Design of a Hybrid-Control Soccer Robot using an ESP32 and Laptop-Based Computer Vision

Chacón Gómez. José Daniel
Computer Engineering Student
Universidad Nacional Experimental del Táchira (UNET)
San Cristóbal, Táchira, Venezuela
josedaniel.chacon@unet.edu.ve

Castillo Gimenez. Alba Patricia
Mechanical Engineering Student
Universidad Nacional Experimental del Táchira (UNET)
San Cristóbal, Táchira, Venezuela
alba.castillo@unet.edu.ve

***Abstract*—This paper presents the *Auto Soccer Bot*, a low-cost mobile robot for robot-soccer tasks built around an ESP32-CAM and a laptop-based perception stack. The system follows a hybrid control paradigm with two modes: (i) manual teleoperation driven by vision-based hand gestures (MediaPipe) and (ii) an automatic mode in which the ESP32-CAM streams MJPEG video over Wi-Fi to a host application that performs hybrid ball detection (HSV color thresholding plus YOLO) and feeds a finite-state machine (FSM) for control. All commands are transported via an HTTP API (control) decoupled from the MJPEG streaming service. The architecture emphasizes low latency through latest-frame retention and rate-limited, deduplicated command posts. In addition to the live ball detector, we train and validate YOLOv11 models for the `goal` and `opponent` classes (validation mAP@0.5 = 0.991), which are not yet integrated into the automatic mode's state machine. We detail the hardware, firmware, and host-side software, and report on the current project status: robust manual control and reliable ball-following in the automatic loop, with multi-object decision fusion left for future integration.**

***Index Terms*—Mobile Robotics, ESP32-CAM, Computer Vision, HTTP, Teleoperation, Hand Gesture Control, Object Detection, YOLO**

I. Introduction

Autonomous mobile robotics—and robot soccer in particular—offers a compact setting to integrate perception, decision-making, and control. A persistent challenge is enabling advanced visual perception on low-cost platforms without overburdening the on-board microcontroller.

Many educational robots either implement simple on-board autonomy (e.g., line following) or basic remote control. They seldom combine (i) intuitive teleoperation, (ii) vision-in-the-loop autonomy, and (iii) a clean separation between high-level perception and low-level actuation on constrained hardware.

This paper introduces the *Auto Soccer Bot*, which pairs a low-cost ESP32-CAM robot with a laptop-based perception and decision stack. The distributed design offloads computation-heavy vision to the laptop, while the ESP32 focuses on real-time streaming and motor control. The system provides two modes: a manual mode for gesture-driven teleoperation (MediaPipe) and an automatic mode that ingests the ESP32-CAM MJPEG stream and runs a hybrid detector (HSV + YOLO) inside a finite-state controller to follow the ball. Communication uses a simple HTTP API over a local network, decoupling the control plane from the video stream.

Implementation status. Manual teleoperation is robust and operates in real time; the automatic mode achieves reliable ball-following. Beyond the live ball detector, YOLOv11 models for the `goal` and `opponent` classes have been trained and validated (mAP@0.5 = 0.991 on the validation set), but are not yet wired into the automatic mode's state machine; multi-object decision fusion is reserved for future work.

Contributions. This work offers: (i) a low-cost, hybrid teleoperation/autonomy architecture on ESP32-CAM; (ii) a low-latency MJPEG intake with latest-frame retention and resilient HTTP command transport; (iii) a hybrid HSV+YOLO perception pipeline integrated with an FSM for ball following; and (iv) trained and validated detectors for `goal` and `opponent` to enable upcoming multi-object behaviors.

Paper organization. Section II reviews related work. Section III presents the architecture and communication scheme. Section IV details the hardware. Section V describes the firmware and host applications. Section VI reports experiments, and Section VII concludes.

II. Related Work

The design of the Auto Soccer Bot draws upon several established paradigms in mobile robotics. The core architectural choice—offloading computation from a resource-constrained robot to a more powerful base station—is a well-documented strategy for creating low-cost, effective autonomous systems [4]. In this model, the robot acts as a mobile sensor and actuator platform, streaming raw or minimally processed data over a wireless link to a host computer that performs intensive tasks like AI-based decision-making and video processing [4], [5]. This approach, prototyped in systems like the "SAVIOUR" search and rescue platform [4], allows for the use of inexpensive microcontrollers like the ESP32-CAM, which excel at video streaming but cannot run complex vision models locally [5].

For autonomous decision-making, particularly in dynamic environments like robot soccer, Finite-State Machines (FSMs) are a proven and widely used tool [6], [7]. FSMs provide a transparent and manageable way to structure a robot's behavior, defining states such as 'APPROACH_BALL' or 'DRIB-

BLE_TOWARDS_GOAL' and the sensory conditions that trigger transitions between them [6]. The perception systems that feed these FSMs often employ hybrid strategies. While deep learning models like YOLO [3] provide robust object detection, they can be fused with faster, simpler methods to improve real-time performance. Combining a primary detector with a secondary tracking algorithm, such as a particle filter or color-based tracker, creates a more resilient system that can handle momentary detection failures or occlusions [8].

In the domain of human-robot interaction, vision-based gesture control has emerged as an intuitive and low-cost alternative to physical interfaces [9]. Frameworks like Google's MediaPipe have significantly accelerated this trend by providing real-time, high-fidelity hand landmark detection from a standard 2D camera [1], [10]. This enables developers to map specific hand poses (e.g., a closed fist) or continuous movements to robot commands, creating a natural and accessible teleoperation experience without requiring specialized hardware [9].

III. System Architecture

The system (Fig. 1) is a distributed architecture composed of two primary agents: the **ESP32 robot** and the **laptop host**. Computationally intensive tasks (computer vision and decision logic) execute on the laptop, while the ESP32 focuses on real-time video streaming and low-level actuation. Two control loops are available—selected by running the corresponding Python application on the laptop.

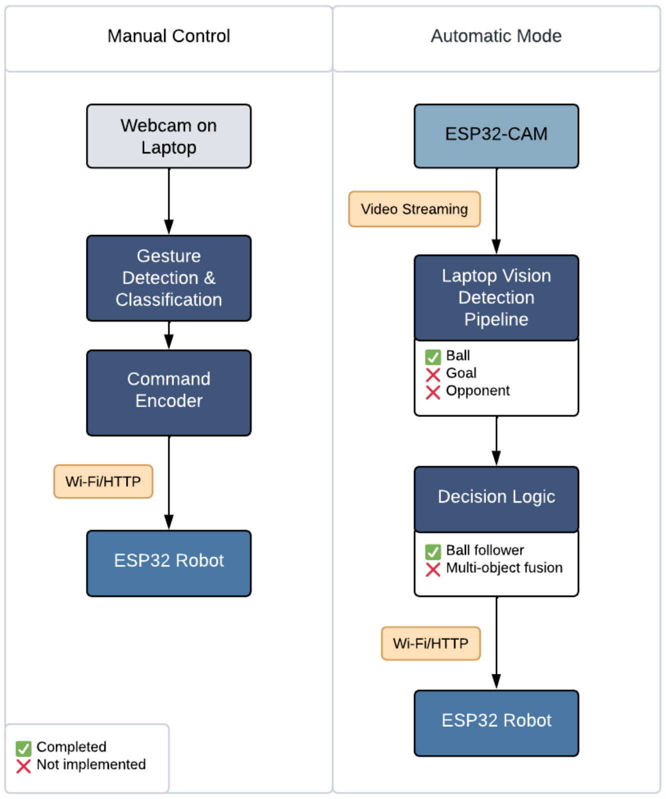

Fig. 1. System architecture and data flow for manual and automatic modes.

A. Manual Control Mode

In manual mode, the primary data flow originates from the laptop webcam and drives a gesture-to-command loop:

1) **Sensing:** A laptop webcam captures RGB frames of the user.
2) **Perception:** A Python application (MediaPipe Hands) detects hand landmarks and classifies the right-hand gesture into discrete commands (e.g., `forward`, `left`, `stop`); the left-hand thumb–index distance is mapped to speed.
3) **Decision/Encoding:** The classified intent is mapped to motion primitives and serialized as JSON.
4) **Actuation:** The command is sent via HTTP `POST` to the ESP32 robot endpoint `/move`. The firmware parses the payload and updates motor PWM and direction.

B. Automatic Mode

In automatic mode, the primary data flow originates from the on-board ESP32-CAM:

1) **Sensing:** The ESP32-CAM streams an MJPEG feed over Wi-Fi at QVGA (320×240) from a dedicated streaming server.
2) **Perception:** A host-side Python application subscribes to the stream and performs *hybrid* ball detection: fast HSV color thresholding every frame, augmented by a YOLO detector every N frames. (YOLOv11 models for `goal` and `opponent` are trained and validated but not yet integrated into this live loop.)
3) **Decision:** The detected ball pose feeds a finite-state machine (FSM) that selects actions such as `SEARCHING` and `APPROACHING` to center the ball and advance toward it.
4) **Actuation:** The chosen action is serialized as JSON and issued to `/move` via HTTP `POST`.

C. Communication Protocol

Laptop–robot communication uses HTTP over a local Wi-Fi network. The ESP32 firmware hosts two independent HTTP servers:

- **Control server (port 80):** Handles non-streaming requests and exposes `/`, `/status`, `/control`, `/capture`, and the motion endpoint `/move` (accepts `application/json`, e.g., `{"direction":"forward","speed":150}`).
- **Streaming server (port 81):** Serves `/stream` as MJPEG using `multipart/x-mixed-replace`.

This dual-server design isolates the control plane from the high-bandwidth video path, minimizing command latency. On the host, latest-frame retention and rate-limited, deduplicated command posts further reduce end-to-end delay and avoid saturating the microcontroller.

Note: The architecture is intentionally decoupled so that additional detectors (e.g., `goal`, `opponent`) can be wired into the perception layer and fused in the FSM without modifying firmware.

IV. Hardware Implementation

The robot uses low-cost, off-the-shelf parts. An ESP32-CAM handles sensing and networking, while an L298N dual H-bridge provides motor power/level shifting. Table I lists the main components.

TABLE I
Key Hardware Components

Component	Primary Role
ESP32-CAM (AI-Thinker)	Camera + Wi-Fi; hosts HTTP servers; provides control signals to motor driver
L298N Dual H-Bridge	Drives two DC motors; fans out power (5 V logic rail)
DC Gear Motors (2x)	Differential drive locomotion
Battery Pack (2S, 2×3.3 V)	System supply (VIN for L298N), master switch inline
Robot Chassis	Structural frame

A. Core Controller and Motor Driver

ESP32-CAM: The AI-Thinker ESP32-CAM module powers from the 5 V rail and exposes two HTTP servers (control and streaming). Its GPIOs are *3.3 V logic*, which is adequate to drive L298N inputs (typical high threshold ≈2.3 V). The module captures video, serves endpoints, and emits PWM/logic signals for actuation via the motor driver.

L298N Motor Driver: The L298N receives the series battery at `VIN` (often labeled "+12V") to power the motor H-bridges. In our build we *keep* the on-board 5 V regulator enabled (the `5V-EN` jumper is installed) and use the board's `+5V` pin to supply both the L298N logic and the ESP32-CAM 5 V pin (see Fig. 2). With a 2S pack of two 3.3 V cells (≈ 6.6 V nominal) this regulator operates near its dropout region; it worked reliably in our tests, though headroom is limited during motor transients.
Robust alternative (optional): If brownouts or camera resets are observed, remove `5V-EN` and feed both the L298N `+5V` pin and the ESP32-CAM from a dedicated 5 V DC–DC module (≥ 1 A), keeping grounds common.

B. Power and Actuation

The platform uses a differential drive with two DC gear motors.

- **DC Motors:** Two 6 V-rated gear motors are driven from the L298N. Direction is set via `IN1--IN4`; speed uses PWM on `ENA/ENB` (ESP32 LEDC). Typical PWM in the low–mid kHz range avoids audible noise while remaining within L298N limits.
- **Power System:** Two 3.3 V cells in series (2S, ≈6.6 V nominal) feed the L298N `VIN` through a master switch (SW1). With `5V-EN` installed, the L298N's on-board regulator provides the `+5V` rail used by both the L298N logic and the ESP32-CAM 5 V pin. All grounds (battery, L298N, ESP32-CAM) are common.

A wiring diagram consistent with this configuration is shown in Fig. 2.

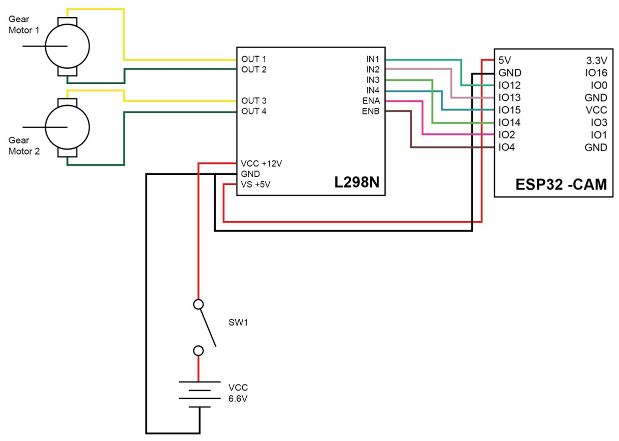

Fig. 2. Circuit diagram showing 2S battery (2×3.3 V) to L298N `VIN`, `5V-EN` installed, and `+5V` powering the ESP32-CAM. Motor outputs are on `OUT1/OUT2` and `OUT3/OUT4`; control lines connect ESP32-CAM GPIOs to `IN1--IN4` and `ENA/ENB`.

V. Software Implementation

Functionality is split between the on-robot firmware (ESP32-CAM) and host-side Python modules. The firmware provides real-time I/O and a small HTTP API; the host performs perception and decision-making. All host modules use isolated virtual environments and configuration files (`manual_control/config.py`, `auto_soccer_bot/config_auto.py`) to keep dependencies and runtime options reproducible.

A. ESP32 Firmware

The firmware is written in C++ atop the Arduino ESP32 core and the ESP-IDF `esp_http_server`. On boot it initializes GPIOs (motor enable/direction), the camera (*QVGA*, JPEG quality 30), joins Wi-Fi (station mode), then starts two HTTP servers.

Dual-server layout: Is managed by the nexts files (inside `esp32cam_robot` directory): `WebServerManager.h/.cpp` and `WebRequestHandlers.h/.cpp`:

- **Control server (port 80)**: routes `/`, `/status`, `/control`, `/capture`, `/move`.
- **Streaming server (port 81)**: route `/stream` (MJPEG, `multipart/x-mixed-replace`).

  Endpoints (schema/semantics).:

- `GET /status` → JSON with heap, FPS estimate, Wi-Fi RSSI, and GPIO state.
- `GET /control?var=`*`name`*`&val=`*`int`* → adjust camera parameters (framesize, quality, brightness, etc.).
- `GET /capture` → single JPEG frame.
- `GET /stream` → continuous MJPEG with explicit boundary; each part carries a JPEG frame.
- `POST /move` (JSON) → motion command:

```
{
    "direction": "
        forward|
        backward|
        left|
        right|
        soft_left|
        soft_right|
        stop",
  "speed": 0..255,
  "turn_ratio": 0.0..1.0 (optional)
}
```

Responds `200 OK` with a small JSON ack; invalid payloads return `400`.

Actuation.: `MotorControl.h/.cpp` abstracts differential-drive control using ESP32 LEDC PWM and L298N direction pins. High-level calls (`moveForward()`, `turnLeft()`, `stopMotors()`) internally clamp PWM duty, apply a calibrated deadband, and translate `turn_ratio` to asymmetric wheel speeds for soft turns.

B. Host Application: Manual Control

The manual teleoperation module (`manual_control/`) implements gesture-driven control using a laptop webcam.

Pipeline (sense $\rightarrow$ interpret $\rightarrow$ command).:

1) **Sensing** (`camera_manager.py`): captures frames via OpenCV with safe open/close and FPS throttling.
2) **Perception** (`hand_detector.py`): MediaPipe Hands returns 21-landmark sets and handedness; frames are flipped to "selfie view" and converted to RGB.
3) **Classification** (`gesture_classifier.py`): the right hand encodes the discrete direction using fingertip-vs-MCP logic; the left hand's thumb–index distance (normalized by frame size) maps to a bounded speed in $[0, 255]$.
4) **Actuation** (`robot_communicator.py`): builds JSON and posts to `/move` asynchronously (HTTPX). A dedup layer suppresses repeated identical commands, and a rate limiter caps send frequency to avoid control-plane flooding.

Runtime configuration.: `config.py` holds the robot IP/ports, webcam index, MediaPipe confidences, and speed mapping bounds. The app runs via `python -m manual_control.main` and provides an overlay window (ESC to exit).

C. Host Application: Automatic Mode

The autonomous controller (`auto_soccer_bot/`) closes the perception-to-actuation loop from the ESP32-CAM stream. The orchestration in `application.py` comprises four concurrent async tasks: stream intake, perception, decision, and command transport.

1) Stream intake (`camera_manager.py`): An `httpx.AsyncClient` connects to `http://<ESP32_IP>:81/stream` with a connect-timeout and no read-timeout. The MJPEG boundary is parsed incrementally; only the *latest* decoded frame is retained (dropping stale frames) to minimize end-to-end latency. Default geometry is QVGA (320×240); an optional resize path is disabled by default.

2) Hybrid perception (`ball_detector.py`).: We combine a lightweight HSV color detector with a scheduled YOLO pass:

- **HSV**: threshold in HSV using `LOWER_BALL_COLOR / UPPER_BALL_COLOR`, mild morphology, and a minimum-area filter.
- **YOLO (Ultralytics)**: executed every `DETECTION_INTERVAL` frames (default 6); detections filtered by `TARGET_CLASS_NAMES` and a confidence threshold. Results are cached with a short TTL to bridge frames between YOLO passes.

Both detectors emit a unified tuple (c_x, c_y, area). A priority rule prefers a valid YOLO result; otherwise the HSV estimate is used.

3) Decision (`robot_controller.py`).: A finite-state machine governs behavior: `SEARCHING` $\rightarrow$ `BALL_DETECTED` $\rightarrow$ `APPROACHING` $\rightarrow$ `CAPTURED`. A target corridor $[x_{\min}, x_{\max}]$ around the image center reduces oscillation; outside the corridor, steering uses soft turns with a configurable `APPROACH_TURN_RATIO`. Confirmation windows and grace timers (`BALL_CONFIRMATION_THRESHOLD`, `BALL_LOST_TIMEOUT_MS`) debounce transitions.

4) Command transferring / transport (`robot_communicator.py`).: Commands are posted to `http://<ESP32_IP>:80/move` with JSON {`direction`, `speed`, `turn_ratio`}. The communicator deduplicates consecutive identical payloads, enforces a minimum spacing between sends, and applies bounded retries with backoff on transient errors. Success/failure is logged for traceability.

Configuration.: All thresholds, URLs, HSV bounds, and YOLO paths live in `config_auto.py`. Default stream is QVGA with moderate JPEG quality; perception uses `SATURATION=3.5` and `BRIGHTNESS=1` pre-enhancement (optional) for color separation.

D. Vision Model Training (`soccer_vision/`)

The `soccer_vision/` module provides training/evaluation for YOLOv11 models targeting two classes: `goal` and `opponent`.

Data and annotation.: Images were labeled in Label Studio and exported in YOLO format (images/labels), then placed under `soccer_vision/dataset/` (`train/`, optional `val/`). Class names are validated against `classes.txt`.

Training workflow.: A notebook `notebooks/01_retrain_yolo.ipynb` calls a helper `notebooks/modules/train.py` to:

1) verify dataset layout and class map;
2) create a validation split if missing;
3) generate a `data.yaml` for Ultralytics;
4) launch YOLOv11 training with pinned seeds and saved checkpoints.

Artifacts (best weights, confusion matrices, PR/F1 curves, logs) are copied to `soccer_vision/results/` for inclusion in the paper. A second notebook `02_test_and_demo.ipynb` demonstrates inference on fresh media.

Reproducibility.: Each Python submodule ships its own `requirements.txt`; environments are created inside the module directories.

VI. Experiments and Results

We evaluated (i) the vision models trained in `soccer_vision/` and (ii) the behaviour of the integrated autonomous loop (streaming $\rightarrow$ perception $\rightarrow$ decision $\rightarrow$ actuation). Unless otherwise stated, Ultralytics defaults were used for training/inference and the ESP32-CAM streamed QVGA (320×240) MJPEG.

A. Vision Model Performance

We trained a lightweight **YOLOv11s** detector on a custom two-class dataset (`goal`, `opponent`). The pipeline produced a held-out validation split and generated standard artifacts (confusion matrix, PR/F1 curves). Table II summarizes the key metrics;[1] Figures 3–4 show the confusion matrix and PR curve.

TABLE II
YOLOv11s Validation Performance (2 classes)

Metric	Value
mAP@0.5 (macro, all classes)	0.991
Peak F1 score	0.86–0.90
Average Precision (goal)	0.995
Average Precision (opponent)	0.987

The detector achieved near-perfect AP on `goal` (0.995) and robust performance on `opponent` (0.987). The normalized confusion matrix in Fig. 3 indicates a 1.00 correct rate for `goal` and 0.95 for `opponent` (i.e., ≈5% missed as background). The PR curve in Fig. 4 shows consistently high precision across recall levels, supporting the suitability of the model for downstream control.

B. Integrated System Evaluation

Setup and instrumentation.: We evaluated the full autonomous loop (streaming $\rightarrow$ perception $\rightarrow$ decision $\rightarrow$ actuation) under the runtime used in Sec. V-C: QVGA (320×240) MJPEG, JPEG quality 30, YOLO scheduled every N=6 frames, HSV every frame, and the FSM in `RobotController.py` with confirmation windows and a horizontal target corridor. We logged timestamps at each stage, tracked which detector (YOLO cache vs. HSV) provided the active observation, and recorded command emission statistics (rate limiting & deduplication).

Perception evolution: color-only $\rightarrow$ YOLO-only $\rightarrow$ hybrid.:

1) *Color-only:* Starting with pure HSV thresholding delivered per-frame responsiveness but was brittle under illumination and background hues. The controller reacted to

[1] mAP@0.5 is the mean Average Precision computed at IoU threshold 0.5. Peak F1 is the maximum harmonic mean of precision and recall across score thresholds.

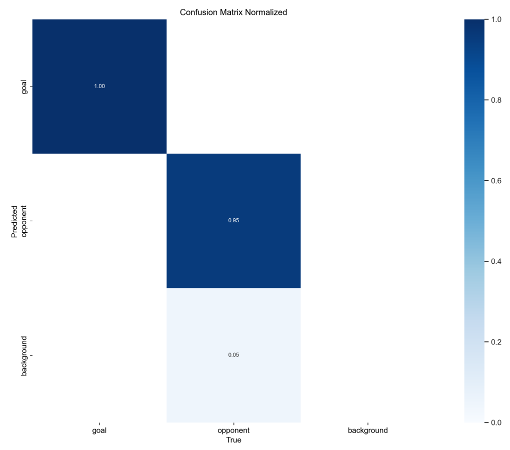

Fig. 3. Normalized confusion matrix for YOLOv11s.

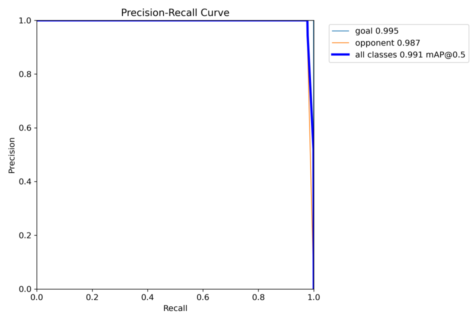

Fig. 4. Precision–Recall curve; overall mAP@0.5 = 0.991.

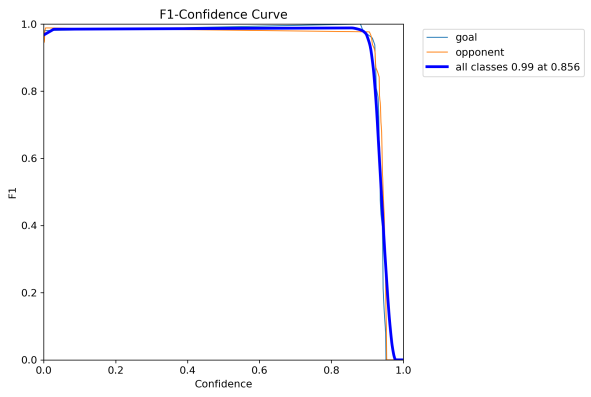

Fig. 5. F1 curve across score thresholds.

instantaneous left/right sign changes of the ball's image x position, yielding rapid left–right pivots ("thrashing") around center. Because actuation lags sensing, the robot often *chased the past*: by the time a pivot executed, the next frame suggested the opposite correction.

2) *YOLO-only:* Switching to YOLO improved robustness to lighting and partial occlusions, but introduced heavier, bursty compute. When run every frame, inference accumulated and, together with naive URL capture, built frame queues; when run sparsely, transient misses caused state flapping (approach $\leftrightarrow$ search). Either way, decisions could be based on stale frames, which again manifested as over-corrections.
3) *Hybrid + fresh intake:* The final design combines both detectors and fixes intake. `BallDetector` runs YOLO every `DETECTION_INTERVAL` frames and treats detections as valid for `yolo_ttl_frames = max(N×2, 3)` frames; otherwise it returns the current HSV result. In parallel, the intake moved to an HTTPX MJPEG parser with *latest-frame retention*, eliminating queue buildup and keeping decisions anchored to the most recent image.

Controller refinements that stabilized heading.: We complemented the hybrid perception with three mitigations in `RobotController.py`:

1) **Confirmation window (`BALL_DETECTED`).** On first sighting, the FSM requires `BALL_CONFIRMATION_THRESHOLD` consecutive detections (tracked by `ball_detected_counter`) before committing to `APPROACHING_BALL`. During this window, corrective turns are issued at a linearly decaying speed and a shrinking grace timeout (`MAX_ADJUSTMENT_TIMEOUT_MS`) prevents premature aborts on brief misses.
2) **Target corridor with soft turns (`APPROACHING_BALL`).** Instead of pivoting on instantaneous lateral error, the controller defines pixel thresholds $[x_{\min}, x_{\max}] = [$`TARGET_ZONE_X_MIN` $\cdot W$, `TARGET_ZONE_X_MAX` $\cdot W]$. If the ball center x lies outside, it applies *soft* turns (`soft_left`/`soft_right`) with bounded `APPROACH_TURN_RATIO`; otherwise it `forwards`. This deadband filters jitter without needing an explicit normalized error function.[2]
3) **Loss timers.** Short gaps trigger a safe `stop` while maintaining state; prolonged loss beyond `BALL_LOST_TIMEOUT_MS` reverts to `SEARCHING_FOR_BALL`. In the adjustment phase, a separate `adjustment_lost_timer` shortens as confidence grows.

Observed outcomes.:

- **Latency & freshness.** Replacing OpenCV URL capture with HTTPX streaming plus *latest-frame retention* removed queue-induced lag; the loop stayed visually responsive with no observable backlog.
- **Robustness.** The hybrid schedule (YOLO every N with TTL cache, HSV every frame) maintained continuous tracking through short-term color failures and illumination changes without the computational overhead of running YOLO on every frame.
- **Stability.** The confirmation window, corridor deadband, and soft turns eliminated center-crossing oscillations and reduced state flapping; heading adjustments became gradual and monotonic.
- **Command-plane health.** Deduplication and a minimum send interval prevented redundant `/move` posts from flooding the ESP32, yielding more predictable actuation timing.

Manual teleoperation sanity check.: The gesture-based manual mode remained smooth in front-lit scenes; failure cases (partial occlusions, non-frontal hand poses) were mitigated by raising MediaPipe confidence thresholds and improving lighting.

Limitations.: Results reflect one camera geometry (QVGA) and our custom dataset; generalization across fields and lighting warrants further trials. The autonomous FSM currently fuses only the ball detector; integrating `goal`/`opponent` into decision-making is future work (Sec. VII). Finally, control uses unencrypted HTTP suitable for a lab LAN but not for untrusted networks.

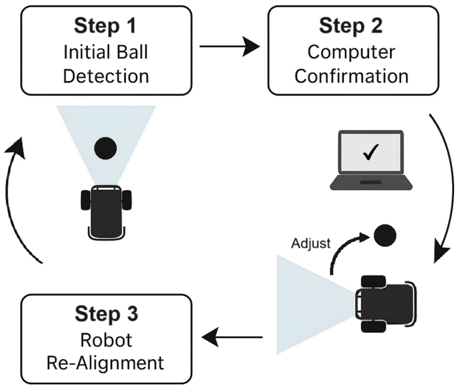

Fig. 6. Corridor-based heading realignment with soft turns and confirmation window.

VII. Conclusion and Future Work

A. Conclusion

This paper presented the *Auto Soccer Bot*, a low-cost, hybrid-control mobile robot that pairs an ESP32-CAM platform with a laptop-based perception and decision stack over an HTTP API. By offloading compute-intensive vision to the host while keeping real-time actuation on the robot, the system achieves behaviors otherwise infeasible on the microcontroller alone.

Summary of contributions.

[2]Conceptually akin to a deadband on $e_x = \frac{c_x}{W} - 0.5$, but implemented with pixel thresholds for simplicity.

- A complete **dual-mode control** pipeline: (i) manual teleoperation via MediaPipe-based hand gestures and (ii) autonomous ball pursuit driven by host-side perception and an FSM.
- A **hybrid perception** design that combines YOLOv11s (scheduled every N frames with a short TTL cache) and per-frame HSV color detection, delivered over a low-latency HTTPX MJPEG intake with latest-frame retention.
- A **stabilized controller** featuring a confirmation window, a horizontal target corridor with soft turns, and loss timers, implemented in `RobotController.py`, which eliminates center-crossing oscillations and state flapping.
- A **trained two-class detector** (`goal`, `opponent`) with strong validation performance (mAP@0.5 = 0.991), reproducible via the `soccer_vision` notebooks and scripts.

Across experiments, the combination of hybrid detection, fresh-frame intake, and FSM refinements resolved the main engineering issues observed early on (stream latency, stale decisions, heading oscillations), resulting in responsive and predictable autonomous behavior.

B. Future Work

The current system validates the architecture and demonstrates robust ball-following; several extensions are natural:

- **Multi-object decision fusion.** Integrate the validated `goal` and `opponent` detectors into the perception-to-action loop. Extend the FSM to handle *align-to-goal*, *shoot*, *avoid-opponent*, and *re-acquire* behaviors with explicit priority/rules.
- **Controller upgrades.** Explore lightweight proportional control on lateral error within the corridor (bounded gains) and simple anticipatory terms (e.g., short horizon smoothing) while preserving the deadband to avoid oversteer.
- **Evaluation at scale.** Add quantitative trials over diverse lighting/fields and report latency histograms, success rates for re-acquisition, time-to-center, and approach stability.
- **Multi-robot coordination.** Add inter-robot communication for role assignment (attacker/defender) and simple collision avoidance, keeping the distributed compute model.
- **Transport and security.** Replace unencrypted HTTP with a minimal authentication layer (pre-shared token) or migrate commands to a lightweight UDP/WebSocket channel with sequence numbers; consider TLS for untrusted networks.
- **Model portability.** Package YOLO weights and configs via artifact storage/Git LFS and support dynamic model selection (CPU vs. GPU) with on-start sanity checks.

These extensions preserve the system's core design—a thin, real-time robot and a flexible, host-side intelligence—while enabling richer autonomous play and more robust deployments.

References

[1] C. Lugaresi, J. Tang, H. Nash, C. McClanahan, E. Uboweja, M. Hays, *et al.*, "MediaPipe: A Framework for Building Perception Pipelines," *arXiv preprint arXiv:1906.08172*, 2019.

[2] G. R. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[3] Ultralytics, "YOLO by Ultralytics (v11)," GitHub repository, 2024. [Online]. Available: https://github.com/ultralytics/ultralytics

[4] A. A. A. Sethaputri, A. F. P. A. P. Putra and I. K. E. Purnama, "Displacing computing operations to an operator station over a wireless link in autonomous mobile robots," *International Journal of Computer and Communication Engineering*, vol. 1, no. 2, pp. 125–129, 2012.

[5] S. Tatipamula, "Computer Vision with OpenCV on ESP32-CAM: Building Intelligent Vision Systems," *ThinkRobotics*, 2023. [Online]. Available: https://thinkrobotics.com/blogs/learn/computer-vision-with-opencv-on-esp32-cam-building-intelligent-vision-systems

[6] J. J. G. R., A. M. L. G. and J. S. A., "Decision-making system of soccer-playing robots using finite state machine based on skill hierarchy and path planning through Bezier polynomials," in *2017 IEEE 9th International Conference on Humanoid, Nanotechnology, Information Technology, Communication and Control, Environment and Management (HNICEM)*, 2017, pp. 1–6.

[7] J. G. Guarnizo and M. Mellado Arteche, "Robot Soccer Strategy Based on Hierarchical Finite State Machine to Centralized Architectures," *IEEE Latin America Transactions*, vol. 14, no. 8, pp. 3586–3596, 2016.

[8] Y.-C. Lin, C.-Y. Lin, C.-C. Chen and C.-Y. Chang, "A Hybrid YOLOv4 and Particle Filter Based Robotic Arm Grabbing System in Nonlinear and Non-Gaussian Environment," *Sensors*, vol. 21, no. 10, p. 3430, 2021.

[9] Y. Chen, Z. Wang, Z. Li, and S. Li, "Vision-based Gesture Tracking for Teleoperating Mobile Manipulators," in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2022, pp. 8889–8895.

[10] A. M. Al-Khafaji and H. T. S. Al-Rikabi, "Tracked Robot Control with Hand Gesture Based on MediaPipe," *Journal of Engineering*, vol. 29, no. 6, pp. 123–136, 2023.