (left) vs tightly annotated image (right)



example of images that's already been pre-processed (resized and stretched into a 1:1 aspect ratio).





## Creating the AI Model

### Exploring the SH17 Dataset

The **Safe Human** dataset, consisting of **17 different classes** (referred to as **SH17**), is dedicated to PPE detection and safety monitoring in construction sites. The dataset uses **8,099 high-quality stock photos from actual construction sites**, with **professional-grade bounding box annotations** ideal for training a PPE compliance model. For SCM, I selected **8 critical classes** (mentioned previously), filtering the dataset to remove unused annotations. This resulted in **7,972 images** with **55,552 annotations**. Initial training used a pre-trained **YOLOv8n model**. The class distribution analysis (shown on the right) **revealed an unexpected challenge**.

dataset: <https://github.com/ahmadmughees/SH17dataset>

Class Distribution and Initial Training Results (150 epochs)

| Class Name | Samples Count | Ratio*    | Accuracy (mAP50) |
|------------|---------------|-----------|------------------|
| person     | 16,879        | 1.05 : 1  | 85.3%            |
| safetyvest | 1,570         | 11.32 : 1 | 23%              |
| head       | 14,952        | 1.19 : 1  | 88.2%            |
| hardhat    | 2,809         | 6.3 : 1   | 56.5%            |
| hand       | 17,782        | 1 : 1     | 83.7%            |
| gloves     | 3,352         | 5.3 : 1   | 50.5%            |
| face       | 11,000        | 1.61 : 1  | 91.9%            |
| facemask   | 709           | 25.08 : 1 | 60.9%            |

\*The ratio is each class compared to hand (17,782 samples, which is the most common)

## Problem: Severe Class Imbalance

The PPE classes have severely **imbalanced distribution** ratios (best case only 5:1), revealing a critical problem: accuracy for **PPE detection (23-61% mAP)** is **unacceptable for production**.





# Creating the **AI Model**

## Solving the Imbalanced Class Distribution

To balance the dataset, the **distribution ratio could not exceed 4:1**. For example, if the **hand** class has 17,782 samples, **facemask** (709 samples) must have **at least 4,445 samples**, an **additional 3,376 samples**. I had two options: **find a supplemental dataset** for the underrepresented classes, or **create one myself**. Finding a suitable dataset on *Roboflow Universe* was unrealistic, leaving **dataset creation as the only viable option**. However, manual annotation is time-consuming, annotating a single object typically took **15 seconds**, balancing the PPE classes (**±9,000 samples**) would take at least **37.5 hours** of constant annotating. If only I could **automate the annotation process**.

## Proposed Solution: **AI Model Zoo**

Training specialized single-class models to **bypass class imbalance constraints**, then use these models to **generate additional training data automatically**, for both the single models and the **main model**.

Here's what I should do:

### 1 Separate dataset

Split the 8-class dataset into 8 separate **single-class datasets** (e.g., **person**-only, **hardhat**-only, etc).

### 2 Train each model

Train a **specialized YOLOv8 model** out of each dataset that will be used for auto annotation.

### 3 Add supplements

Run models on the new supplement images to **automatically generate annotations** for classes that are underrepresented.

### 4 Review & iterate

Review the annotation manually, and then retrain the model. **Repeat the process until performance target are met**.



### Initial Dataset Separation and Training

Before training each specialized model, I needed to **duplicate the dataset 8 times, remove unused annotations, and removed all images with empty annotations**, which resulted in the same amount of samples from the first full-model training. Finally, by training using the same **YOLOv8n** pre-trained model, here are the result of each specialized class:

| Class Name | Samples / Images | Accuracy (mAP50) |
|------------|------------------|------------------|
| person     | 16,879 / 7,615   | 84.8%            |
| safetyvest | 1,570 / 228      | 39.5%            |
| head       | 14,952 / 6,421   | 86.9%            |
| hardhat    | 2,809 / 453      | 72.4%            |
| hand       | 17,782 / 6,353   | 81.8%            |
| gloves     | 3,352 / 1,290    | 39.8%            |
| face       | 11,000 / 5,645   | 92.1%            |
| facemask   | 709 / 392        | 43.1%            |

### Creating the Annotator

```
{
  // Input/Output
  "input": "./unlabeled_images",
  "output": "./annotations",

  // Default confidence threshold
  "defaults": { "conf": 0.5 },

  // Per-model configuration
  "models": {
    "hardhat": { // Key as class name
      "path": "./models/hardhat.pt",
      "conf": 0.8, // Base confidence

      // Size-aware thresholds (Optional):
      // [max_width_ratio, required_confidence]
      "size_rules": [
        [0.02, 0.4], // <2% frame width → 0.4 conf
        [0.05, 0.65] // <5% frame width → 0.65 conf
      ]
    },
    // ... load more models
  }
}
```

Annotator Configuration

The Model Zoo approach requires experimenting with different confidence thresholds and model combinations across multiple iterations. To allow the Annotator to achieve this flexibility without code modifications, I implemented a **JSON**-based configuration system that **separates detection parameters from implementation logic**. This makes **annotating both single and multiple class datasets straightforward**:



#### Load configuration

Set the training configuration (models, sizing rules, etc).



#### Process images

Run all the selected models on each input images.



#### Combine and format

Merge the detections and convert to YOLO format.



#### Export dataset

Generate previews and save the annotations in a **.txt** file.

<https://github.com/SwiftDSG/gidence-scm/tree/main/annotator>

### Review and Retraining

After creating the Annotator, I decided to test it only on the **hardhat** class first. I gathered an additional **600 images** of people wearing a hardhats from various angles. By using the specialized **hardhat** model (72.4% accuracy), the Annotator was able to detect **544 out of 600 images** with pretty a good bounding box.



Preview of the Annotator Output

The next thing I did was **review all 544 images manually (2,700 samples)** which took me around **2 hours (less than a second per bounding box!)**, which is a significant time improvement instead of having to spend 15 seconds per bounding box (**95% more efficient!**). After that, I train the specialized **hardhat** model with the supplemental data and **the results were promising**:

| Class Name | Samples / Images | New Accuracy   |
|------------|------------------|----------------|
| hardhat    | 5,509 / 997      | 90.3% (+17.9%) |

**This proved that the data flywheel works.** All I needed to do was to do this for the 3 other PPE classes and **repeat the process until balanced**.





Supplementing the Underrepresented Dataset

Knowing that the Model Zoo approach works on the **hardhat** class, I'll repeat the process for the other 3 classes. But, there's a caveat, the accuracy of the other PPE classes (**39-43% accuracy**) wasn't as good as the **hardhat** class. Meaning, I'll supplement these underrepresented classes **100-200 images** at a time, rather than feeding them 600 images immediately, so that I can see if there's any improvement.

| Class Name | Samples / Images | New Accuracy   |
|------------|------------------|----------------|
| safetyvest | 1,863 / 590      | 78.4% (+38.9%) |
| gloves     | 4,203 / 1,353    | 62.6% (+22.8%) |
| facemask   | 2,357 / 493      | 55.7% (+12.6%) |

First iteration

Improving the three classes took me around **12 hours**, since most of the initial accuracy aren't that high, I had to use a higher confidence threshold for the Annotator and **adjust the generated bounding box to make it consistent as the rest of the dataset**.

Same as the **hardhat** class, the Model Zoo approach shows a significant improvement, where the accuracy of the **safetyvest** class **improved by 38.9%**, followed by **hardhat**, and **facemask**. For the next iteration, I will add an additional  $\pm 500$  images to each dataset to improve its performance even further. The annotation process should not take that long since the accuracy of each model improved quite significantly.

| Class Name | Samples / Images | New Accuracy   |
|------------|------------------|----------------|
| safetyvest | 4,400 / 1,078    | 87% (+8.6%)    |
| gloves     | 5,990 / 1,499    | 74.3% (+11.7%) |
| facemask   | 4,347 / 1,035    | 71.3% (+15.6%) |

Second iteration

The second iteration took **7 hours**, since both the **gloves** and **facemask** class are way too small to review without zooming in. But this approach is still much better compared to manually annotating  $\pm 1,500$  images that could take 30 hours (**still 77% faster**).

Now that the PPE classes already have a good performance (all classes have an accuracy greater than 70%) and a good distribution ratio, I can combine the specialized datasets with the base SH17 dataset. Or so I thought.

The Forgotten Rule

Not annotating all objects of interest within an object detection is worse than having no image at all. Where it could lead the model to learn that the unannotated objects are part of the background (**False Positive Misclassification**).

Merging the Supplemented Dataset — 1 / 2

The unforeseen problem by creating a specialized dataset is merging them back, **it's not as easy as appending a new set of images**. The **new dataset might contain a lot of objects of interest that are unannotated**. For example: within the **hardhat** dataset, there are  $\pm 1,000$  images of people wearing a hardhat, where within the main dataset, every person is an object of interest and should be annotated.

I initially thought that using the 8 specialized models would solve this problem easily. What I didn't realize is: **when adding supplemental images for underrepresented classes, I'm also increasing the required sample count for already-represented classes to maintain the 4:1 balance**. Which means, that the number of needed samples also increases for each under-represented class. To assess the impact on class distribution, I used the Annotator with **all 8 specialized models to annotate a new supplemental dataset** ( $\pm 1,500$  new images, to prevent duplicates). This revealed the sample counts across all classes.

| Class Name | Samples / Images |
|------------|------------------|
| person     | 3,859 / 1,485    |
| safetyvest | 3,111 / 957      |
| head       | 3,458 / 1,476    |
| hardhat    | 1,887 / 785      |
| hand       | 4,771 / 1,350    |
| gloves     | 1,674 / 480      |
| face       | 1,585 / 920      |
| facemask   | 3,753 / 1,033    |





■ Merging the Supplemented Dataset — 2 / 2

After the new supplement dataset has been annotated with all the 8 specialized models. I added them to the main dataset so that I could see whether the distribution ratio had met our target of 4:1 or not.

| Class Name | Samples / Images | Ratio   |
|------------|------------------|---------|
| person     | 20,738 / 9,100   | 1.1 : 1 |
| safetyvest | 4,681 / 1,185    | 4.8 : 1 |
| head       | 18,410 / 7,897   | 1.2 : 1 |
| hardhat    | 4,496 / 1,234    | 5.1 : 1 |
| hand       | 22,553 / 7,703   | 1 : 1   |
| gloves     | 5,026 / 1,770    | 4.4 : 1 |
| face       | 12,585 / 6,565   | 1.8 : 1 |
| facemask   | 4,462 / 1,425    | 5.1 : 1 |

Based on the table above, the distribution ratio indeed improved by a lot, but still hasn't met our target of 4:1. But, I believe that in the future, whenever I add a new supplement, the distribution ratio could be very different, either it could meet the target or the ratio could be completely off. To address this problem, whether with the current dataset or whenever I'm going to add another supplement dataset, I have to **create a balancer** that could **balance out the distribution ratio automatically** by deciding which images to keep.

```
def value_image(image, current_counts, target_ratio=4.0):
    # Load this image's annotations
    annotations = image.load_annotations()

    # Count classes in this image
    image_counts = {}
    for ann in annotations:
        class_name = ann['class']
        image_counts[class_name] = image_counts.get(class_name, 0) + 1

    # Simulate removing this image from the dataset
    simulated_counts = current_counts.copy()
    for class_name, count in image_counts.items():
        simulated_counts[class_name] -= count

    # Calculate balance metrics
    current_min = min(current_counts.values())
    current_max = max(current_counts.values())
    current_ratio = current_max / current_min

    simulated_min = min(simulated_counts.values())
    simulated_max = max(simulated_counts.values())
    simulated_ratio = simulated_max / simulated_min

    # Score based on whether removal improves balance
    if simulated_ratio < current_ratio: # Removing this image IMPROVES balance
        improvement = current_ratio - simulated_ratio
        value = -improvement * 100 # Negative = keep this image
    else: # Removing this image WORSENS balance
        degradation = simulated_ratio - current_ratio
        value = degradation * 100 # Positive = remove this image

    # Additional penalty for images with only overrepresented classes
    majority_threshold = current_max * 0.8
    has_majority_only = all(
        current_counts[cls] > majority_threshold
        for cls in image_counts.keys()
    )

    if has_majority_only:
        value += 50 # More likely to remove

    # Additional bonus for images with underrepresented classes
    minority_threshold = current_min * 1.2
    has_minority = any(
        current_counts[cls] < minority_threshold
        for cls in image_counts.keys()
    )

    if has_minority:
        value -= 50 # More likely to keep

    return value
```

To achieve this, I'm going to use a **greedy optimization algorithm**, which means that each image will be valued by its contribution towards achieving balance. **If removing the image would make the dataset more balanced, then the image will have a higher value, and vice versa. Images with larger values will be removed, meanwhile, the ones with smaller values will be kept.** The process is **repeated until the desired distribution ratio is met.** The snippet on the left shows how the logic behind giving a value to each image.

By using this approach, **I was able to achieve a dataset with a distribution ratio of 4:1.** The number of images initially was 9,487 images, and the balancer removed around 900 images to achieve a balanced distribution, **resulting in a new dataset with 8,531 images.** All that's left to do is to train the new dataset and see the results.

| Class Name | Samples / Images | Ratio (hand vs) | Accuracy (mAP50) |
|------------|------------------|-----------------|------------------|
| person     | 17,741 / 8,152   | 1.0 : 1         | 81.1%            |
| safetyvest | 4,680 / 1,184    | 3.8 : 1         | 85.3%            |
| head       | 16,865 / 7,629   | 1.1 : 1         | 82.6%            |
| hardhat    | 4,496 / 1,234    | 3.9 : 1         | 88.9%            |
| hand       | 17,951 / 7,288   | 1 : 1           | 79.5%            |
| gloves     | 5,020 / 1,768    | 3.5 : 1         | 76.1%            |
| face       | 12,312 / 6,490   | 1.5 : 1         | 90.9%            |
| facemask   | 4,460 / 1,424    | 4 : 1           | 70.7%            |

Now that **the AI model is production-ready**, I can move on to the **Main Runtime**.





# The Main Runtime

## Overview of the Whole Operation

This part of the processor layer is responsible for **making the Inference Engine able to run reliably, managing the information generated by the Inference Engine, and sending those information to the Server Layer.**

To be able to manage those responsibilities, the Main Runtime has **5 workers that work in parallel to make sure that everything stays performant** without needing to take turns between each task.

This design choice makes **the processor layer able to run independently** without depending on the availability of a server. **Making it industry-ready.**

### 1 Inference Engine Watchdog

This worker's job is to **run the Inference Engine based on the used configuration**, such as the number of cameras and which AI model to use. This worker also **automatically restarts it if it ever crashed.**

### 2 UDS Reader

**Reads all the evidences sent by the Inference Engine** via UDS, and queues them for the evidence manager to process.

### 3 Evidence Manager

**Filters the evidences** to make sure that **every violation occurrences are unique**, and saves them locally before sending.

### 4 Information Sender

An optional worker to **send the filtered violations to the assigned Webhooks.** Ideal for custom integration on a custom solution.

### 5 Request Listener

**Enables the processor layer to operate independently from the server**, making it a flexible edge device that can be reconfigured remotely.





### ■ Inference Engine Watchdog

```
let _ = tokio::spawn(async move {
    let delay = Duration::from_secs(5);

    loop {
        // Spawn the Inference Engine process
        let mut child = match Command::new("bash")
            .arg("-c")
            .arg("source setup.sh && python3 -m inference.main")
            .spawn()
        {
            Ok(child) => child, // Inference Engine started
            _ => {
                // Failed to run the Inference Engine, restart after a delay
                sleep(delay).await;
                continue;
            }
        };

        // Wait for the process to exit
        if let Ok(status) = child.wait() {
            if status.success() { // Clean exit, do not restart
                break;
            }
        }

        // The process crashed, wait before restarting to prevent rapid restart loops
        sleep(delay).await;
    }
});
```

Logic of the Inference Engine Watchdog

Within the snippet above, the Main Runtime spawns a new child process, which is the **Inference Engine process**, by running a **bash** command followed by a command to **activate a virtual environment** and a **run a Python program**. If the child process **fails to start or crashes midway**, it will **automatically restart the whole process**.

### ■ UDS Reader

To receive violations from the Inference Engine via UDS, I created a **.sock file** which will be the place where both the Inference Engine and the Main Runtime could communicate with each other. Then, **create a stream that fills a buffer whenever a changes occurs within the .sock file**. Every time the **buffer** is filled, try parsing it to check whether the **buffer** is a correct evidence or not. If the **buffer** indeed contains a correct evidence, then **enqueue the evidence for the Evidence Manager to handle**.

```
let _ = tokio::spawn(async move {
    let _ = fs::remove_file("/tmp/scm_uds.sock").await;
    let listener = UnixListener::bind("/tmp/scm_uds.sock").unwrap();

    loop {
        let (mut stream, _) = if let Ok(conn) = listener.accept().await {
            conn
        } else {
            // Fails to start the listener
            continue;
        };
        let mut buffer = vec![0; 65536];

        if let Err(e) = stream.read(&mut buffer).await {
            println!("Error reading from stream: {}", e);
            continue;
        }

        // Get actual buffer length
        let len = buffer.iter().position(|&x| x == 0).unwrap_or(buffer.len());

        // Parse buffer into Evidence struct
        let mut evidence = if let Ok(v) = from_slice::<Evidence>(buffer[0..len].as_ref()) { v } else {
            continue; // Parsing failed
        };

        // Enqueues the evidence for the Evidence Manager
        let mut queue = queues.write().await;
        queue.push_back(evidence.clone());
    }
});
```

A Snippet of the UDS Reader

### ■ Evidence Manager

```
let _ = tokio::spawn(async move {
    loop {
        let evidence = {
            let mut queue = queues.write().await;
            match queue.pop_front() {
                Some(e) => e,
                None => continue // Nothing in the queue
            }
        };

        // Check if there's a new violation committed
        let mut new_violation = false;
        for person in evidence.person.iter() {
            // Check if a violation by this person on this camera already exists
            let key = format!("{}", evidence.camera_id, person.id);
            if let Some(existing) = {
                let violation = violations.read().await;
                violation.get(&key).cloned()
            } {
                // Check if the old evidence is older than 10 minutes than the new one
                if evidence.timestamp - existing.timestamp < 10 * 60 * 1000 {
                    continue; // Skip inserting this evidence
                }
            }

            // Insert or update the violation record
            new_violation = true;
            let mut violation = violations.write().await;
            violation.insert(key, evidence.clone());
        }

        if new_violation {
            let image = match fs::read(format!("/tmp/{}.jpg", evidence.camera_id)).await {
                Ok(d) => d,
                Err(_) => continue,
            };

            // Save the evidence and the image to ./evidence
            fs::write(&format!("./evidence/{}.jpg", evidence.id), &image).await.unwrap();
            fs::write(&format!("./evidence/{}.json", evidence.id), &serde_json::to_string(&evidence).unwrap()).await.unwrap();

            sleep(Duration::from_millis(100)).await;
        }
    }
});
```

The Evidence Manager's Thread

This worker's job is to **prevent saving duplicate violations** by making sure that **each violation is tracked by keeping tab on the person's ID and the timestamp**. If the **violation is by a new person**, or if **the same person's previous violation is older than 10 minutes**, only then the evidence is saved.





## Information Sender

```
let _ = tokio::spawn(async move {
  loop {
    // Load all evidences with a format: ./evidence/{}.json
    // (NOT ./evidence/uploaded.{}.json) that have not been sent yet
    let mut evidences = Vec::new();
    let mut evidences = Vec::new();
    let entries = match fs::read_dir("./evidence").await {
      Ok(entries) => entries,
      Err(_) => {
        sleep(Duration::from_millis(100)).await;
        continue;
      }
    };

    let mut dir = entries;
    while let Ok(Some(entry)) = dir.next_entry().await {
      let file_name = entry.file_name().into_string().unwrap();
      if file_name.ends_with(".json") && !file_name.starts_with("uploaded.") {
        let data = match fs::read_to_string(entry.path()).await {
          Ok(d) => d,
          Err(_) => continue,
        };
        let evidence: Evidence = match serde_json::from_str(&data) {
          Ok(e) => e,
          Err(_) => continue,
        };
        evidences.push(evidence);
      }
    }

    if evidences.is_empty() {
      sleep(Duration::from_millis(500)).await;
      continue; // No new evidences to process
    }

    for evidence in evidences.drain(..) {
      // Send evidence and captured image to the webhook
      let image = match fs::read(format!("./evidence/{}.jpg", evidence.id)).await {
        Ok(d) => d,
        Err(_) => continue,
      };

      let mut webhook = {
        let device = devices.read().await;
        device.processor.webhook.clone()
      };

      let mut success = true;
      for wh in webhook.drain(..) {
        let payload = serde_json::to_string(&evidence).unwrap();
        if !wh.send(payload, image.clone()).await {
          success = false;
        }
      }

      if success {
        // Rename the evidence files to mark them as uploaded
        let _ = fs::rename(
          format!("./evidence/{}.json", evidence.id),
          format!("./evidence/uploaded.{}.json", evidence.id),
        ).await;
        let _ = fs::rename(
          format!("./evidence/{}.jpg", evidence.id),
          format!("./evidence/uploaded.{}.jpg", evidence.id),
        ).await;
      }
    }

    sleep(Duration::from_millis(100)).await;
  }
});
```

### Optional Worker

The Information Sender is an optional worker where it can be skipped entirely by having an empty list of target endpoints. This means that the **Processor Layer could work independently without the need of a server.**

The way the Information Sender works is by having a list of target endpoints for it to be able to send the saved evidence to. First, it **scans all the saved evidence within the ./evidence directory**, then it **filters out all the uploaded evidence** to prevent it from uploading duplicate data, which **can be differentiated by the file name**. Evidence that hasn't been uploaded yet has a file name with the following format: **[EVIDENCE\_ID].json** for the details of the occurred event and **[EVIDENCE\_ID].jpg** for the captured image of the occurred event. Meanwhile, the evidence that has been uploaded have a **uploaded.** prefix in the file name (for example: **uploaded.1234abcd.json** and **uploaded.1234abcd.jpg**). Second, **iterate through each filtered evidence item and uploads it to all listed endpoint** via a **multipart/form-data** HTTPS Request. Lastly, if the upload succeeds, **mark the file by adding an uploaded. prefix in the file name**, that way it won't be uploaded again by the worker on the next iteration.

## Request Listener

To make the Processor Layer accessible and configurable without needing an internet connection, I need to make it easily accessible from a technical standpoint. One of the easiest approaches that's heavily adopted in the industrial-grade devices is **making endpoints that are accessible via HTTP request**. This layer needs to provide: the ability to **view and modify the processor's configuration, manage cameras**, and **access captured evidence**. Knowing that, here are the endpoints available within the processor:

- GET /config**  
Returns the processor's configuration (name, model to use, list of Webhooks, etc.).
- POST /config**  
Update the processor's configuration, it will also restarts the Inference Engine if succeeds.
- GET /cameras**  
Loads the configuration for the cameras, which is the RTSP address of the camera.
- POST /cameras**  
Update the configuration for the cameras, it will also restarts the Inference Engine if succeeds.
- GET /evidences**  
Loads a list of locally-stored evidences.
- GET /ping**  
Pings the processor to check if it's running.

**On the next iteration, I will introduce a web interface** so that the user can tinker with the processor locally without needing an internet connection.



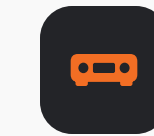


# Server Layer

Bridging the Processor and the User

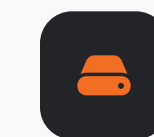
Starting from this layer and afterwards are optional, but for the sake of creating a whole system, I'm making an example of how to **leverage the processor layer's capabilities of sending violations to a Webhook, so that they can be stored in the database** and to help the end-user (HSE supervisors), I'm also going to **integrate Apple Push Notification service (APNs)** to notify them whenever a PPE usage violation occur.

For this use-case, the server layer is the one responsible for **bridging information between the processor layer and the user layer via HTTPS**. The endpoints have basic purposes such as viewing, creating or updating data, so documentation for them is unnecessary.



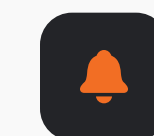
## Runtime

The system's runtime will be written in **Rust**, where its nature of being **memory-safe and high performance would benefit this use-case by a lot**. The runtime is the one responsible for **connecting multiple technologies to work together seamlessly**.



## Data Storage

This is where the **whole ecosystem's configuration and captured data are stored**, such as processors, cameras, users, cluster, etc. I will use **MongoDB for its ability of storing data in a JSON-like format**. Knowing the processor layer's evidence output is in JSON.



## Notification Sender

For this use-case, I'm going to integrate **Apple Push Notification service (APNs)**, since I'm only focusing for Apple devices for now. This way, **HSE supervisors could subscribe if they want a notification to appear whenever a violation occur**.





### Creating a Webhook for the Processor

To create an endpoint that could accept HTTP request from the processor, I will use Rust's **actix** crate to create an endpoint that accepts **multipart/form-data** encoding, this is needed because the processor sends a mixed payload of an image of the occurred event and the text which contains the detailed information of the occurred event formatted in **JSON**.

Based on the snippet on the right, I created an endpoint that receives a **POST** request on **/{{processor\_id}}**. The first thing the handler does is to **check if the processor is indeed valid or registered** by finding it on the database. And then, it **collects the multipart/form-data fields to make sure that both the image and data fields are present and contain the expected data**. Lastly, if everything is according to plan, it **saves the evidence to the database**, returns **201 created** response back to the processor, and queues the evidence for a notification distribution to the end users..

<https://github.com/SwiftDSG/gidence-scm/tree/main/server>

```
#post("/{processor_id}")
pub async fn create_evidence(
    processor_id: web::Path<String>,
    queue: web::Data<Arc<Mutex<VecDeque<Evidence>>>>,
    mut pool: Mutipart,
    db: web::Data<Database>,
) -> HttpResponse {
    let processor_id = processor_id.into_inner();

    // Verify processor exists
    let processor = match Processor::find_by_id(&processor_id, db.get_ref()).await {
        Ok(v) => v,
        Err(e) => return HttpResponse::NotFound().body("PROCESSOR_NOT_FOUND"),
    };

    // Collect multipart fields
    let mut field_data: (Option<Vecu8>, Option<EvidenceRequest>) = (None, None);
    while let Some(chunk) = pool.payload.next().await {
        let field_name = match chunk.content_disposition() {
            Some(cd) => cd.get_name().unwrap_or("").to_string(),
            None => continue,
        };

        if field_name == "image" {
            let mut data = Vec::new();
            while let Some(chunk) = field.next().await {
                match chunk {
                    Ok(bytes) => data.extend_from_slice(&bytes),
                    Err(e) => break,
                }
            }
            if !data.is_empty() {
                field_data.0 = Some(data);
            }
        } else if field_name == "data" {
            let mut json_string = String::new();
            while let Some(chunk) = field.next().await {
                match chunk {
                    Ok(bytes) => json_string.push_str(&String::from_utf8_lossy(&bytes)),
                    Err(e) => break,
                }
            }
            field_data.1 = Some(json_string);
        }
    }

    // Verify both fields are present
    let (image_data, evidence_data) = match field_data {
        (Some(img), Some(data)) => (img, data),
        _ => return HttpResponse::BadRequest().body("MISSING_DATA"),
    };

    // Generate evidence ID
    let evidence_id = uuid::new_v4().to_string();
    let file_path = format!("{}/evidence/{}.jpg", evidence_id);

    // Save image to ./evidence/EVIDENCE_ID-.jpg
    if let Err(e) = web::block(move || {
        let mut file = File::create(&file_path.clone()).unwrap();
        file.write_all(&image_data);
    })
    .await
    .map_err(|_| ())
    .and_then(|_| r.map_err(|_| ()))
    {
        return HttpResponse::InternalServerError().body("FAILED_TO_SAVE_IMAGE");
    }

    // Create evidence record
    let evidence = Evidence {
        id: evidence_id,
        cluster_id: processor.cluster_id.clone(),
        processor_id: processor_id,
        camera_id: evidence_data.camera_id,
        frame_id: evidence_data.frame_id,
        timestamp: evidence_data.timestamp,
        person: evidence_data.person,
    };

    // Save to database
    match evidence.save(db.get_ref()).await {
        Ok(_) => {
            let mut queue = queue.write().await;
            queue.push_back(evidence);
            HttpResponse::Created().finish() // Save succeeds, TODO: Implement APNs here
        }
        Err(e) => {
            // Delete the image if database save fails
            let _ = fs::remove_file(&file_path);
            error_handler(e);
        }
    }
}
```

### Sending Real-Time Notifications

Each time an evidence with violations is received by the Webhook, it gets queued for the Notification Sender to process. And then, once the evidence's turn has come, find all users associated with the evidence's place of event or cluster. Next, find the devices that's subscribed for push notification for each user. Lastly, send the notification to the user's devices.

For this iteration, I'm only including **Apple Push Notification service** integration. But, based on the **SubscriberKind** enum on the snippet, I can easily add another notification service when I need to scale the system to support more kinds of notifications. For example: Android, Emails, Telegram, and many more.

Now that the main features of the server layer has been implemented, I'm going to move on to the user layer, where the end-user can directly interact with the system that I've been documenting so far.

```
let _ = tokio::spawn(async move {
    // Initialize APNS client
    let file = File::open("keys/apns.p8").expect("APNS_NOT_FOUND");
    let apns = Client::token(
        &mut file, "APNS_KEY", "APNS_TEAM", ClientConfig::new(a2::Endpoint::Sandbox)
    ).expect("TOKEN_CREATION_FAILED");

    loop {
        // Wait for new evidence in the queue
        let evidence = {
            let mut queue = queues.write().await;
            match queue.pop_front() {
                Some(e) => e,
                None => continue
            }
        };

        // Find users associated with the evidence's cluster
        let mut users = match User::find_many_by_cluster_id(&evidence.cluster_id, &database).await {
            Ok(v) => v,
            _ => continue,
        };

        for user in users.drain(..) {
            // Find user's devices subscribed to notifications
            let mut subscribers = match Subscriber::find_many_by_user_id(&user.id, &database.clone).await {
                Ok(v) => v,
                _ => continue,
            };

            for subscriber in subscribers.drain(..) {
                // Send notification based on subscriber kind
                match &subscriber.kind {
                    SubscriberKind::Apple(token) => {
                        let builder = DefaultNotificationBuilder::new()
                            .set_title("Terjadi Pelanggaran Baru!")
                            .set_subtitle("Cek detailnya di aplikasi SCM.")
                            .set_sound("ping.flac");

                        let payload = builder.build(token, NotificationOptions {
                            apns_topic: Some("com.gidence.scm"),
                            ..Default::default()
                        });

                        if let Err(err) = apns.send(payload).await {
                            let response = match err {
                                a2::Error::ResponseError(v) => v,
                                _ => continue,
                            };

                            // Remove subscriber if token is invalid or unregistered
                            if response.code == 403 || response.code == 410 {
                                let _ = subscriber.delete(&database.clone).await;
                            }
                        }
                    }
                }
            }
        }
    }
});
```





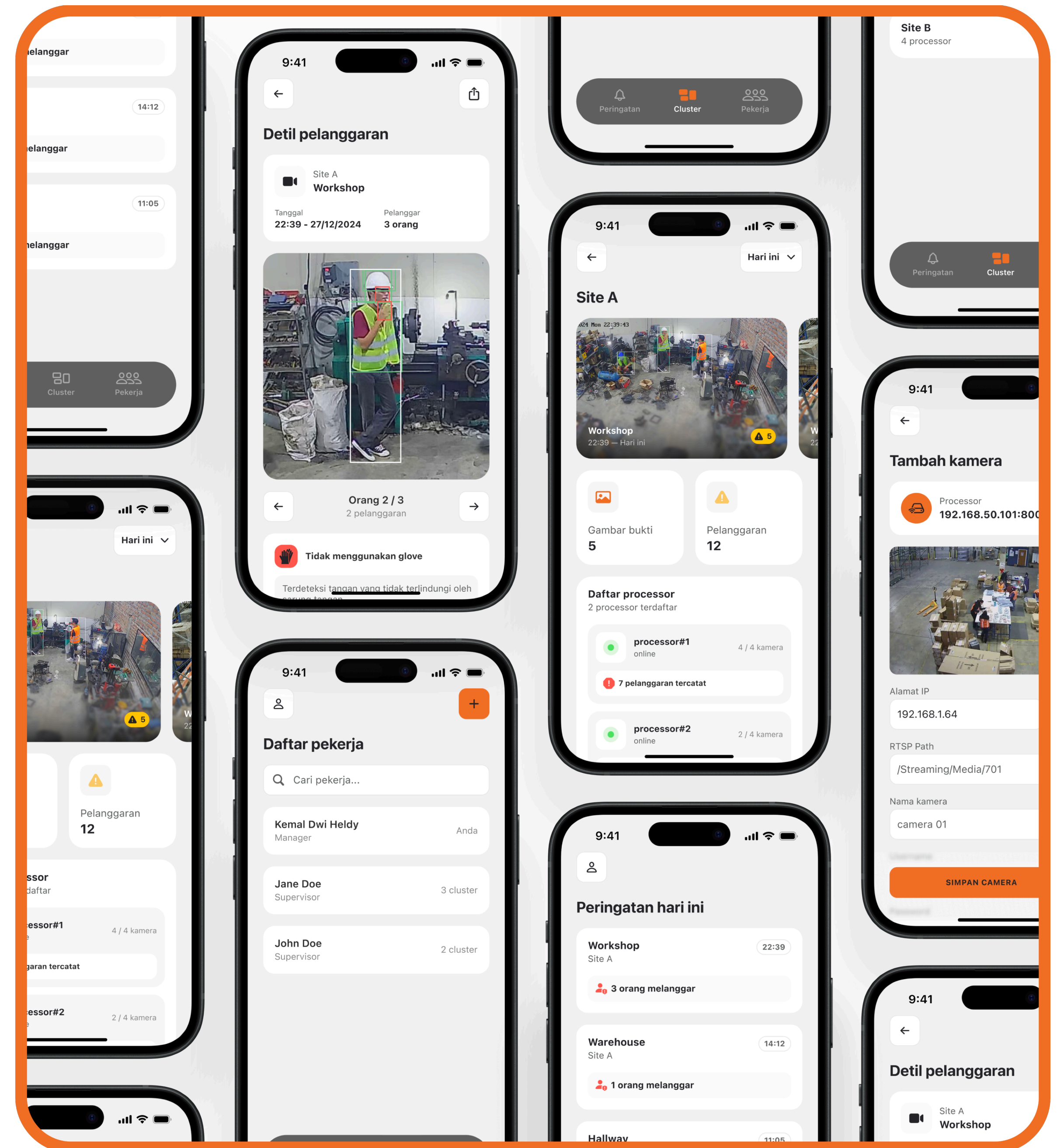
### User Layer

Making the System Accessible for the User

To allow the users to access the system, I'm going to make **an iOS application that lets HSE supervisors to be alerted whenever a PPE usage violation occurs**. Not only that, the application should also allow them to **view violation details**, and **see the overview of clusters that they're assigned to**.

Now, to accommodate the needs of managers to tinker with the system's configuration, this application allows them to **manage users, clusters, and cameras within each cluster**.

Since the front-end of the system isn't my main learning journey, **I won't be covering the technical details**.







# User Layer

## Seeing the Cluster Details

Within this page, the user can see bunch of information that's been aggregated so that it's easier to understand:

### 1 Time-based filter

Filter evidence based on the selected time range.

### 2 List of captured evidence

Every event that contains a unique violation will be available here.

### 3 Cluster's overview

A small summary of filtered events happened within the cluster.

### 4 List of processors

List of processors assigned within a the selected cluster. Includes a small summary of captured events within the time range.







# User Layer

## Accessing the Details of Violation

Enabling the user to see the details and the evidence of the occurred violation.

### 1 Event details

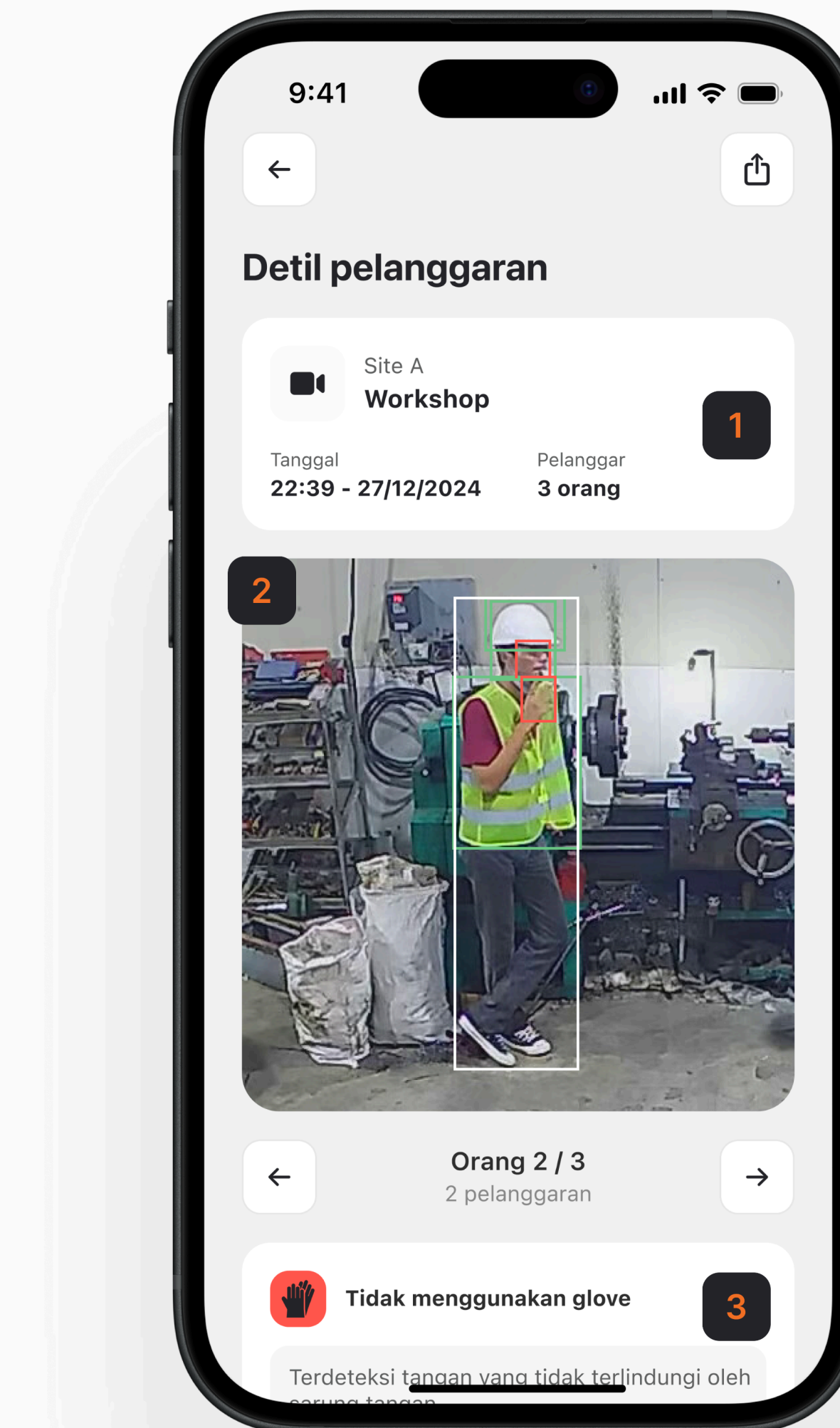
The details of the occurred event, such as camera name, site name, date, and time.

### 2 Close-up look of each person

A cropped and zoomed screen capture of each person and the properties assigned to them, such as PPE items and body parts.

### 3 Person's properties

Detailed information about each person's compliance towards PPE usage. Each violation is provided with an explanation message as well, so that the readers knew the reason of the evaluation.







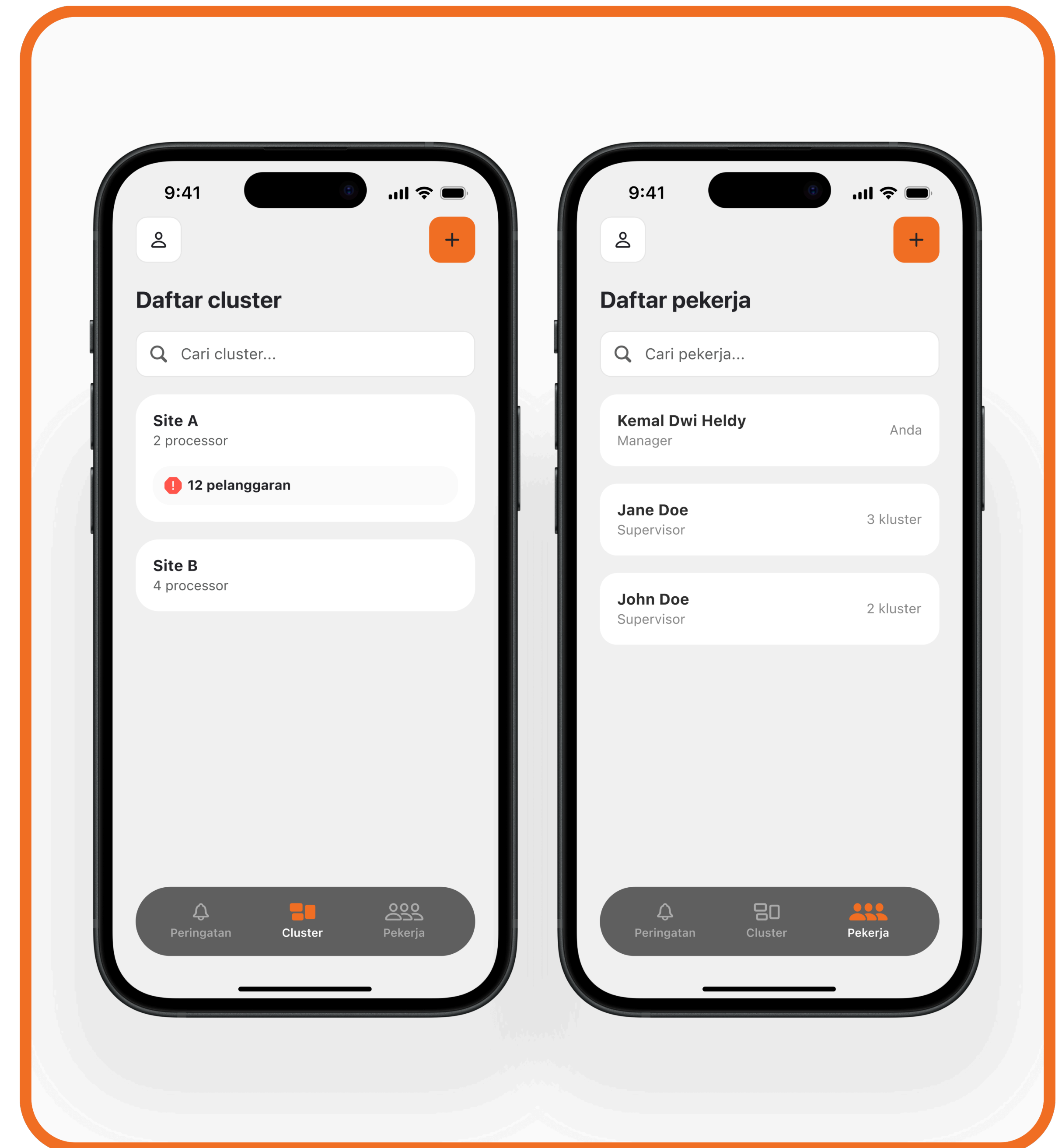
## User Layer

### Managing Users and Clusters

To fulfill the manager's needs of **seeing and changing both the clusters and users** within the application, I created these pages:

The user page, where the manager could **manage the users of the application by assigning a role of supervisor or manager to each user**. Not only that, if the manager decides to assign a user as supervisor, they could also **assign which cluster should be associated to the supervisor**. That way, the user only receive notification from the associated cluster.

Within the cluster page, the manager could **add or remove both a cluster and the processors within**.







# Deployment and Testing

## Simulating the Industrial Environment

To validate the complete system in real-world conditions, I **deployed all three layers to their target hardware and conducted component-level testing to evaluate each component's performance**. For this use case, the processor was deployed on a **Raspberry Pi + Hailo-8L** combo. The server was hosted on a **local computer** that's **connected via VPN** to enable notification delivery and data synchronization across different networks. The iOS application was **deployed on an iPhone 15**. This configuration **simulates a realistic industrial deployment** where the **processor operates on-site**, the **server is hosted remotely**, and the supervisors **access the system from mobile devices**.

## Validating Every Layer

Since end-to-end testing requires extended deployment time, I conducted **component-level tests** to verify each layer's **functionality**.

### Processor Layer

**Tested with pre-recorded CCTV feeds**, the processor is able to detect PPE when workers were clearly visible in frame. **Validated violation processing, evidence capture, and queue management**.

### Server Layer

Violations captured by the processor are successfully **sent to the server, stored in MongoDB, and notifications distributed via APNs**. **This verifies webhook endpoint functionality and cross-network VPN communication**.

### User Layer

Validated that the iOS application **receives push notifications from APNs, displays violation details with evidence images, and allows supervisors to view cluster information**.

In future iteration, I will be conducting an extended deployment testing with quantitative metrics and comprehensive end-to-end validation in production environment.





# Wrapping it Up

## Summary of the Journey

I worked on this system from the middle of December 2025 until the 29th of January 2026. **One and a half months to completely rework the whole system from the ground up**, up to the point where I could document the journey in this portfolio.

Of course, the system is still far from perfect. But, thanks to the journey of working on SCM, **I could share with you a key learning that I'm really proud of**. I hope that you could learn a thing or two from this.

I will also share about the next step to **improve the system's functionality and reliability** even more. Finally, I would like to thank you for reading my portfolio.

## Key Learning

When I trained the model initially, the accuracy was not suitable for production at all, where PPE classes received an accuracy of 23-43%. Upon further investigation, this was caused by an imbalanced class distribution ratio, where the **facemask has a distribution ratio of 25:1**. By balancing the distribution ratio, the PPE classes' accuracy increased significantly, where the PPE classes have an accuracy of 70-88%. By solving this problem, I learned that **data quality matters more than model complexity**.

## Achievements

Other than having a working system, upon the journey, I built tools that will **save hours of manual labor and headaches**. I created an **auto-annotator** that can be **scalable with many other models**. This tool can be used to annotate any object detection dataset. I also created a tool to **solve imbalanced class distribution** that picks which image to keep and which image to remove from the dataset automatically by using **greedy optimization algorithm**.

## Future Improvements

Due to the time constraints, there are many obvious missing key activities that I should've done to improve and measure the systems functionality, including: **blackbox testing with measurable outcomes**, **adding more PPE models** to allow applications in a broader use-cases, **using a more sophisticated devices** like Nvidia Jetson, etc.





# Every Worker Deserves a Tomorrow



+62 813 5776 1518



kdwiheldy@gmail.com



kdwiheldy.com

