AI-Assisted Match Scouting for FIRST Robotics Competition: A Hybrid Computer Vision Approach to Ball and Robot Tracking

Ze Rui Zheng

August 2026

Abstract

In FIRST Robotics Competition (FRC), teams scout other robots to make strategy decisions, especially during alliance selection. Tools like Statbotics help because they estimate team performance from official match results, but they don't measure what a robot actually does during a match. Each team also plays a limited number of qualification matches, so those estimates get skewed by alliance partners, opponents, penalties, ranking-point strategies, and other things a robot can't fully control.

I tried to build a video-based AI scouting system that watches public match footage and tracks game pieces ("fuel") and robots. It's meant to detect scoring events and work out which robot made each one. I tested three methods for ball tracking: a promptable segmentation model, classical color-based detection, and a smaller real-time detector. I tested four for robot tracking: segmentation, a multimodal large language model, a distilled detector, and bumper-color detection combined with a vision-language model that reads team numbers.

Ball tracking worked reasonably well. Robot tracking was much harder, and it wasn't reliable enough to automate this season. The main problem was losing a robot when another robot, a field element, or the camera view hid it. So the final system was a hybrid: a human traced the robots and marked shots, and the system tracked fuel automatically and helped with shot detection and attribution.

Last, I propose an attention-based multi-camera approach as a possible way to improve robot tracking in future seasons.

Keywords: *computer vision, object detection, video-based tracking, multimodal large language models, robotics scouting, FIRST Robotics Competition*

1 Introduction

1.1 The Role of Scouting in FRC

In FRC, scouting means collecting information about other teams while they compete. Teams use it to learn what different robots are good at and to make strategy decisions. It matters most during alliance selection, when teams pick playoff partners whose abilities fit their own robot.

1.2 Limitations of Existing Tools

Statbotics is a widely used tool that estimates robot performance from official match results and historical statistics. It's useful, but it doesn't directly measure everything a robot does during a match.

Most teams only play around ten qualification matches at an event, so the estimate rests on a small amount of data. It's also affected by alliance partners, opponents, penalties, ranking-point goals, and other factors outside one robot's control. A robot can end up looking stronger or weaker in the statistics than it really was.

Team 334 saw this between competitions. After its first event, its Expected Points Added (*EPA*) was around 60. After the team improved the robot, it performed closer to an 80-EPA level at its next competition. Statbotics needs new match data before its estimate changes, so the improvement took a while to show up.

1.3 Challenges Specific to the 2026 Game

The 2026 FRC game added another scouting problem. During a match, dozens of small yellow game pieces called "fuel" can be moving around the field, and several robots can shoot at nearly the same time.

Counting every scored ball by hand over a two-and-a-half-minute match is hard, especially when balls overlap or move together. Other teams working on video scouting have run into similar problems.

Hand scouting and aggregate statistics have another problem: total points aren't always enough. In alliance selection, a team might care whether a robot is good in autonomous, shoots well from a specific area, or can fill a particular role. That information is more useful when it's measured for each robot separately.

1.4 Objectives

I wanted a system that could watch match video and measure what each robot actually did. It needed to:

- track fuel across video frames,
- track robot movement across the field,
- detect when shots happened,
- connect those shots to the correct robot, and
- produce per-robot statistics that could be used by a scouting system.

The plan was to automate the parts computers are actually good at and use human input where the computer vision was still unreliable.

2 Related Work and Tools Used

This project uses several recent computer vision and multimodal AI tools. There isn't much published work specifically on FRC video scouting, though other teams have tried similar ideas.

The models I used fall into three groups: promptable segmentation, real-time object detection, and multimodal large language models.

Meta's SAM 3 (Segment Anything Model 3) is a foundation model that can detect, segment, and track objects from a text prompt or example image. I used it with prompts such as "yellow ball" and "FRC robot" [1].

RF-DETR is a real-time transformer-based object detector from Roboflow designed for custom datasets and relatively fast inference [2]. It belongs to the larger group of real-time object detectors that includes the YOLO family [6].

Gemini 3 Flash is a multimodal model from Google DeepMind that works with images and video along with text [3]. Qwen3-VL is an open-weight vision-language model from Alibaba's Qwen team [4]. I used Qwen3-VL mainly to read team numbers from robot bumpers.

Statbotics is the existing statistics-based baseline for comparison [5].

3 System Overview

The system takes a match video as input. FRC posts match videos to YouTube soon after matches are played. The system also needs the participating team numbers, the camera view to use, the target frame rate, and the time range to process.

It uses camera calibration to map image coordinates from the video onto a top-down field map. The deployed version combines human robot tracing with automated fuel tracking. Fuel is tracked across frames so the system can detect and attribute successful shots. Figure 1 shows how the stages fit together.

The main outputs are:

- annotated videos showing detected balls, robots, and shots,
- robot movement paths and movement timelapses,

- team statistics and estimated scoring by match period,
- ferry counts, meaning game pieces moved without being scored, and
- whether a robot was disabled during the match.

The annotated videos, paths, and timelapses were mostly for debugging and checking that the system worked. The real scouting output is the per-robot statistics table, which can be pulled through an API or entered into a scouting application.

I built the system with Gradio, so it runs through a web interface or an API. The data goes into Castle, Team 334's open-source scouting application (`https://github.com/Team334/Castle`; `https://castlescouting.com`).

3.1 Field Calibration

Before the system can tell where a robot is on the field, it has to convert a pixel location in the video into a real field location. In the video, a robot's position is just something like "pixel (x, y)," which isn't useful for scouting, so the system needs a transformation from the camera image to a top-down field map.

The transformation is computed from eight landmarks. Each landmark has a clicked pixel location in the video and a known field location measured from the official field drawings. For the two side cameras, and for the center camera's default alignment, the system uses OpenCV's `cv2.findHomography` to turn those landmark pairs into a mapping from the image to the field. For the center camera's default alignment it uses RANSAC, so a single badly placed landmark is ignored instead of pulling the whole fit off.

For the center camera, a user can also recalibrate the view by clicking the same eight landmarks. In that case the system fits a simpler affine transformation, which doesn't model perspective but is less sensitive to clicking error. For quick hand recalibration, that trade-off was worth it.

After calibration, the system reports how far each landmark ended up from its true field position. A landmark with a large error is usually the one that was clicked badly, so that's the one to re-click.

4 Ball Tracking

Ball tracking was one of the more successful parts of the project. I compared three approaches, summarized in Table 1.

4.1 SAM 3 (Promptable Segmentation and Tracking)

With the text prompt "yellow ball," SAM 3 generally found fuel on the field, including moving balls, blurry balls, and crowded scenes. That mattered because match video often

has background objects that look yellow too.

SAM 3 also has built-in tracking, so it can try to keep the same identity for an object across frames instead of treating every frame as a new detection.

The main detection code is shown in Listing 1.

```
def _run_sam3_on_region(frame_bgr, predictor, min_radius, max_radius,
                        x_offset=0, y_offset=0):
    """Run SAM 3 on an image region using a text prompt."""
    detections = []
    predictor.set_image(frame_bgr)
    results = predictor(text=["yellow ball"])

    for result in results:
        if result.boxes is None:
            continue
        for box in result.boxes:
            x1, y1, x2, y2 = box.xyxy[0].cpu().numpy()
            cx = int((x1 + x2) / 2) + x_offset
            cy = int((y1 + y2) / 2) + y_offset
            radius = int(max(x2 - x1, y2 - y1) / 2)
            if min_radius <= radius <= max_radius:
                detections.append((cx, cy, radius))
    return detections
```

Listing 1: SAM 3 text-prompted ball detection, adapted from `_run_sam3_on_region`.

There were two main problems.

First, SAM 3 is computationally expensive and needs a GPU. Second, its built-in tracker sometimes mixed up balls when several identical-looking balls were close together.

This is harder than tracking something like people, because every fuel ball looks almost the same. When two balls cross paths, the model has very little visual information to tell which is which.

Results improved when I combined SAM 3's detections with custom tracking. Instead of trusting SAM 3's identity tracking alone, the system uses each ball's recent motion and expected trajectory to decide which detection belongs to which ball.

4.2 HSV Color Tracking

HSV color tracking was the cheapest way to detect fuel.

There's no AI model involved. The system looks for pixels that fall inside a yellow color range, then finds connected regions that look roughly like balls.

```
# Yellow fuel: hue/saturation/value thresholds tuned for the field
    carpet

lower_yellow = np.array([15, 60, 40])
upper_yellow = np.array([85, 255, 255])
```

```
5
6 hsv = cv2.cvtColor(frame_bgr, cv2.COLOR_BGR2HSV)
7 mask = cv2.inRange(hsv, lower_yellow, upper_yellow)
8 mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel, iterations=1)
9 mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel, iterations=1)
10 contours, _ = cv2.findContours(
11     mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE
12 )
13
14 # Contours are then filtered by area/circularity; contours that are
15 # too large or insufficiently circular are split via a distance-
     transform
16 # watershed step to separate balls that are touching or overlapping.
```

Listing 2: HSV color-space fuel detection, adapted from `detect_fuel`.

It was fast and didn't need a GPU, but it was very sensitive to the environment.

Several balls close together could merge into one detected object. People wearing yellow could be mistaken for fuel, and yellow walls or field elements caused false detections.

It worked fairly well at the NYC Regional because the venue didn't have many yellow background objects and there was unusually little visible audience. So HSV tracking can work well in the right environment, but how reliable it is depends heavily on the venue.

4.3 Distilled YOLO / RF-DETR Model

I also tested a smaller detector distilled from SAM 3.

I used SAM 3 to label balls in frames from multiple matches, then trained a smaller YOLO- or RF-DETR-based model on those labels.

The smaller model was much faster and cheaper than SAM 3, but less accurate in testing. It sometimes missed balls that were only partially visible, and the training set didn't have enough variation between venues and camera positions.

It wasn't the most accurate method this season, but it looks promising for a larger future system because the final detector would be much cheaper to run.

4.4 Scoring Detection

When a ball is shot, the system has to figure out whether it's actually going into the goal. That gets hard when several balls overlap or when a ball leaves the camera view before it reaches the goal.

Instead of needing to see the whole shot, the system uses the ball's recent movement to predict where it's going. First, it looks for a short burst of upward movement to decide whether a ball has probably been shot. Using several recent frames is more reliable than checking one frame.

```python
def _get_shot_launch_metrics(self, prev_pos, recent_positions, current_y):
    """Measure whether a ball is lifting into a shot over a short window."""
    instant_rise = 0.0
    if prev_pos is not None:
        instant_rise = prev_pos[1] - current_y

    y_history = [
        p[1]
        for p in recent_positions[-(self.motion_history_size - 1):]
    ]
    y_history.append(current_y)

    window_gain, rising_steps = 0.0, 0
    if len(y_history) >= 2:
        window_gain = y_history[0] - y_history[-1]
        rising_steps = sum(
            1
            for y0, y1 in zip(y_history, y_history[1:])
            if (y0 - y1) >= self.min_launch_rise_pixels
        )

    return instant_rise, window_gain, rising_steps
```

Listing 3: Shot launch heuristic, adapted from `BallTracker._get_shot_launch_metrics`.

Once a shot is detected, the system estimates the ball's current velocity from its last few positions, giving newer frames more weight because they better show where the ball is heading now. It then predicts the ball's path with a simple projectile model that includes gravity but ignores air drag, which is reasonable over the short time a fuel ball is in flight. The model runs in the 2D image rather than in 3D, which isn't exact, but over a fraction of a second the error is small compared with detection noise.

If the predicted path enters the goal region and stays there for multiple frames, the shot is counted as a predicted make. The system also checks that the ball actually moved toward the goal. This helps separate a real shot from a ball that a robot just pushed sideways.

The scoring detector uses this simple physics to answer one practical scouting question: "Does this ball look like it was shot toward the goal?"

4.5 Data Association: Keeping Track of the Same Ball

The ball detectors find balls in each frame, but they don't know which ball in one frame is the same ball from the previous frame.

For example, suppose two balls are close together and then move in different directions. The system has to decide which new detection belongs to which existing track.

To handle this, each tracked ball keeps its most recent position and an estimate of where

it should move next. When a new frame arrives, the system matches detections to existing tracks so that the total distance between predicted and detected positions is as small as possible. It solves this with the Hungarian algorithm (`scipy.optimize.linear_sum_assignment`).

Matches that are too far apart are rejected, with a looser limit for fast-moving balls and for balls that have been hidden for a few frames, since their predictions are less certain. Unmatched detections start new tracks. Unmatched tracks are kept alive for a few frames using their predicted positions, so a briefly hidden ball can be reconnected to its old track instead of being counted as a new ball.

Matching everything together matters because a greedy rule, "repeatedly match the closest track–detection pair," can swap two balls that pass close to each other.

The result is a set of continuous ball tracks that can be followed across the entire match. That lets the rest of the system know that a ball seen leaving a robot is the same ball that later shows up near the goal.

Method	Type	Strength	Limitation
SAM 3 (prompted)	Foundation segmentation model, GPU-based	Works well with clutter, blur, and noisy backgrounds; includes object tracking	Expensive to run; can mix up nearby identical balls
HSV color tracking	Classical computer vision	Very cheap and fast; no GPU needed	Can fail with yellow background objects or clustered balls; depends on venue
Distilled YOLO / RF-DETR	Detector trained on SAM-3-labeled data	Fast and inexpensive to run	Can miss partially visible balls; needs more varied training data

Table 1: Comparison of the three ball-tracking approaches tested.

5 Robot Tracking

Robot tracking was much harder than ball tracking, and harder than I expected when I started.

Humans are naturally good at it. A person can watch a robot disappear behind another robot and still know it's there. A computer vision model usually only sees what's in the current frame, which makes tracking much harder.

5.1 Camera Placement Constraints

My early testing used an assumed camera position that was higher than the real YouTube broadcast angle, which made robot tracking look more achievable than it was.

In real match footage the side cameras were lower than I expected, so one robot could easily block another.

The side-camera setup also changed between regionals and sometimes even between matches. Cameras could be angled differently, moved slightly, lose signal, or cover different parts of the field depending on the venue.

A large event banner also covered part of some broadcasts and blocked some of the side-camera view.

Because of all this, I couldn't rely on consistent side-camera coverage. In some matches the center camera was basically the only useful view, and if a robot moved into a corner outside it, the system couldn't reliably keep tracking it.

Figure 2 shows a representative frame. The center camera has the alliance scores, match clock, and team numbers above the field. The two side-camera views below show much less of the field and are affected by the event banner.

5.2 SAM 3 for Robot Detection

I also tested SAM 3 for robot tracking, with two types of prompts. The first used text such as "FRC robot." The second used an example image of a robot and asked SAM 3 to find similar objects.

Neither worked well enough.

SAM 3 did much better with a simple object like "yellow ball" than with a more complicated category like "FRC robot." The example-image approach produced false positives and got much less reliable when a robot was partly hidden.

A closer side-camera view might have helped, but SAM 3 wasn't enough for reliable robot tracking in this setting.

5.3 Multimodal LLMs and Distillation

Multimodal large language models worked surprisingly well on simulated match footage. Gemini 3 Flash and similar models could usually locate robots in simulation.

The same approach didn't transfer to real competition video, though. Real match footage has motion blur, camera changes, overlays, compression, and other noise that hurt the model's performance.

There was also a practical problem: running a large multimodal model on every frame of every match would be too expensive.

So I ran a distillation experiment to see whether a smaller model could copy some of Gemini's behavior.

I used Gemini to generate training labels from seven minutes of random simulated match

footage, about 2,100 training frames. A YOLO model trained on those labels reached roughly 90% of Gemini's performance on unseen simulated footage, and it ran much faster and more cheaply.

That's promising, but it's only a simulation result. Gemini itself was unreliable on real competition footage, so this method wasn't reliable enough to use for real matches this season.

5.4 Bumper-Color Detection with Vision-Language Number Reading

The method that worked most consistently for finding robots was based on bumper color.

Every FRC robot has a red or blue bumper depending on its alliance. The system searched for red and blue regions that looked like robot bumpers and treated them as possible robot locations.

Once it found a bumper region, it cropped the region and passed the image to Qwen3-VL 9B to read the team number.

This worked better than standard OCR for several teams, especially when the number font was unusual. The model could also make a reasonable guess when part of the number was blocked.

The main problem was separating robots when bumpers were partly hidden or when two robots from the same alliance were close together and had the same bumper color.

```
# Center-camera bumper color ranges (dark-shade variants included so
# navy/maroon bumpers are not missed)

_BUMPER_BLUE_LOWER = np.array([92, 100, 30])
_BUMPER_BLUE_UPPER = np.array([122, 255, 200])
_BUMPER_RED1_LOWER = np.array([0, 80, 80])
_BUMPER_RED1_UPPER = np.array([10, 255, 220])

# Once a bumper region is cropped, an open-weight vision-language model
# reads the printed team number.

prompt = (
    "What number is the one closest to the center of this image? "
    f"Choose ONLY from: {numbers_str}. Reply with JUST the number. "
    'If unsure, say "none" -- accuracy matters more than guessing.'
)
```

Listing 4: Bumper color ranges and the Qwen3-VL team-number prompt, adapted from `detect_robots_by_bumper_color` and `query_local_llm_for_team_number`.

5.5 Multi-Camera Position Fusion

The two side cameras sit at opposite ends of the field, one behind the blue alliance wall and one behind the red, so the same robot is often visible in both. Instead of picking one camera and ignoring the other, the system combines the two position estimates. It first converts each camera's estimate into field coordinates using that camera's own calibration, then takes a weighted average.

Each camera's weight depends on how big the robot looks in that camera, based on the robot's bounding-box area. A robot that looks bigger is closer to the camera, so that camera's estimate is more accurate, and the weight grows with the square of the area. A camera is also trusted much less (weight multiplied by 0.05) when the robot is on the far half of the field. There the robot is at the edge of the calibrated region and is more likely to be partly blocked by other robots.

This turns the intuitive rule "trust the camera where the robot looks bigger" into a number the system can compute.

5.6 Why Automated Robot Tracking Fell Short

The biggest problem across all the robot-tracking methods was keeping track of a robot once it disappeared from view.

A human can see that a robot is probably still behind another robot or a field structure. A typical detector has no memory like that. If the robot isn't visible in the current frame, the detector may simply stop detecting it.

This matters most for shot attribution. If a robot is hidden while shooting, the system might give the shot to whichever robot happens to be visible in front of it.

Recognizing the act of shooting is hard too. FRC robots have very different designs, so a model trained on a limited set of robot shapes may not generalize to unusual mechanisms.

6 The Deployed Hybrid Human-in-the-Loop System

Automated robot tracking was still unreliable, so the system I used at competition was a hybrid instead of fully automatic.

A human scout traced each robot's position through the match video and clicked when the robot was shooting. The computer handled what it was better at: tracking fuel, detecting shot motion, and matching shots to the human-provided robot position and timing.

That's less autonomous, but it was much more reliable for actual competition use.

The human path is stored as a small number of timestamped points. The system interpolates between those points to estimate the robot's position at any time in the video.

Method	Best result	Why it fell short
SAM 3 (text / exemplar prompt)	Weak overall	Broad category prompts ("FRC robot") worked worse than simple ones ("yellow ball"); performance dropped under occlusion
Gemini 3 Flash (multimodal LLM)	Reliable in simulation only	Real match footage caused much more noise; too expensive to run on every match
Distilled YOLO (from Gemini)	~90% of Gemini's performance, simulation only	Simulation-only result; not shown to be reliable on real footage
Bumper color + Qwen3-VL 9B	Most consistently useful	Has trouble with hidden bumpers and nearby robots with the same alliance color

Table 2: Comparison of the four robot-tracking approaches tested.

```python
def _interpolate_manual_robot_position(robot_track, target_time):
    """Interpolate a human-traced robot position at an arbitrary time."""
    times = robot_track.get("times") or []
    samples = robot_track.get("samples") or []

    if not times or not samples:
        return None

    if target_time <= times[0]:
        _, x, y, shooting = samples[0]
        return x, y, bool(shooting)
    if target_time >= times[-1]:
        _, x, y, shooting = samples[-1]
        return x, y, bool(shooting)

    idx = bisect_left(times, target_time)
    t0, x0, y0, _ = samples[idx - 1]
    t1, x1, y1, _ = samples[idx]

    alpha = (target_time - t0) / max(t1 - t0, 1e-6)

    x = x0 + ((x1 - x0) * alpha)
    y = y0 + ((y1 - y0) * alpha)

    shooting = _manual_track_shooting_active_at_time(
        robot_track, target_time
    )

    return x, y, shooting
```

Listing 5: Interpolating a human-traced robot path between marked samples, adapted from `_interpolate_manual_robot_position`.

The resulting statistics were more detailed than aggregate values such as EPA because they describe what each robot actually did during the match.

The statistics went into Castle to support alliance-partner decisions based on robot-specific data instead of rankings or averages alone.

6.1 Score Grounding with the Broadcast Score

The system was later improved by comparing its shot count with the actual score shown on the broadcast.

The scoreboard can tell the system that, for example, three points were just added. It can't tell the system which robot scored them. The system reads the score from the broadcast overlay, and when the score goes up it handles the increase in two steps. First, any made shots the human scout explicitly marked in the window just before the increase are credited directly. Second, the points that remain are divided as evenly as possible among the robots marked as shooting during that window. When they don't divide evenly, the leftover shots go to the robots with the largest remainders, with ties broken by a fixed order, so the totals always add up to the scoreboard. The same splitting is reused to spread a robot's corrected makes across match periods (autonomous, teleoperated, endgame) in proportion to the makes already detected in each period.

There's one limitation. With an equal split, ties are common, and the fixed tie-break always favors the same team label. Over a full event, those small errors add up in the same direction instead of canceling. A fairer option is to carry each robot's unused fraction forward to the next score change, so the rounding errors cancel over a match instead of piling up on one robot.

This gives the computer vision system an extra check. Instead of trusting every detected shot on its own, it uses the scoreboard as a second source of information: the vision system decides *who* shot, and the scoreboard decides *how many* went in.

7 Discussion and Limitations

Ball tracking was generally successful, especially when SAM 3 detections were combined with trajectory-based tracking. The main limitation is that SAM 3 needs a GPU, while the cheaper HSV method depends heavily on a background with no other yellow objects.

Robot tracking was the main bottleneck. Every automated method I tested did worse than a human scout, mostly because robots disappear behind objects and the available camera views change between venues.

There's also a limitation in how I evaluated all this. I tested the system qualitatively and used it during real competition, so the question was whether the output was accurate and consistent enough to be useful for scouting. I didn't finish a formal benchmark against

hand-labeled ground-truth video. The results in Tables 1 and 2 should be read as practical observations, not a full quantitative comparison.

A stronger evaluation would use a manually labeled set of match video, with every robot and ball labeled frame by frame, and compare the system's detections against that ground truth.

Detection would be scored with the standard precision and recall, counting a detection as correct when its intersection-over-union with a labeled region is above a threshold. Those metrics aren't enough for scouting, though, because they don't notice when a correctly detected shot is credited to the wrong robot. Crediting the wrong robot is worse than missing a shot: a miss lowers one robot's count slightly, while a misattribution lowers one robot's count *and* inflates another's.

So the evaluation should separate detection from attribution. Attribution accuracy would be the fraction of correctly detected shots that the system credited to the right robot. A second useful measure is the average per-robot count error, which captures misses, false detections, and misattributions together. The scoreboard check keeps the totals matching, so for shots inside a scoring window, the remaining error in that measure would come from how credit is split rather than from how many shots were counted.

Building a hand-labeled validation set and measuring these metrics is the next step toward a more rigorous evaluation.

8 Future Work: Attention-Based Multi-Camera Vision

The main future direction is a system that uses several cameras and several moments in time at once.

The current system mostly depends on what's visible in the current frame, which causes problems when a robot disappears behind another robot or a field element.

A human scout handles this differently. If a robot disappears, the scout remembers where it was and watches nearby frames and other camera views to figure out where it probably went. I want to give the computer a similar ability.

Instead of treating every frame equally, the model would learn which observations are most useful for the current decision. For example, the current camera might have the robot mostly blocked while another camera shows it clearly at the same moment. Or the current frame might be unclear while a frame from a few moments earlier gives a better idea of the robot's movement.

This is where attention comes in. A model with attention can learn how much to trust each camera view at each moment, instead of relying on hand-picked weights. It could pick up cues that a fixed rule can't, such as "this camera's view of the robot is partly blocked" or "this observation from half a second ago is more trustworthy than the blurry current frame."

The goal is to combine information across time and across cameras so a robot doesn't "disap-

pear" from the system just because it's hidden for a few frames. That's the object permanence problem, the biggest one I ran into in this project.

Building this would take a much larger dataset of real matches with synchronized camera views and labeled robot positions, and it would have to handle robots that look very similar to each other. The first version would probably focus on one narrow problem, like recovering a robot's position after a short occlusion, before trying to solve all of robot tracking.

9 Conclusion

This season showed that fully automated scouting in FRC is still hard.

Ball tracking was often accurate enough to give real scouting value. Robot tracking was much less reliable, mainly because of occlusion, changing camera angles, and the difficulty of keeping the same robot identity over time.

The hybrid system showed that a scouting system can be useful without being fully autonomous. A human handled tracking the robots, which computer vision still struggled with, and the computer handled the repetitive parts like fuel tracking and shot analysis.

It produced more detailed robot-level data than rankings, averages, or public aggregate statistics alone, and the data was usable in actual competition.

For teams building something similar, I'd start by automating one part of scouting well instead of trying to automate everything at once. Assisted scouting is already practical, especially when computer vision, human review, multi-camera information, and a strong scouting application work together.

Software and Data Availability

Castle, Team 334's open-source scouting application used to view and work with the data this system produces, is available at `https://github.com/Team334/Castle` (see also `https://castlescouting.com`).

I plan to release the video-processing pipeline from this paper publicly, including the Gradio app that implements the ball- and robot-tracking methods in Sections 4–6: `https://github.com/cherriae/ai-scouting`.

References

[1] Meta AI. *SAM 3: Segment Anything with Concepts.* Released November 19, 2025. `https://ai.meta.com/blog/segment-anything-model-3/`

[2] Robinson, I., Robicheaux, P., Popov, F., Ramanan, D., & Peri, N. (2026). *RF-DETR: Real-Time Detection Transformer.* International Conference on Learning Representations (ICLR 2026). https://arxiv.org/abs/2511.09554

[3] Google DeepMind. *Introducing Gemini 3 Flash.* 2026. https://blog.google/products/gemini/gemini-3-flash/

[4] Qwen Team, Alibaba Cloud. *Qwen3-VL.* GitHub repository. https://github.com/QwenLM/Qwen3-VL

[5] Statbotics. FRC statistics and Expected Points Added (EPA) estimates. https://www.statbotics.io/

[6] Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). *You Only Look Once: Unified, Real-Time Object Detection.* IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

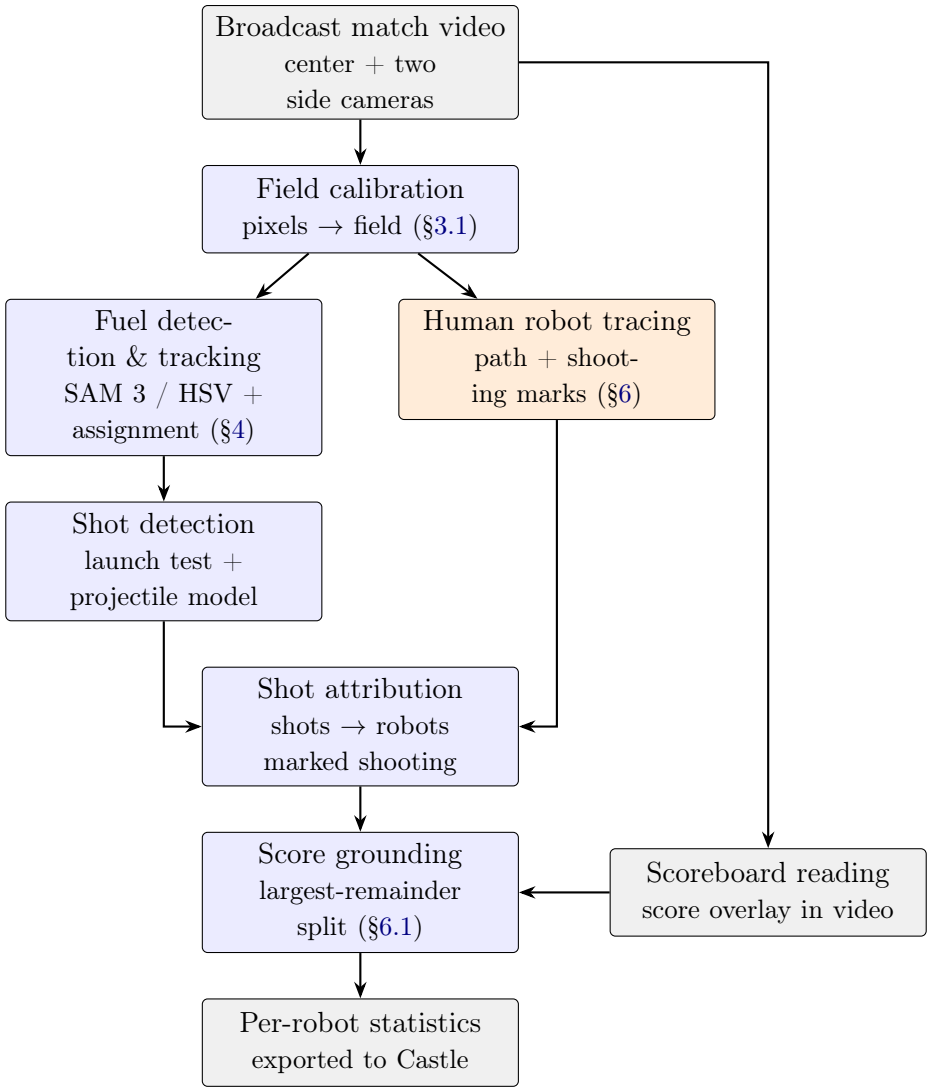


Figure 1: The deployed pipeline. Blue stages are automated, the orange stage is done by a human scout, and gray boxes are inputs and outputs. The scoreboard provides an independent count of made shots, which is used to correct the per-robot totals.

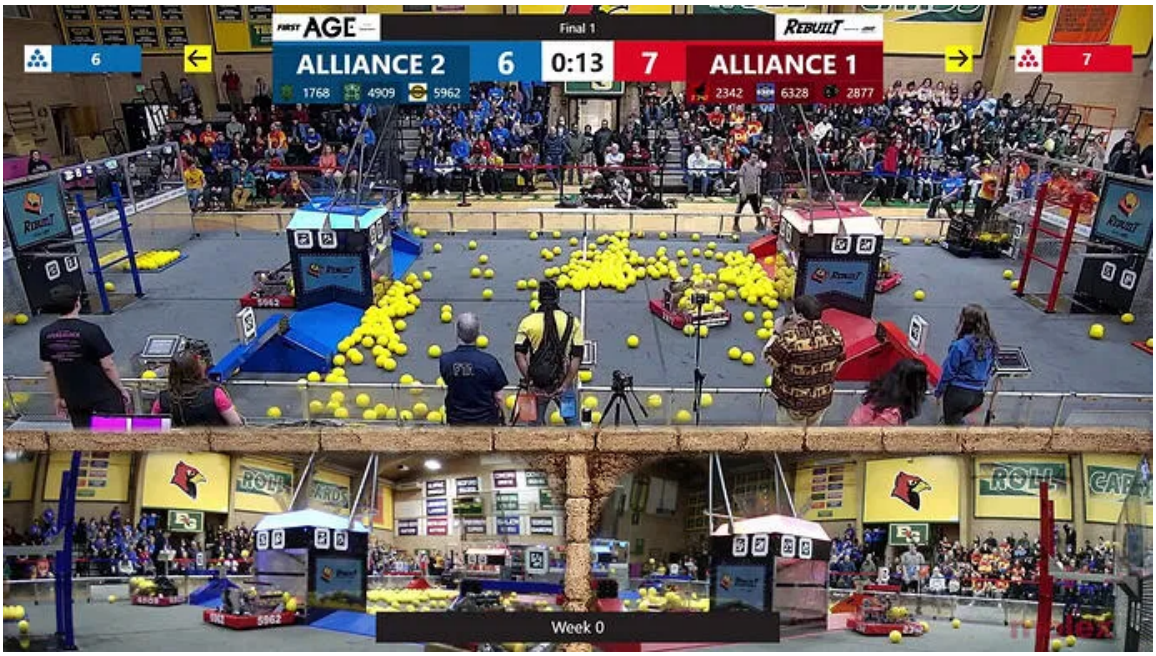


Figure 2: A composite broadcast frame used as system input. The center camera (top) contains the score and timer overlay and shows fuel spread across the field. The two side-camera views (bottom) are narrower and are separated by an event banner, here showing "Week 0." The banner and the changing camera placement made side-camera coverage inconsistent between events.