Module 7: Applied Computer Vision with OpenCV & YOLO

Capstone: From Pixels to Dashboard — A Real-Time Vehicle Analytics System in 20 Minutes

Overview

This capstone proves the four claims that anchor Module 7 — and it does it on real, ugly, real-world traffic footage:

- An image is just a tensor — and the same preprocessing tricks from Lecture 7.2 (resize, CLAHE for low light) turn marginal footage into something a model can actually work with, in less than a millisecond per frame.
- Modern object detection is three lines of Python — yolo, model.track, read xyxy/conf/cls — and it runs on a CPU at a frame rate good enough for most traffic-counting use cases.
- Tracking turns detection into a system — persistent IDs from ByteTrack are what let us count, not just see. The single argument persist=True is the difference between a demo and a product.
- The pipeline output is data, not video — a CSV log feeding a self-contained HTML dashboard is the artifact a real business would consume, and we will build it live.

You will start with a deliberately mixed-quality traffic clip, preprocess every frame, run YOLOv8 with ByteTrack, count vehicles by class as they cross a virtual lane line, write a CSV log AND a JSON snapshot of running totals, and open a working HTML dashboard in the browser — all inside one 20-minute live session.

Before You Start

- Have Python 3.10+ installed. Verify with python --version.
- Install the three libraries the entire pipeline needs: pip install ultralytics opencv-python numpy.

— saves 10 seconds during the live demo.

- Pre-clone the course repo with vehicle_analytics.py at the project root. The full source is reproduced at the end of this document for reference.
- Have a terminal open in the project directory, a browser open to a blank tab, and your AI chat panel open to a fresh conversation. We will use all three.
- Optional: have a personal traffic clip ready. The sample is fine, but the room reacts more when the input is local — a clip from a phone of the parking lot outside lands harder than a stock video.

During Hands-On — Four Segments at a Glance

- SEGMENT 1 — Capture and preprocess the video, run a baseline detection, read the contract — ~5 min
- SEGMENT 2 — Add tracking, add the counting line, watch IDs persist across frames — ~5 min
- SEGMENT 3 — Wire up the CSV log and the JSON snapshot — turn the demo into a data product — ~5 min
- SEGMENT 4 — Open the HTML dashboard, prove it auto-refreshes, demo a live failure-mode debug with AI — ~5 min

Remember Module 7 is not about writing fancy computer vision. It is about taking the same five-stage pipeline — capture, preprocess, detect, track, analyze — and delivering it as a system that emits data a business can use. Every line in this demo exists for one reason: to drop the cost of building a deployment-grade CV product from weeks to minutes. By the end of these 20 minutes you will not just have detected vehicles — you will have demonstrated the entire AI-native CV workflow that the rest of the students will use for the rest of their careers.

Segment 1 — Capture, Preprocess, Detect

What to Learn Before you train anything, before you tune anything, you have to capture and preprocess. Half of real-world CV bugs trace back to a frame the engineer never actually looked at. We open the source, resize, equalize if needed, and only then call YOLO. That order is non-negotiable — it is the discipline that separates engineers who ship CV pipelines from ones who debug them on Friday nights.

Step 1. Run the script and watch the baseline.

Open a terminal in the demo folder. Run the script with no arguments — it will download the sample traffic clip on first run, ~10 MB, and start detection immediately:

What you should get A playback window with a yellow horizontal line across the lower-middle of the frame, green boxes on every car/truck/bus/motorcycle with a track ID label, and a black HUD in the top-left showing IN total, OUT total, and live FPS. Let it run for ten seconds. The room sees the entire payoff of the module compressed into one window. That ten seconds is the visual hook — everything that follows is unpacking what just happened.

```
python vehicle_analytics.py
```

Step 2. Read the preprocess function.

Stop the script and open vehicle_analytics.py. Find the preprocess function near the top. Read it out loud:

```
def preprocess(frame, target_width: int = 960):
    if frame.shape[1] != target_width:
```

```
        scale = target_width / frame.shape[1]
        frame = cv2.resize(frame, (target_width, int(frame.shape[0] * scale)))
    if frame.mean() < 70:
        hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
        h_, s_, v_ = cv2.split(hsv)
        v_ = cv2.createCLAHE(2.0, (8,8)).apply(v_)
        frame = cv2.cvtColor(cv2.merge([h_, s_, v_]), cv2.COLOR_HSV2BGR)
    return frame
```

The summary is Eight lines, two conditional branches. The first resizes to 960 pixels wide — small enough to run fast, big enough that vehicles are still detectable. The second checks if the frame is dark on average; if so, it equalizes the V channel of HSV using CLAHE. Tell the room: 'CLAHE is the single most useful low-light fix in OpenCV. One line, no parameters to tune for most footage, and it turns dawn or dusk video from useless to usable.' This is the only preprocessing you usually need — anything fancier belongs in a training pipeline, not a runtime one.

Step 3. The detection contract — read it once, read it again.
Scroll down to the YOLO call. Five arguments, every one earning its keep:

Remember These five arguments are the entire vocabulary of production YOLO tracking. persist=True is the one that turns boxes into trajectories. classes=[...] is the cheapest possible domain filter — much cheaper than fine-tuning. conf is the precision/recall knob. tracker= picks the algorithm. verbose=False is the difference between a demo terminal and a debugging terminal. Memorize this block.

```
results = model.track(
    frame,
    persist=True,                           # carry track IDs across frames
    classes=list(VEHICLE_CLASSES.keys()),   # car, motorcycle, bus, truck
    conf=args.conf,                         # threshold — tunable from CLI
    tracker='bytetrack.yaml',               # built into Ultralytics
    verbose=False,                          # silence per-frame stdout
)
```

Segment 2 — Tracking + The Counting Line

What to Learn Detection alone says who is in this frame. Tracking says who THIS car is across frames. Counting needs the second one. We demonstrate persistent IDs visually, show the foot-point trick from Lecture 7.4, and then move the line live to prove that lane configuration is a deployment decision, not a code change.

Step 1. Watch the IDs persist.

Re-run the script and focus on a single car as it enters and crosses:

```
python vehicle_analytics.py --line 0.55
```

What you should get Watch one specific track ID. It stays glued to the same vehicle as long as that vehicle is visible. When the car leaves the frame and a new one enters, the ID increments — not because the model is smarter, but because ByteTrack assigns the next available number. Tell the room: 'IDs are unique-within-a-session, not globally unique. If you restart the script, IDs reset. For long-running deployments, persist them yourself in a database.'

Step 2. Move the line.

Stop the script. Re-run with a different line position, three times in a row, and watch the counts diverge:

```
python vehicle_analytics.py --line 0.30   # line near the top — catches entries
python vehicle_analytics.py --line 0.55   # default
python vehicle_analytics.py --line 0.80   # near the bottom — catches exits
```

What you should get Three different IN/OUT totals on the same clip. Frame the takeaway for the room: 'Line placement is camera-mount-specific. A real deployment doesn't ship one number — it ships a configuration UI that lets the installer drag a line onto the video. We just demonstrated that the underlying engine doesn't care where the line is. It is a parameter, not a hard-coded rule.'

Step 3. Show the crossing logic.

Open vehicle_analytics.py and find the crossing block. Read every line:

What you should get Three guards in eight lines. The previous foot-y has to exist. The direction has to be consistent. The track ID has to be in the right counted-set. That last guard — 'and int(tid) not in counted_in' — is the entire reason a car that stops on the line doesn't get counted ten times. Set-based counting is the production trick. Make sure the room sees the line and asks the question: 'what happens if you remove that guard?' The answer is hilariously bad — and that's the lesson.

```
prev_y = track_last_y.get(int(tid))
if prev_y is not None:
    crossed_down = prev_y < line_y <= fy and int(tid) not in counted_in
    crossed_up   = prev_y > line_y >= fy and int(tid) not in counted_out
    if crossed_down:
        counted_in.add(int(tid))
        counts_in_by_class[cls_name] += 1
track_last_y[int(tid)] = fy
```

Segment 3 — Two Outputs That Make This a Product

What to Learn A demo emits a window. A product emits data. We are going to wire up two parallel outputs: a CSV of every crossing event (high-fidelity, every row is a fact) and a JSON snapshot of running totals (low-frequency, refreshed every 5 seconds). Different consumers need different shapes — that is the production lesson.

Step 1. Open the CSV while the script runs.

Start the script in one terminal. Open a second terminal and tail the CSV as crossings happen:

```
Linux / Unix
tail -f capstone_output/crossings.csv

Windows PowerShell
Get-Content capstone_output\crossings.csv -Wait -Tail 10
```

Run: `python vehicle_analytics.py`

What you should get Every time a vehicle crosses, a new row appears: timestamp, track_id, vehicle_class, direction, line_y. The room sees the data emerging from the pipeline in real time. Tell them: 'this CSV is your audit log. Every business stakeholder eventually asks WHEN did vehicle X happen — that question only has an answer because we are writing this file right now.'

Step 2. Open the JSON snapshot.

In the same second terminal:

```
Linux / Unix
watch -n 1 cat capstone_output/snapshot.json

Windows PowerShell
while ($true) { Clear-Host; Get-Content capstone_output\snapshot.json -Raw | ConvertFrom-Json | ConvertTo-Json -Depth 10; Start-Sleep -Seconds 1 }

Run: Run: python vehicle_analytics.py
```

What you should get Refreshing every second, you see the snapshot file rewrite itself every 5 seconds of pipeline time. It shows running totals broken down by direction and by class. Tell the room: 'CSV is for analytics. JSON snapshot is for monitoring. The same pipeline serves both with no extra compute cost — just two writers tapping the same in-memory state.'

Step 3. Why two outputs, not one.

Remember A junior engineer would write a single output and let the consumers query it. That works until you have 200,000 crossings and a dashboard that polls every 2 seconds — at which point you start scanning a giant CSV every poll and your CPU melts. Two outputs, two cadences, two consumers. The CSV is append-only history. The JSON snapshot is constant-size current state. This pattern — log + snapshot — is the universal shape of every real-time analytics system you will ever build.

Step 4. Test the headless mode.

Stop the script. Run it with --no-display so the pipeline runs faster (no GUI):

What you should get The terminal still emits FPS-per-second-ish output, the CSV still grows, the JSON snapshot still updates, and the annotated MP4 still saves at the end. No window opens. This is the mode you actually deploy. Display windows are for demos and debugging — the real pipeline runs headless on a Raspberry Pi in a closet somewhere, emitting data to a queue.

```
python vehicle_analytics.py --no-display
```

Segment 4 — The Dashboard and the Live Debug

What to Learn Data without presentation is invisible to the stakeholders who fund this work. A self-contained HTML dashboard is the fastest path from CSV to executive screen. We open it, prove it auto-refreshes, intentionally break the pipeline, and use AI to debug live from the error log — exactly like the Module 4 capstone closing move, but on CV instead of CI.

Step 1. Open the dashboard.

After the first run, the script wrote dashboard.html into the output folder. Open it from the same folder as crossings.csv (so the fetch() can find the CSV):

```
cd capstone_output
python -m http.server 8080
# then open http://localhost:8080/dashboard.html in your browser
```

What you should get A clean dashboard with summary cards (total events, IN, OUT, per-class breakdowns) and a table of recent crossings. The page reloads the CSV every 2 seconds, so if you re-run the pipeline in another terminal the dashboard updates live. Show that update. The room understands instantly: 'this is what a real product looks like — pipeline emits data, browser reads data, both update on their own schedule.'

Step 2. Live debug — break it on purpose.

Open vehicle_analytics.py. Change line_y = int(fh * args.line) to line_y = int(fh * args.line * 100) — an obviously wrong scaling. Run the script. The line disappears off the bottom of the frame, IN/OUT counts stay at 0, the dashboard goes silent. The room watches the failure. Now the demo move — copy the script and the symptom into your AI chat:

Open vehicle_analytics.py. Change line_y = int(fh * args.line) to line_y = int(fh * args.line * 100) — an obviously wrong scaling. Run the script. The line disappears off the bottom of the frame, IN/OUT counts stay at 0, the dashboard goes silent. The room watches the failure. Now the demo move — copy the script and the symptom into your AI chat:

```
I have a Module 7 vehicle counter that suddenly stopped counting.
The IN and OUT totals stay at zero. The counting line is not visible
in the playback window. Here is the line that computes the line
position from the --line argument:

    line_y = int(fh * args.line * 100)

args.line defaults to 0.55 and fh is the frame height (about 540).
Why is the line off-screen and how do I fix it?
```

What you should get Within seconds, AI returns: 'line_y is 540 * 0.55 * 100 = 29,700 — far below the frame. Remove the * 100; args.line is already a fraction (0.0–1.0). Replace with int(fh * args.line).' Apply the fix, re-run, line reappears, counts climb. Total time from broken to fixed: under a minute. This is the workflow the room will reach for the next time a CV pipeline breaks at 11pm.

Step 3. Document the pipeline.

Pipelines are infrastructure and infrastructure deserves docs. One last prompt — paste into your AI chat:

What you should get A roughly 60-line README section. Paste it into the project README and commit. The next engineer who picks up this pipeline can be productive on day one. That delta — days saved per onboarding — is the kind of compounding return that makes the AI-native CV engineer worth what they get paid.

```
Generate a README section for vehicle_analytics.py that documents:
  - The five-stage pipeline (capture, preprocess, detect, track, analyze)
  - All CLI arguments and what they tune
  - The CSV schema (one row per crossing)
  - The JSON snapshot schema (constant-size, refreshed every 5 sec)
  - The three most likely runtime failures and their fixes:
      camera unavailable, slow FPS, line off-screen
```

Wrap-Up — The Pipeline Is Live

Remember In 20 minutes you took raw traffic footage, preprocessed every frame for size and lighting, ran YOLOv8 detection with ByteTrack persistent IDs, counted crossings by class using a set-based deduplication trick, emitted a CSV log and a JSON snapshot at two different cadences, opened a live HTML dashboard reading the same CSV, broke the pipeline on purpose, and debugged it back to green in under a minute using AI. That is not a toy. That is the actual shape of a deployed real-time analytics system your students will adapt to retail traffic, transit ridership, classroom attendance, or any other 'count things crossing a line' problem starting Monday morning.

Key takeaways:

- Preprocess before you predict. Resize and CLAHE are 90% of the runtime preprocessing you will ever need.
- Detection alone is not enough. Tracking — model.track(persist=True) with ByteTrack — is what turns boxes into a system.
- Use the foot-point, not the box center. The bottom-center is glued to the ground plane. The center bobs around.

- Set-based counting is the production trick. counted_in.add(tid) prevents a stationary vehicle from being counted ten times.
- Emit two outputs at two cadences. CSV log for history; JSON snapshot for monitoring. Same in-memory state, two writers.
- A green pipeline is one prompt away. And debugging a red one is under a minute when you paste the symptom and the suspect line into AI with the right framing.

Student Lab — Module 7 Capstone Submission Take the vehicle_analytics.py pipeline from this demo and extend it on a video of your own choosing. You must do all five of the following: (1) record or download a real traffic clip of at least 60 seconds, (2) tune --line and --conf for that clip and document why you chose the values you did, (3) add a second counting line and modify the script to emit per-lane counts to the CSV, (4) extend dashboard.html with one additional chart of your choosing (per-class breakdown, hourly histogram, or dwell time), and (5) write a 3-5 minute video walkthrough explaining one engineering decision you made and the failure mode you saw it prevent. Submit five things: the GitHub repo link, the modified script, the extended dashboard, the CSV output of one full run, and the video. This is the gate to Module 8.

Appendix — vehicle_analytics.py (the script used in this demo)

This is the ~200-line Python script that drives the entire 20-minute demo. Save it as vehicle_analytics.py at the root of the demo repo before class begins. The pipeline is the production shape; the demo just runs it on a small sample. The five stages — capture, preprocess, detect, track, analyze — are explicitly named in the comments so students can map them to the lecture slides at a glance.

```
"""
vehicle_analytics.py
--------------------
Module 7 · Capstone · Real-Time Vehicle Analytics

The synthesis of the entire module. One ~200-line script that:
  1. Reads a traffic video (OpenCV)
  2. Preprocesses each frame (Module 7.2: resize + optional CLAHE for low light)
  3. Runs YOLO detection (Module 7.3) with class filtering for vehicles
  4. Adds ByteTrack tracking (Module 7.4) for persistent IDs
  5. Counts each lane crossing by class (car, truck, bus, motorcycle)
  6. Writes a per-event CSV log AND a JSON snapshot every 5 seconds
  7. Generates a simple self-contained HTML dashboard from the CSV

Usage:
    python vehicle_analytics.py
    python vehicle_analytics.py path/to/traffic.mp4 --line 0.55
    python vehicle_analytics.py path/to/traffic.mp4 --no-display    # headless

Outputs (created in --outdir, default ./capstone_output):
    annotated_traffic.mp4   — video with boxes + lane line + live counts
    crossings.csv           — one row per vehicle crossing event
    snapshot.json           — running totals, refreshed every 5 seconds
    dashboard.html          — static HTML reading the CSV in the browser
"""
```

```
import os
import csv
import json
import time
import argparse
import urllib.request
from datetime import datetime, timezone
from pathlib import Path
from collections import defaultdict

import cv2
from ultralytics import YOLO


SAMPLE_URL = ("https://github.com/intel-iot-devkit/sample-videos/raw/master/"
              "car-detection.mp4")
SAMPLE_PATH = "sample_traffic.mp4"

# COCO class ids we care about for traffic analytics
VEHICLE_CLASSES = {
    2: "car",
    3: "motorcycle",
    5: "bus",
    7: "truck",
}


def ensure_sample(path: str = SAMPLE_PATH) -> str:
    if os.path.exists(path):
        return path
    print(f"Downloading sample traffic video -> {path}")
    urllib.request.urlretrieve(SAMPLE_URL, path)
    return path


def preprocess(frame, target_width: int = 960):
    """Resize and optionally apply CLAHE on the V-channel for low-light frames."""
    h, w = frame.shape[:2]
    if w != target_width:
        scale = target_width / w
        frame = cv2.resize(frame, (target_width, int(h * scale)))
    # Cheap low-light check: if the frame is dark, equalize luminance.
    if frame.mean() < 70:
        hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
        h_, s_, v_ = cv2.split(hsv)
        clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
        v_ = clahe.apply(v_)
        frame = cv2.cvtColor(cv2.merge([h_, s_, v_]), cv2.COLOR_HSV2BGR)
    return frame


def foot_point(x1, y1, x2, y2):
```

```
    return int((x1 + x2) / 2.0), int(y2)


def main():
    ap = argparse.ArgumentParser(description=__doc__,
                                 formatter_class=argparse.RawDescriptionHelpFormatter)
    ap.add_argument("source", nargs="?", default=None,
                    help="Path to a traffic video. Webcam: pass 0.")
    ap.add_argument("--line", type=float, default=0.55,
                    help="Counting line as a fraction of frame height (0.0-1.0).")
    ap.add_argument("--conf", type=float, default=0.35)
    ap.add_argument("--model", default="yolov8n.pt")
    ap.add_argument("--outdir", default="./capstone_output")
    ap.add_argument("--no-display", action="store_true",
                    help="Run headless (skip the cv2.imshow window).")
    args = ap.parse_args()

    src = args.source if args.source is not None else ensure_sample()
    if src == "0":
        src = 0  # webcam

    outdir = Path(args.outdir)
    outdir.mkdir(parents=True, exist_ok=True)

    cap = cv2.VideoCapture(src)
    if not cap.isOpened():
        raise SystemExit(f"Could not open video source: {src}")

    fps = cap.get(cv2.CAP_PROP_FPS) or 25.0
    width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
    height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
    # Output uses preprocessed width (960). Recompute output height after one frame.
    fourcc = cv2.VideoWriter_fourcc(*"mp4v")
    writer = None  # initialized once we see the first preprocessed frame

    model = YOLO(args.model)

    # state -----------------------------------------------------------------
    track_last_y: dict[int, int] = {}     # last foot-y per track id
    track_class: dict[int, str] = {}      # remember class so we still know it after the cross
    counted_in: set[int] = set()          # track ids already counted IN
    counted_out: set[int] = set()         # track ids already counted OUT
    counts_in_by_class = defaultdict(int)
    counts_out_by_class = defaultdict(int)

    # logging ---------------------------------------------------------------
    csv_path = outdir / "crossings.csv"
    csv_fh = open(csv_path, "w", newline="")
    csv_writer = csv.writer(csv_fh)
    csv_writer.writerow(["timestamp", "track_id", "vehicle_class", "direction", "line_y"])

    snapshot_path = outdir / "snapshot.json"
    last_snapshot_ts = 0.0
```

```
print(f"Source: {src}")
print(f"Model: {args.model}    conf={args.conf}    line={args.line}")
print(f"Outputs going to: {outdir.resolve()}")
print("Press 'q' to quit early (display mode only).")

frame_idx = 0
t0 = time.time()
while True:
    ok, raw = cap.read()
    if not ok:
        break
    frame_idx += 1
    frame = preprocess(raw)
    fh, fw = frame.shape[:2]
    line_y = int(fh * args.line)

    # initialize writer once we know the preprocessed dimensions
    if writer is None:
        writer = cv2.VideoWriter(
            str(outdir / "annotated_traffic.mp4"),
            fourcc, fps, (fw, fh),
        )

    # YOLO + ByteTrack
    results = model.track(
        frame,
        persist=True,
        classes=list(VEHICLE_CLASSES.keys()),
        conf=args.conf,
        tracker="bytetrack.yaml",
        verbose=False,
    )
    result = results[0]

    # draw line
    cv2.line(frame, (0, line_y), (fw, line_y), (0, 220, 220), 2)
    cv2.putText(frame, f"line y={line_y}", (10, line_y - 8),
                cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 220, 220), 1)

    # process detections
    if result.boxes is not None and result.boxes.id is not None:
        xyxy_t = result.boxes.xyxy.cpu().numpy()
        conf_t = result.boxes.conf.cpu().numpy()
        cls_t = result.boxes.cls.cpu().numpy().astype(int)
        ids_t = result.boxes.id.cpu().numpy().astype(int)

        for (x1, y1, x2, y2), c, k, tid in zip(xyxy_t, conf_t, cls_t, ids_t):
            cls_name = VEHICLE_CLASSES.get(int(k), "vehicle")
            track_class[int(tid)] = cls_name
            fx, fy = foot_point(x1, y1, x2, y2)

            # draw box
```

```
                cv2.rectangle(frame, (int(x1), int(y1)), (int(x2), int(y2)),
                              (0, 200, 0), 2)
                label = f"ID {tid} {cls_name} {c:.2f}"
                cv2.putText(frame, label, (int(x1), int(y1) - 6),
                            cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 200, 0), 1)
                cv2.circle(frame, (fx, fy), 4, (0, 0, 255), -1)

                # crossing logic
                prev_y = track_last_y.get(int(tid))
                if prev_y is not None:
                    crossed_down = prev_y < line_y <= fy and int(tid) not in counted_in
                    crossed_up = prev_y > line_y >= fy and int(tid) not in counted_out
                    if crossed_down:
                        counted_in.add(int(tid))
                        counts_in_by_class[cls_name] += 1
                        csv_writer.writerow([
                            datetime.now(timezone.utc).isoformat(),
                            int(tid), cls_name, "IN", line_y,
                        ])
                        csv_fh.flush()
                    if crossed_up:
                        counted_out.add(int(tid))
                        counts_out_by_class[cls_name] += 1
                        csv_writer.writerow([
                            datetime.now(timezone.utc).isoformat(),
                            int(tid), cls_name, "OUT", line_y,
                        ])
                        csv_fh.flush()
                track_last_y[int(tid)] = fy

        # HUD overlay
        total_in = sum(counts_in_by_class.values())
        total_out = sum(counts_out_by_class.values())
        cv2.rectangle(frame, (0, 0), (260, 110), (0, 0, 0), -1)
        cv2.putText(frame, f"IN  total: {total_in}", (12, 26),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0), 2)
        cv2.putText(frame, f"OUT total: {total_out}", (12, 56),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 255), 2)
        elapsed = time.time() - t0
        live_fps = frame_idx / elapsed if elapsed > 0 else 0
        cv2.putText(frame, f"FPS: {live_fps:.1f}", (12, 86),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255, 255), 1)

        writer.write(frame)
        if not args.no_display:
            cv2.imshow("Vehicle Analytics — Module 7 Capstone", frame)
            if cv2.waitKey(1) & 0xFF == ord("q"):
                break

        # periodic JSON snapshot (every 5 seconds of wall time)
        now = time.time()
        if now - last_snapshot_ts >= 5.0:
            snap = {
```

```
                "updated_at": datetime.now(timezone.utc).isoformat(),
                "elapsed_seconds": round(elapsed, 2),
                "fps": round(live_fps, 2),
                "in_total": total_in,
                "out_total": total_out,
                "in_by_class": dict(counts_in_by_class),
                "out_by_class": dict(counts_out_by_class),
                "active_tracks": len(track_last_y),
            }
            snapshot_path.write_text(json.dumps(snap, indent=2))
            last_snapshot_ts = now

    cap.release()
    if writer is not None:
        writer.release()
    csv_fh.close()
    cv2.destroyAllWindows()

    # write a self-contained HTML dashboard that reads crossings.csv in the browser
    dash_path = outdir / "dashboard.html"
    dash_path.write_text(DASHBOARD_HTML)
    print(f"\nDone. Wrote:\n  {outdir/'annotated_traffic.mp4'}\n  {csv_path}\n  {snapshot_path}\n
    print(f"Open dashboard.html in a browser from the same folder as crossings.csv.")


DASHBOARD_HTML = """<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8" />
<title>Vehicle Analytics Dashboard</title>
<style>
  body { font-family: -apple-system, Segoe UI, Arial, sans-serif; margin: 24px; color: #1f2a44; }
  h1   { margin: 0 0 4px 0; }
  .sub { color: #666; margin-bottom: 24px; }
  .row { display: flex; gap: 16px; flex-wrap: wrap; }
  .card { flex: 1 1 220px; background: #f4f7fb; border: 1px solid #d6dde6;
          border-radius: 8px; padding: 14px 18px; }
  .card .num { font-size: 30px; font-weight: 700; }
  .card .lbl { font-size: 13px; color: #555; text-transform: uppercase; letter-spacing: 0.04em; }
  table { width: 100%; border-collapse: collapse; margin-top: 20px; }
  th, td { padding: 8px 10px; border-bottom: 1px solid #e0e6ee; text-align: left; font-size: 14px
  th { background: #eaf1f8; }
  .pill { display: inline-block; padding: 2px 8px; border-radius: 10px; font-size: 12px; font-wei
  .IN  { background: #d8f3dc; color: #1b5e20; }
  .OUT { background: #fde0e0; color: #8b1a1a; }
  #status { font-size: 13px; color: #888; }
</style>
</head>
<body>
  <h1>Vehicle Analytics Dashboard</h1>
  <div class="sub">Module 7 Capstone &mdash; reads <code>crossings.csv</code> from this folder.</
  <div id="status">Loading...</div>
  <div class="row" id="cards"></div>
```

```
  <h3>Recent events</h3>
  <table
id="events"><thead><tr><th>Time</th><th>Track</th><th>Class</th><th>Direction</th></tr></thead><t

<script>
async function load() {
  try {
    const r = await fetch('crossings.csv?_=' + Date.now());
    if (!r.ok) throw new Error('No crossings.csv yet.');
    const text = await r.text();
    const lines = text.trim().split('\\n').slice(1); // strip header
    const rows = lines.map(l => l.split(','));
    const byClass = {}, byDir = {IN: 0, OUT: 0};
    for (const row of rows) {
      const [ts, id, cls, dir] = row;
      byClass[cls] = (byClass[cls] || 0) + 1;
      byDir[dir]   = (byDir[dir]   || 0) + 1;
    }
    const cards = document.getElementById('cards');
    cards.innerHTML = '';
    function card(num, lbl) {
      return `<div class="card"><div class="num">${num}</div><div class="lbl">${lbl}</div></div>`
    }
    cards.insertAdjacentHTML('beforeend', card(rows.length, 'total events'));
    cards.insertAdjacentHTML('beforeend', card(byDir.IN  || 0, 'in'));
    cards.insertAdjacentHTML('beforeend', card(byDir.OUT || 0, 'out'));
    for (const c of Object.keys(byClass)) {
      cards.insertAdjacentHTML('beforeend', card(byClass[c], c));
    }
    const tbody = document.querySelector('#events tbody');
    tbody.innerHTML = '';
    for (const row of rows.slice(-25).reverse()) {
      const [ts, id, cls, dir] = row;
      tbody.insertAdjacentHTML('beforeend',
        `<tr><td>${ts}</td><td>${id}</td><td>${cls}</td><td><span class="pill ${dir}">${dir}</spa
    }
    document.getElementById('status').textContent = 'Updated ' + new Date().toLocaleTimeString();
  } catch (err) {
    document.getElementById('status').textContent = err.message;
  }
}
load();
setInterval(load, 2000);
</script>
</body>
</html>
"""


if __name__ == "__main__":
    main()
```