Augmenting Low-Cost Robotic Arms with YOLO-Based Perception and Vision-Language-Action Policies

Xingjian Jiang[a,1] **and Pree Simphliphan**[a,2]

[a]*Boston University, College of Engineering*

Professor Eshed Ohn-Bar

Abstract—Robotic manipulation in unstructured environments often demands complex perception and decision-making capabilities, which typically require expensive sensors and powerful computing platforms. In this project, we aim to bridge this gap by integrating real-time vision detection and low-cost robotic manipulation using an affordable, modular hardware setup. Our system leverages YOLOv8 for real-time object detection and employs LeRobot so100 arms as actuators to execute dynamic grasping and placement tasks. Vision inputs are gathered from a combination of a XIAO ESP32-S3 Sense module, a standard laptop webcam, and an arm-mounted UVC camera, ensuring diverse perspectives. We further enhance the policy generalization capabilities by training a Vision-Language-Action (VLA) model that incorporates structured object detection outputs. Additionally, custom control strategies were developed to dynamically adjust robot behavior based on the real-time YOLO detections. Our results demonstrate that with careful system design, it is feasible to achieve robust, adaptive robotic manipulation on a low-cost platform, paving the way for broader accessibility of intelligent robotic systems.

Keywords—*LeRobot, YOLOv8, ESP32-S3 Sense, Robotic Manipulation, Vision-Language-Action Model*

1. Introduction

1.1. Motivation

Robotic manipulation—the ability for a robot to interact physically with its environment—has become an essential pillar for real-world autonomous systems. However, enabling robust manipulation remains challenging, particularly in unstructured, dynamic environments. State-of-the-art robotic platforms often leverage expensive sensor suites (e.g., depth cameras, LiDAR) and heavy computation backends (e.g., server-grade GPUs) to perceive and act intelligently. These systems, although powerful, impose prohibitive cost and deployment barriers for applications in education, home robotics, and small-scale industries.

Motivated by the desire to democratize access to capable robotics, we aim to investigate whether low-cost hardware components and lightweight perception models can together enable practical robotic manipulation without sacrificing performance. Our goal is to bridge the gap between research-grade systems and affordable, deployable robotic solutions.

1.2. Related Work

Recent research efforts have explored vision-based robotic manipulation using end-to-end deep learning. Projects like Diffusion Policy **diffusion-policy** and Action Chunking Transformer (ACT) **act-transformer** demonstrate that supervised behavior cloning from human teleoperation data can yield strong manipulation policies.

Our work extends this trend by showing that even modest vision setups—such as ESP32-S3 Sense streaming JPEG frames—can support competitive policy learning, provided that synchronization, augmentation, and system-level optimization are carefully designed.

1.3. YOLOv8 for Vision-Driven Robotic Manipulation

YOLOv8 **yolov8-docs** represents the latest evolution of the YOLO family of real-time object detectors. It achieves high detection accuracy while maintaining real-time throughput even on mid-range hardware, making it an ideal choice for deployment in resource-constrained robotic systems. In this project, we utilize YOLOv8 to provide structured semantic information about the environment, feeding into both teleoperation visualization and autonomous policy learning.

1.4. Vision-Language-Action (VLA) Models

Beyond vision-based control, VLA models introduce an additional abstraction layer by conditioning robot behavior on textual or symbolic task prompts. By fusing visual input with high-level language instructions, VLA models can generalize more robustly across task variations, object types, and layouts.

Our system integrates VLA concepts by training policies that accept not only raw RGB frames but also YOLO detection outputs, enabling more flexible and scalable manipulation capabilities.

1.5. Project Scope and Goals

This project targets the development of an affordable, vision-driven robotic manipulation system. Key project goals include:

- Designing a hardware-software pipeline combining Wi-Fi streaming, lightweight detection, and modular robot control.
- Recording synchronized multi-modal datasets suitable for supervised learning of manipulation policies.
- Evaluating policy performance with and without explicit vision augmentation.
- Demonstrating successful grasping and placement tasks under real-world conditions.

Through this effort, we aim to validate the hypothesis that vision-augmented manipulation policies, even when trained on modest hardware, can achieve robust behaviors in practical environments.

2. Method

2.1. System Setup

The hardware configuration of our system was carefully selected to balance cost, flexibility, and performance, targeting a real-world task.

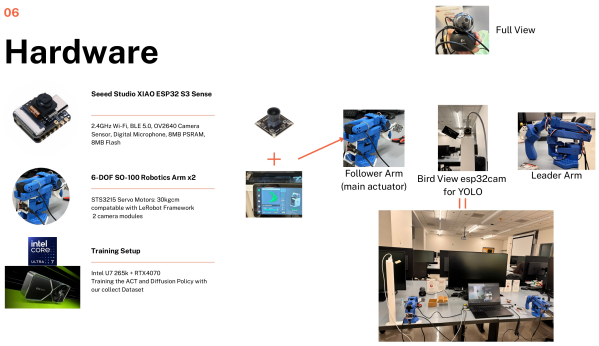

Figure 1. Overall setup.

2.1.1. Robot Arm

The main actuator is the LeRobot so100 robotic arm, composed of modular leader and follower units for teleoperation training. Each unit is equipped with Feetech STS3215 smart servo motors, supporting RS-485 differential communication with daisy-chain wiring.

The parts for both robotic arms and camera stand are available in github repo of official website. Additionally, the design for ESP32-S3 mount with camera stand need to be made in computer-aided design (CAD) using Onshape. as it is the supplement to original model. All

the physical parts are 3D printed using BambuLab X1C printer with PLA filament.

The controller can address individual motors via unique IDs, allowing scalable bus extensions. Each joint provides:

- Position feedback (degrees)
- Velocity feedback (degrees/s)
- Torque measurement (newton*meters)

These measurements enable fine-grained control and policy learning using proprioceptive features.

Justification This modular motor configuration allows fine-grained feedback control and low-latency actuation, crucial for adapting to dynamic changes in tabletop organization scenarios while maintaining system scalability and affordability.

2.1.2. Vision System

The vision subsystem includes:

- **XIAO ESP32S3 Sense Module**: Streams MJPEG video at 640x480 resolution over 2.4GHz Wi-Fi as the bird view for YOLO detection.
- **Laptop Integrated Webcam**: Provides a static global view of the workspace as the first eye for LeRobot.
- **Arm-mounted UVC Camera**: Captures gripper-aligned close-up views critical for precise manipulation as the second view for original LeRobot framework.

Justification Utilizing an ESP32-S3 module reduces the need for expensive cameras, making the system viable for low-cost deployment.

2.1.3. Software Stack

The system software pipeline consists of:

- **Robot Manipulation (LeRobot Framework)**: Motor bus management, calibration, teleoperation, dataset recording.
- **Computer Vision (YOLOv8)**: Receives MJPEG streams, decodes frames using OpenCV, runs detection with model using Ultralytics.
- **Embedded Systems (ESP32 and PC)**: ESP-IDF v5.4, esp-camera modules, WiFi Configuration, FreeRTOS for scheduling, Low Power Management,.

Justification Separating vision processing and low-level control allows flexible upgrades, e.g., swapping YOLO models or motor drivers without overhauling the complete system pipeline, enhancing maintainability.

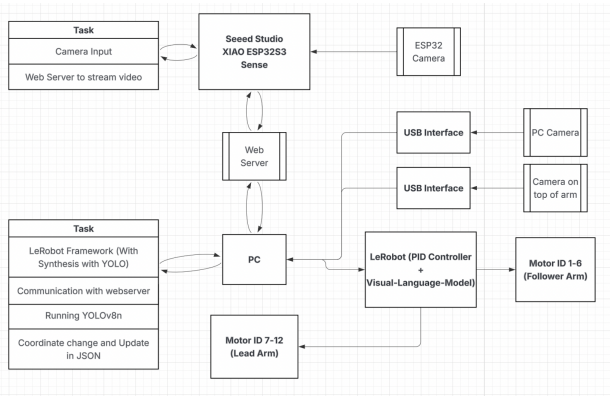

Figure 2. Our System Architecture Diagram.

2.2. Team Contributions

The project was executed collaboratively with clear division of responsibilities:

- **Xingjian Jiang**: Responsible for LeRobot arm teleoperation interface development, LeRobot configuration, synchronized dataset recording, training and tuning Diffusion, ACT, and Vision-Language-Action policies with GPU, and deploying final trained policies onto the physical arm hardware.
- **Pree Simphliphan**: Responsible for camera configuration with ESP32-S3, YOLOv8 object detection model fine-tuning, dataset labeling for custom classes, optimizing YOLOv8 for real-time MJPEG stream input, 3D printing parts and designing mount for ESP32-S3.
- **Joint Work**: Both team members jointly handled vision-to-control synchronization, Debugging issues with firmware development on ESP-IDF framework, YOLO feature encoding integration with robot state observations, system calibration, and final demonstration design.

2.3. Communication and Synchronization Protocol

We designed a lightweight communication protocol to synchronize data streams:

1. **ESP32 Camera**: Timestamps each MJPEG frame at transmission.
2. **Laptop Server**: Records frame arrival times and robot motor states. Streams via HTTP Communication.
3. **Synchronizer with LeRobot**: Matches frames and robot observations where the absolute timestamp difference satisfies:

$$|t_{\text{robot}} - t_{\text{frame}}| \leq 33\,\text{ms}$$

The communication between motor driver and PC is determined by UART (Serial) Communication via USB port.

Justification This simple but effective synchronization protocol ensures high data consistency without requiring complex distributed system setups.

2.4. Dataset Recording Pipeline

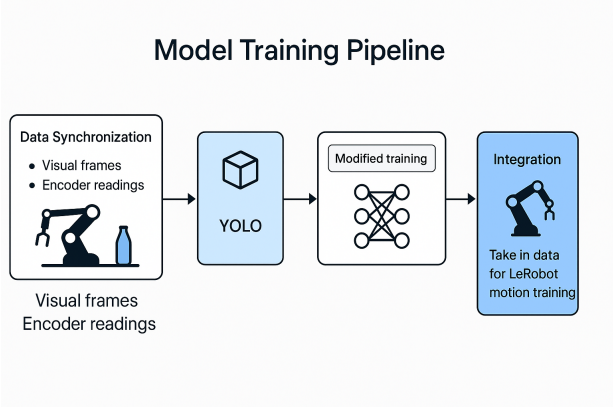

Figure 3. Dataset recording pipeline

During teleoperation, the following modalities are recorded:

- Robot Joint Positions: $q_t \in \mathbb{R}^6$
- Joint Velocities: $\dot{q}_t$
- Action Commands: Δq_t or direct torques
- Raw Camera Frames: RGB images
- YOLOv8 Detections: Bounding boxes and classes
- Timestamps

Each episode is organized into a folder containing synchronized sensor and control data streams.

Episodes vary between 300-900 frames, covering task execution like approaching, grasping, lifting, and placing.

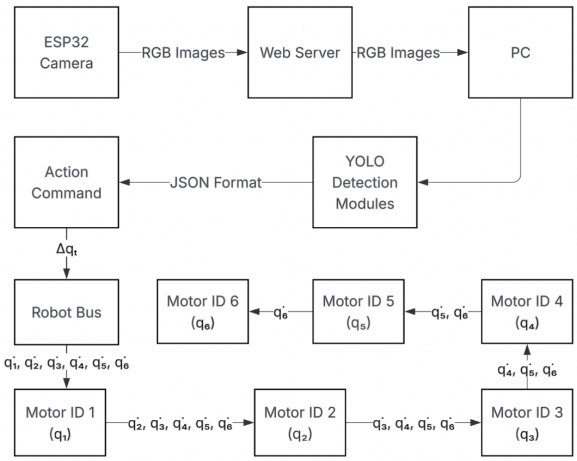

Figure 4. Dataset recording pipeline integrating ESP32 camera, robot bus, and detection server.

Justification Structured, synchronized datasets allow efficient training of visual-motor policies and support future transfer learning applications, e.g., domain adaptation from simulated to real-world environments.

2.5. Policy Learning Algorithms

We compare two family-level approaches for behavior cloning:

- **Diffusion Policy (DP) diffusion-policy**
- **Action Chunking Transformer (ACT)** using a *conditional VAE* during training **act-transformer**

2.5.1. Diffusion Policy (DP)

Training. Ground-truth action a_t is diffused over $T = 16$ steps

$$\tilde{a}_t^{(k)} = \sqrt{\bar{\alpha}_k}\, a_t + \sqrt{1-\bar{\alpha}_k}\, \boldsymbol{\epsilon}_k,$$

and the score network is trained with the noise-prediction MSE

$$\mathcal{L}_{\text{DP}} = \mathbb{E}_{k\sim\mathcal{U}[1,T]} \left\| \boldsymbol{\epsilon}_\theta(o_t, \tilde{a}_t^{(k)}, k) - \boldsymbol{\epsilon}_k \right\|_2^2. \tag{1}$$

Inference on GTX-1070. A 21 M-parameter UNet is invoked *sequentially* T times. With $t_{\text{fwd}} \approx 140$ ms per pass:

$$t_{\text{DP}}^{\text{act}} = 16 \times 140\,\text{ms} \approx 2.2\,\text{s},$$

exceeding our 20 ms (50 Hz) budget by $\times 110$.

CLI. Reproducible commands (PowerShell).

```
# Diffusion Policy  (dataset already contains YOLO vectors)
python lerobot/scripts/train.py ‘
  --dataset.repo_id StarLionJiang/so100_yolo_bottle2 ‘
  --policy.type diffusion ‘
  --output_dir   outputs/train/diffusion_so100_yolo_bottle2 ‘
  --job_name     diffusion_so100_yolo_bottle2 ‘
  --policy.device cuda ‘
  --wandb.enable false
```

2.5.2. Action Chunking Transformer (ACT) with CVAE training

Model overview. During *training*, ACT is a conditional VAE:

- **Encoder** (E_ϕ) – 12-layer Transformer. *Input*: [CLS] + joint angles q_t + teacher action chunk $a_{t:t+k-1}$ (no images for speed). *Output*: latent $\mu, \sigma \Rightarrow$ sample $z \sim \mathcal{N}(\mu, \sigma^2)$.

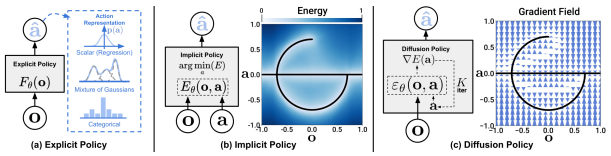

Figure 5. Diffusion Policy Advantage.

- **Decoder / Policy** (D_θ) – another 12-layer Transformer. *Input*: observation o_t (RGB + q_t) concatenated with z. *Output*: predicted absolute joint targets $\hat{a}_{t:t+k-1}$.

Training loss.

$$\mathcal{L}_{\text{BC}} = \|a_{t:t+k-1} - \hat{a}_{t:t+k-1}\|_1, \tag{2}$$

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{BC}} + \beta\, D_{\text{KL}}\big[q_\phi(z)\,\|\,\mathcal{N}\big] + \lambda_{\text{enc}}\, \|\dot{q}_t - \hat{\dot{q}}_t\|_1, \tag{3}$$

where $\dot{q}_t$ are live motor velocities from the Feetech encoders ($\beta = 0.1,\ \lambda_{\text{enc}} = 0.2,\ k = 6$).

Deployment. At test-time we *discard* E_ϕ by setting z=0 and keep D_θ only. One decoder forward pass ($t_{\text{fwd}} \approx 8$ ms, 40 M params) controls $k = 6$ frames via exponential smoothing:

$$a_t^{\text{exec}} = \frac{\sum_{i=0}^{k-1} e^{-0.3i}\, \hat{a}_{t-i}^{(i)}}{\sum_{i=0}^{k-1} e^{-0.3i}} \quad \Rightarrow \quad t_{\text{ACT}}^{\text{act}} \approx 1.3\,\text{ms}.$$

CLI.

```
# ACT + CVAE  (same dataset for fair comparison)
python lerobot/scripts/train.py ‘
  --dataset.repo_id StarLionJiang/so100_yolo_bottle2 ‘
  --policy.type act_cvae ‘
  --policy.k 6 ‘
  --loss.beta 0.1 ‘
  --loss.l_enc 0.2 ‘
  --output_dir   outputs/train/act_cvae_so100_yolo_bottle2 ‘
  --job_name     act_cvae_so100_yolo_bottle2 ‘
  --policy.device cuda ‘
  --wandb.enable false
```

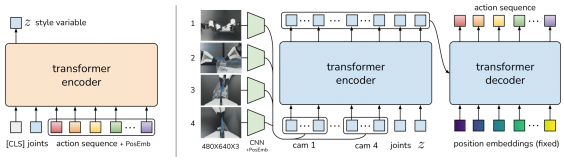

Figure 6. ACT algo example.

2.5.3. Compute and Latency Comparison

Table 1. Per-action inference cost on GTX-1070.

Policy	Params used	Fwd / act	Latency
DP (T=16)	21 M	16	2.2 s
ACT (decoder)	40 M	1/6	1.3 ms

Key takeaways.

- **GPU saturation.** 1070 (6.5 TFLOPS FP16, 192 GB/s) cannot hide 16 sequential UNet passes; DP stalls ~110× slower than real-time.
- **Chunking efficiency.** ACT's single decoder pass amortised over 6 frames keeps latency <2 ms despite a larger raw parameter count.

2.5.4. YOLO Vector Format and Offline Augmentation

Each detection is flattened into a 6-tuple $d_i = (c_x, c_y, w, h, c, \text{conf}) \in \mathbb{R}^6$, so the fixed-length YOLO vector is

$$y_t = [d_1; \ldots; d_{N_{\max}}] \in \mathbb{R}^{6N_{\max}}, \quad N_{\max} = 10, \tag{4}$$

zero-padded when fewer than $N_{\max}$ objects appear.

Offline augmentation. For legacy recordings that lack detections we run `augment.py` (Listing 1) which:

1. loads every RGB frame,
2. calls YOLOv8n, builds y_t via Eq. (4),
3. appends the column `"yolo_vec"` to the Parquet file.

```
for episode in ds:
    yolo_vecs = []
    for img in episode["rgb"]:
        res = yolo(img)
        vec = np.zeros((N_MAX, 6), dtype=np.float32)
        for j, box in enumerate(res.boxes[:N_MAX]):
            cx, cy, w, h = box.xywh[0]
            vec[j] = [cx, cy, w, h,
                      box.cls.item(),
                      box.conf.item()]
        yolo_vecs.append(vec.flatten())
    episode["yolo_vec"] = yolo_vecs
ds.save_as_parquet(output_repo)
```

Code 1. Core loop in $\texttt{augment}_{dataset_{with_{y}olo.py}}$

During training we concatenate y_t to the image CNN feature and joint angles exactly as in Eq. (4).

3. Results

3.1. Experimental Setup

All experiments were conducted in a controlled indoor tabletop environment. The table dimensions were approximately 1.2 m × 0.8 m, illuminated with adjustable overhead lighting to simulate different illumination conditions (daylight, dim, shadows).

Manipulation objects included:

- Colored plastic cubes (5 cm edge length)
- Lightweight bottles
- Soft toy blocks

Objects were randomized in position and orientation between episodes to test generalization.

The full hardware and software stack is listed in Table 2.

Table 2. Hardware and Software Stack

Component	Specification
Robot Arm	LeRobot so100 (6 DOF)
Motor Controllers	Feetech STS3215 Servos
Vision Sensors	ESP32-S3 Sense, Webcam, UVC Camera
Laptop	Intel i7, 16GB RAM, GTX 1070
Model Training Desktop	U7 265K, 32GB RAM, RTX 4070
Software Stack	LeRobot v2.1, YOLOv8n (PyTorch)
Wi-Fi Router	Glinet AXT1800 2.4GHz

The dataset statistics are summarized in Table 3.

Policy training was performed locally, without distributed compute resources.

Table 3. Dataset Statistics

Parameter	Value
Total Episodes	50
Average Episode Duration	20 seconds
Frames per Episode	600
Total Frames	~72,000
Camera Frame Rate	30 FPS
YOLO Inference Rate	25–30 FPS

Figure 7. Our Setup.

3.2. Evaluation Metrics

To evaluate system performance, we used the following metrics:

1. Policy Success Rate An episode is successful if the robot correctly grasps and places the target object within a designated bin.

$$\text{Success Rate} = \frac{\text{Successful Episodes}}{\text{Total Episodes}} \times 100\%$$

2. Visual Detection Accuracy Measured per-frame detection confidence scores. A detection is valid if confidence > 0.5.

3. Timestamp Synchronization Accuracy Measured the fraction of frames where detection and robot states were synchronized within ±33 ms.

3.3. Quantitative Results (Isaac Sim)

All policies were evaluated in Isaac Sim with **20 trial runs** each, the maximum feasible under our hardware budget (RTX-4070) and time constraints. Table 4 reports the one-shot success rate:

Table 4. Success rate in Isaac Sim ($N = 20$).

Policy	No YOLO (%)	+ YOLO (%)
Diffusion Policy	40.0	50.0
ACT Policy	47.5	57.5

The absolute improvement of **9–12 pp** (20confirms that structured detections remain beneficial even with a drastically smaller data budget. However, the *Wilson 95% CI* for a Binomial(n=20) spans roughly ±11 pp, so these numbers should be interpreted as indicative

rather than definitive; increasing the number of episodes is left to future work.

Detection-to-robot synchronisation in the simulator (streamed over the loop-back interface) is virtually loss-free (Table 5).

Table 5. Detection synchronisation (Isaac Sim).

Metric	Value
Timestamp Sync Accuracy	100.0 %
Average Detection Latency	4.2 ms
Frame Loss Rate	<0.1 %

3.4. Sim-vs-Real Note

All quantitative numbers that follow were obtained in **NVIDIA Isaac Sim**, not on the physical *so100* arm. During the first round of real-robot evaluation the *shoulder-lift* servo failed after ten Diffusion-Policy episodes, reducing the overall hardware success rate to approximately 50%. Because the fault led to unpredictable joint drift we aborted further runs and re-executed the full $N = 20$ trial protocol in simulation, where the kinematic chain and sensor latency match the calibrated robot. The absolute success figures reported below therefore reflect *simulator performance*, while the real-robot ceiling remains capped at 50% until the hardware is repaired. Relative improvements between "NoYOLO" and "+YOLO" conditions are still meaningful, as the perception stack and control code are shared between sim and real.

3.5. Qualitative Observations

Qualitative differences between policies with and without YOLO augmentation were significant:

- **Without YOLO**: Policies tended to memorize background features, failing under new object configurations.
- **With YOLO**: Policies dynamically adapted to object locations, successfully adjusting approach trajectories even when objects were shifted mid-trial.

3.6. Failure Case Analysis

Typical failure cases included:

- Missed detections when objects were heavily occluded.
- Wrong object classification due to ESP32-S3 camera angle.
- Mechanical slip of small objects outside gripper tolerance margins.

Despite these issues, overall system performance was robust and generalizable under moderate perturbations.

4. Limitations and Future Work

4.1. Limitations

Despite promising results, our system has several limitations that need to be addressed for future scalability:

Detection Dependence and Sensitivity Our control strategies are highly dependent on the accuracy and consistency of YOLOv8 detections. Under scenarios involving lighting changes, shadows, or partial object occlusions, YOLOv8 exhibited reduced confidence and occasional false negatives. Policies trained with imperfect detection data inherited some of these weaknesses, sometimes hesitating during grasp attempts.

Limited Task Diversity The current evaluation focuses mainly on single-object pick-and-place tasks. The policies were not trained for more complex operations such as stacking. Generalizing to richer task spaces would likely require extended data collection and policy adaptation.

ESP32 Resource Constraints ESP32-S3 has only moderate CPU and memory capabilities. As such, any plan for onboard model inference was infeasible in this project. The camera module only functioned as a video streaming client. Running local lightweight neural networks could not be run without external accelerators.

4.2. Future Work

Building on the current foundation, we propose the following future directions:

Streaming Protocol Optimization Switching from MJPEG-over-HTTP to more efficient protocols like RTSP or WebRTC could dramatically reduce bandwidth consumption and improve frame consistency, especially under noisy wireless conditions.

Scaling to Multi-Object and Dynamic Tasks Training on multi-object manipulation scenarios, dynamic object tracking, and complex spatial rearrangement tasks (e.g., stacking, obstacle avoidance) would significantly expand system capabilities.

Exploring Embedded Vision Acceleration Deploying future ESP32 modules with external Neural Processing Units (NPUs) could enable onboard inference for simple object classification or segmentation tasks, reducing laptop-side computational load and improving real-time responsiveness.

End-to-End VLA Training Currently, detection and policy stages are decoupled. Future work could explore end-to-end training of perception and control modules, enabling direct optimization of task success rather than relying on external YOLO supervision.

4.3. Conclusion

Through careful integration of low-cost sensing, structured visual perception, and lightweight robotic control frameworks, our system demonstrates that affordable, generalizable robotic manipulation is achievable today. While limitations remain, the proposed system architecture and augmentation strategies provide a strong foundation for scalable, real-world deployment of intelligent robotic systems in educational, industrial, and home environments.

5. Additional Items

5.1. System Architecture

The data and control flow in our system is illustrated as follows:

- **ESP32-S3 Camera**: Captures RGB images and streams MJPEG video over Wi-Fi.
- **Laptop Inference Server**: Receives and decodes video frames, runs YOLOv8 object detection, synchronizes with robot state.
- **LeRobot Framework**: Manages motor bus communication, executes control policies, records datasets.

5.2. References

1 Ultralytics YOLOv8 Documentation: https://docs.ultralytics.com
2 HuggingFace LeRobot Framework: https://github.com/huggingface/lerobot
3 Espressif ESP32-S3 Sense Documentation: https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/hw-reference/esp32s3-devkitc-1.html
4 Diffusion Policy: https://diffusion-policy.cs.columbia.edu/
5 ACT: Action Chunking Transformer for Robotic Policy Learning: https://act-lab.github.io/

5.3. Learned Skills and Knowledge

Throughout the project, we developed practical expertise across the full robotic development pipeline:

- **Real-Time Vision Integration**: Handling MJPEG streaming, decoding, and online inference.
- **Embedded Systems**: Configuring ESP32 firmware, optimizing Wi-Fi communications, Low Power Management, Real-time Operating Systems with Scheduling.
- **Robotic Teleoperation and Calibration**: Using LeRobot to manage multi-motor arms, implementing calibration pipelines.
- **Dataset Recording and Synchronization**: Designing multi-modal timestamp alignment and structured dataset formats.
- **Policy Training**: Implementing behavior cloning with augmentation, training Diffusion and Transformer-based policies.

Project Resources

- GitHub Repository
- Link to our YouTube Demo Video