Audion Traffic Counter — a guide for an AI agent

Contents

- Rules that always hold
- The folder
- Making the folder work
- Commands
- Placing the lines by frames
- Counting and reading a run
- Teaching the kinds for the user's cameras
- Teaching the detector for a view it does not see
- When something goes wrong

This file is for an AI agent that works in the folder of the program together with its user: Claude Code, Codex or any other agent that can run commands and look at pictures. The user starts it in the program's folder and says: "Read `Docs\AI_AGENT_GUIDE_EN.md`". Everything the window of the program does goes through the engine commands described here, so the agent can do the same work from the command line.

What the program does: it reads a recording of a traffic camera, finds the vehicles in every frame, follows each one from frame to frame and counts it once when it crosses a line drawn on the frame, along the line's arrow. The result is how many cars, trucks, buses and motorcycles made each movement. The lines of a recording live in a camera file; every recording has its own.

People use the program through its window, `Start.exe`, and the user guide (`Docs\USER_GUIDE_EN.md`). Paths below are relative to the program's folder.

Rules that always hold

- **The recordings are the user's.** They stay where the user keeps them: never copy them into the program's folder, never send them anywhere.
- **Ask before you look.** Whatever you look at - a frame to place the lines by, a sheet of vehicle pictures to label - goes to the service that runs you. Ask the user whether that is allowed for these recordings before the first picture. If it is not, the user labels and checks the pictures; you run the commands.
- **Run the engine only with `runtime\python.exe`.** The system Python has none of the packages and must not get them.
- **Do not start the count of a queue of recordings.** That is the user's COUNT button. One recording, or a part of one, from the command line is fine - say what it is for and how many minutes it takes before starting anything that loads the video card.
- **Nothing is written over.** A model, a set, a batch of sheets keep their names; a new one takes a new name. The runs in `output\` and the training data are the user's.
- **Close the window** before `recordings add`: the window keeps the list too.
- **A number goes with what it stands on**: "buses -6 %" means nothing without "of 51 buses". One vehicle among twenty is five percent.

The folder

Path	What is in it
`Start.exe`, `App\`	the window
`runtime\`	the program's own Python with its libraries (installed by the window)
`models\`	detector weights and what is made from them; `models\classes\` - models of vehicle kinds taught here
`system_core\kinds\`	the model of vehicle kinds that comes with the program (the default one)
`Tools\ffmpeg\bin`	FFmpeg
`system_core\main.py`	the engine: every command below
`config\cameras\<id>.json`	the lines of each recording (camera files)

Path	What is in it
`config\recordings.json`	the list of recordings of the window
`output\<recording>\<date-time>_<model>\`	a run: `events.csv`, `summary.json`, `camera.json`, with pictures `crops\`, `crops.csv`, marks `review.csv`
`classes\` beside the recordings	the training data: pictures, sheets, labels, sets of teaching the kinds (SETTINGS, TRAINING DATA can put them in a folder of the user's choice instead)
`reference\<camera>\<recording>.json`	manual counts to compare with
`logs\`	the journal of the window

The work folder. `config\cameras`, `config\recordings.json`, `output`, `reference` and `logs` are the user's own material. They lie in the folder of the program by default, and in the folder the user chose when SETTINGS, WORK FOLDER says IN THE USER'S FOLDER (`"WorkFolder"` of `config\settings.json`; the settings themselves always stay with the program). The layout is the same in both places, and the engine follows the choice by itself: `recordings list` prints the full path of every lines file, `count` without `--out` writes into `output` of the work folder, `report` reads the runs from there. So before writing a lines file or looking for a run, read where they are from `recordings list` - do not assume the folder of the program. Never move the user's work folder or change this setting unasked.

Making the folder work

The program may be a bare skeleton: the window, the code, the configs, nothing installed. `Start.exe`, section SETUP, **INSTALL WHAT IS NEEDED** installs what this machine lacks (Python, PyTorch for the card, TensorRT on an NVIDIA RTX card, the detector weights, FFmpeg), about six minutes and 6 GB of downloads. The button asks first, saying what comes for this machine. The first step makes sure Windows has the Microsoft Visual C++ Redistributable PyTorch needs and installs it with winget when it has not (`install\Ensure-VCRedist.ps1`; the one thing that goes into Windows, with the administrator's consent). From the command line: `builder_main.cmd install_all`. Then:

```
builder_main.cmd verify
runtime\python.exe system_core\main.py doctor
```

`verify` must end with "Everything is in place"; `backend` in the answer of `doctor` is what this machine counts on: `tensorrt` (NVIDIA RTX, the fast path), `cuda` (other NVIDIA cards), `directml` (any DirectX 12 card - Intel, AMD - several times slower), `cpu`.

Commands

All are `runtime\python.exe system_core\main.py <command>`; every line printed is one JSON object.

```
doctor                                  what is installed, what the count runs on
recordings add FOLDER_OR_FILE [...]     add videos to the window's list (window
closed)
recordings list                         every recording: id, size, lines file, placed
or not, what is wrong
frames [--id ID ...] [--at 0.5]         the middle frame of each recording as
._runtime\frames\<id>.jpg,
                                        with a grid in frame pixels; <id>.lines.jpg -
with its lines drawn
count --video FILE --camera config\cameras\ID.json --out DIR
      [--start S] [--end S] [--model yolo26s.pt] [--device auto]
      [--crops 3] [--classes NAME] [--snapshot-every 10]
                                        count one recording (or a part); --crops
keeps pictures of
                                        every counted vehicle, --classes names the
kinds with a model
compare --summary RUN\summary.json [...] --reference reference\CAMERA\NAME.json
                                        a run (or several) against a manual count
classify --run DIR [--classes NAME]     the kinds of a counted run named anew,
without counting again
harvest --video FILE [--out DIR] [--start S] [--end S] [--grid 2x2]
                                        pictures of every vehicle that moved, no
lines needed
sheets --dir DIR --name NAME [--classes ...] [--sample N] [--lines ...]
                                        sheets of 48 numbered pictures to label by
looking
labels --dir DIR [...]                  the letters written beside the sheets ->
labels.csv
teach --name NAME --train DIR ... --val DIR ... [--epochs 25] [--boost PREFIX N]
      [--minibus bus|car] [--light truck|car]
                                        a model of vehicle kinds ->
models\classes\NAME.pt
check-classes --classes NAME --val DIR ... [--car-to-truck]
                                        the detector, the model and the rule "both"
against labels
report --out DIR [--id ID ...] [--run RUN ...] [--formats xlsx docx md] [--lang
ru|en]
       [--sheets truck bus motorcycle car]
                                        the counted recordings as files people read:
a Word document
                                        and an Excel sheet; without --id and --run -
the newest run of
```

```
                                        every recording of the list; --sheets -
numbered pictures of
                                        every vehicle of those kinds, per section
detector harvest | marks | frames | tiles | pack | check | eye | join | teach | exam
| ship
                                        teaching a detector for a view the general
one does not see:
                                        "Teaching the detector for a view it does not
see"
```

`--classes auto` takes the default model of kinds: the one `config\classes_default.json` names. The window does the same when nothing else is chosen in TRAINING. Its file comes with the program (`system_core\kinds\`), so a freshly installed folder counts with it; a model of the same name in `models\classes` is the user's own and is taken first. `doctor` says which model is the default (`"kinds"`). `--car-to-truck` on `count`, `classify` and `check-classes` is the rule of kinds as it was before 1.5.0, to repeat old numbers.

The report. When the user asks for the result, write it with `report` into the folder he names and tell him what it says: how many recordings, how many are marked "check" and why. A recording is marked when it was read not to its end, or when its kinds are doubtful - the model of kinds renamed at least 20 and at least 5 % of the vehicles the detector called cars (`cars`, `renamed`, `doubt` in the `classes` block of `summary.json`). That is the sign of a camera the model was not taught on: the number of vehicles of such a run is right, their division into cars, buses and trucks is not. The remedy is in "Teaching the kinds for the user's cameras"; do not hand such a division over as a result.

Placing the lines by frames

A good way to set up many recordings: the agent places the lines from one frame of each, the user walks through them in the window (CAMERA, the previous / next buttons), corrects with the mouse and presses COUNT.

1. Close the window. `main.py recordings add <folder>`. Each recording gets an `id` (the file name without its extension, numbered when names repeat).
2. `main.py frames`. Look at each `<id>.jpg`: a frame from the middle of the recording (the beginning is often dark or shaking), shown twice its size when narrow, with a grid whose numbers are frame pixels - read the coordinates straight off it.
3. For every recording write `config\cameras\<id>.json`:

```
{
  "name": "a name of the camera position, for people",
  "frame_size": [640, 480],
  "zones": [
    { "id": "near", "rect": [0, 0, 640, 480], "scale": 1 },
    { "id": "far",  "rect": [0, 196, 640, 292], "scale": 2 }
  ],
  "lines": [
    { "id": "south", "label": "what it counts", "zone": "near",
      "p1": [60, 300], "p2": [280, 300], "pos": "N1", "neg": "" },
    { "id": "gate-A", "label": "", "zone": "near", "p1": [360, 430], "p2": [600,
455],
      "pos": "A:in", "neg": "A:out", "kind": "gate", "branch": "A" }
  ]
}
```

- `frame_size` - the size `frames` printed; coordinates are pixels of the original frame, x from the left, y from the top.
- One line per flow the survey counts: across the carriageway at right angles to the travel, where vehicles are large and apart - not at a stop line, not behind a pole, not at the very edge of the frame (a track must be steady before it reaches the line).
- Direction: a line counts under `pos` the vehicles that end on the left of `p1 -> p2`. Drawn bottom to top it counts those going right; top to bottom - going left; right to left - going up the frame; left to right - going down. Leave `neg` empty unless the opposite flow is counted on the same line too; then `pos` is the larger flow.
- A line before a fork whose branches have lines of their own is a check line: `"check": true`, otherwise its vehicles stand in the total twice.
- Names: the movement names of the user's survey sheet when known, else `N1`, `N2`...
- An intersection or a roundabout: a gate across every branch, over its entry and exit lanes together (`"kind": "gate"`, `"branch": "A"`, `pos` / `neg` = `A:in` / `A:out`); the engine pairs entries and exits into manoeuvres "A → B".
- A far part where vehicles are a few pixels: a zone around it with `scale` 2 and the far lines in that zone. No zones where the frame does not need them.

4. `main.py frames` again. For every recording `problem` must be `null` and `placed` true. Look at every `<id>.lines.jpg`: each line lies across its road, the arrow points the way the counted vehicles go. Correct and repeat.

5. Recordings shot from one spot share a placement: copy the file under the other id and check the copy on its own frame - a tripod gets moved.

6. Tell the user what was placed and what was not, with the paths of the `.lines.jpg` pictures. What a frame cannot tell - which way a road goes when it holds no vehicle, which movements the user's sheet names - ask, do not guess: a line with the arrow the wrong way counts nothing all night.

Counting and reading a run

One recording for a check:

```
main.py count --video <file> --camera config\cameras\<id>.json --out output\<id>\
<date-time>_yolo26s
```

A run writes `events.csv` - a row per counted crossing: `frame`, `time`, `movement`, `class`, `line`, `zone`, `track` (with a model of kinds also `detector_class`, `classifier_class`, `classifier_share`, `mark`) - and `summary.json`: `summary` (movement -> class -> vehicles), `total` (the whole recording, check lines and gate exits left out), `manoeuvres`, `backend`, `complete` (false - the recording was not read to its end). A run is repeated byte for byte on the same backend; different backends differ by single vehicles.

A manual count to compare with is `reference\<camera>\<recording>.json`:

```
{
  "classes": ["Car/Minibus", "Bus", "Small truck", "Medium truck", "Large truck",
"Special"],
  "engine_classes": { "car": [0, 5], "bus": [1], "truck": [2, 3, 4] },
  "counts": { "N1": [1073, 12, 28, 16, 20, 0] },
  "derived": { "P1": ["N2", "N3"] }
}
```

`counts` holds a number per column of the user's sheet, `engine_classes` folds the columns into the program's kinds, `derived` names check movements as sums. `main.py compare` prints manual, engine, difference and percent for every movement and kind.

A manual count is a point to start from, not the truth. People count the clearly seen vehicles exactly and estimate the rest. Where a run and a manual count disagree on a kind, lay the run's vehicles of that line on sheets (count with `--crops 3`, then `sheets --dir <run> --lines <line id> --classes bus truck`, and a sample of its cars), label them and count by the pictures: a third

count anyone can check. On the recording this program was made on, that third count agreed with the manual one within one or two vehicles on the near lines - once route vans were counted as buses, as the people who counted took them.

Teaching the kinds for the user's cameras

The detector (YOLO, taught on photographs of the whole world) names cars well and the rest badly: a route van is now a bus, now a truck, a long truck is a bus. A model of kinds - MobileNetV3-small, 6 MB - looks at three pictures of every counted vehicle and names its kind. It is joined to the detector by the rule "both": what the detector called a car stays a car unless the model is 90 % sure it is a bus - and is never renamed a truck; what the detector called a truck or a bus is the model's to name; a motorcycle stays the detector's. So the model corrects buses and trucks and does not spoil the cars. (Why never a truck: from a high camera dark cars look like trucks to the model, and it is as sure of the wrong ones as of the right ones; cars are nine vehicles in ten, so a few wrong renames spoil the count of trucks.)

A model comes with the program and is on from the start. It was taught on about 4 300 labelled vehicles of city cameras - from a pole, from a bridge, from the roadside, from a roof, in summer and in winter. Teach a model of the user's own when the default one names his cameras badly: check it first (step 5 below) on a few hundred labelled vehicles of his.

Measured on the recording the program was made on, at one line: by the detector buses -53 %, trucks +39 %; with a model taught on the first half of the same camera, on the second half buses +4 %, trucks +6 %, cars 0 %.

Two things decide the result, not the number of epochs and not a larger network:

- **pictures of the same kind of camera.** A model taught on 1400 vehicles of city cameras seen from the side and above called the trucks of a camera that sees them from the front cars (-56 %). About two hundred vehicles of that camera put it right. Teach on the user's own cameras and check on vehicles of them the model was not taught on;
- **the boundary of the kinds the user's sheets draw.** People who count by hand put a route van (windows along the whole side) among the buses; labelled as a car it made buses -75 %. Ask the user how their sheets count route vans and small trucks before labelling.

Time: cutting pictures is the long part (20 minutes of a 1080p recording of four cameras take 80 minutes on Intel graphics, minutes on an RTX card); teaching 25 epochs over about 3500 pictures takes half an hour on a laptop processor and a few minutes on an NVIDIA card. The first teaching downloads the ImageNet weights (10 MB).

The work

1. **Pictures.** Cut every vehicle of a part of a recording - lines are not needed:

   ```
   main.py harvest --video <file> --start 0 --end 1200
   ```

   (into `classes\cut\<recording name>\` beside the recording; a recording of a screen with several cameras: `--grid 2x2`, each camera into its own folder `11 12 21 22`). Or count a recording with `--crops 3`: the run folder then holds the pictures of its counted vehicles. Take two parts apart in time - two recordings, two days, two halves: one to teach on, one to check on.

2. **Sheets.** For every folder of pictures, first everything the detector did not call a car, then a sample of its cars:

   ```
   main.py sheets --dir <folder> --name big --classes bus truck motorcycle
   main.py sheets --dir <folder> --name car --classes car --sample 48
   ```

   A sheet is `<folder>\sheets\<name>_<nn>.jpg`: up to 48 pictures in rows of 8, cells numbered from 1. Buses and trucks are about one vehicle in ten; nearly all are in the first batch.

3. **Labels.** Look at a sheet and write beside it `<name>_<nn>.txt`: one letter a cell, in the order of the cells, a row of eight letters a line (spaces and line breaks mean nothing; the count of letters must equal the count of cells).

Letter	Label	What
`c`	car	a car, an off-roader, a pickup, a minivan, an ambulance, a police car
`v`	minibus	a route van: windows along the whole upper half of the side
`l`	light	a small truck: a van without windows along the side, a flatbed, a box van
`t`	truck	a truck larger than that: a lorry, a tipper, a tractor with a trailer
`g`	cargo	a truck for certain, small or large cannot be told
`b`	bus	a bus
`o`	trolley	a trolleybus
`r`, `m`	tram, moto	a tram, a motorcycle
`x`	junk	cannot be told, two vehicles in one picture, not a vehicle

When in doubt, `x`: a wrong label teaches worse than a missing one. Then `main.py labels --dir <folder>` collects the letters into `<folder>\labels.csv` and says how many of each label there are and how many cells still wait for letters. Show the user a sheet or two with your letters before going on. The marks the user sets in the TRAINING section of the window (`review.csv` of a run) are labels too.

4. **Teach.**

```
main.py teach --name <cameras>-01 --train <folders to teach on> --val <folders to check on>
```

A route van goes with the buses and a small truck with the trucks; `--minibus car` and `--light car` move them - ask the user. When the user's own pictures are few beside others in `--train`, draw them more often: `--boost <the start of their folder name> 4` (the pictures of a set are named after the folder they were cut from; a boost that names no folder of `--train` is refused). The best epoch is chosen by the `--val` folders; the set and a copy of the model are kept in `classes\taught\<name>\`, the model is `models\classes\<name>.pt`.

5. **Check** on labelled vehicles the model was not taught on:

```
main.py check-classes --classes <name> --val <folders>
```

Three confusion tables - the detector, the model alone, the rule "both" - and how many vehicles of each kind a count of those folders would give against how many there are. The number that matters is the last table, column "both". Within 10 % for each kind is a good result; say how many vehicles each percent stands on.

6. **Use.** In the window: TRAINING, MODEL OF KINDS, the model's name - the next count names the kinds with it. A run counted before with `--crops` is named anew without counting the video again: `main.py classify --run <run folder> --classes <name>`. The disputed vehicles of a run (the detector and the model disagree, or the model is not sure) are in TRAINING for the user to mark with a button.

7. **Better.** Look at what the model still gets wrong on the check folders, label more of that kind from the user's cameras, teach a new model under a new name, check again on the same folders. Stop when the check is within the target; more epochs do not help, more pictures of the right camera do.

Teaching the detector for a view it does not see

The general detector (`yolo26s.pt`) misses whole flows on some views: the far carriageway behind a barrier, where a car is a roof in single frames, and an unlit road at night, where a vehicle is its lights. Two taught detectors come with the program in `system_core\detectors\`: `yolo26s-barrier-01.pt` (vehicles behind a barrier, by day) and `yolo26s-night-01.pt` (the near carriageway of an unlit road at night; the far one behind a barrier it still misses). A taught detector works only in the zone that names it - `"model": "yolo26s-barrier-01.pt"` of a zone in the camera file - and every other line keeps the numbers of the general detector. In such a zone a car of the detector stays a car (the model of kinds took roofs for buses), and the same holds for a line with `"keep_car": true`. Set the zone in the file; the window shows it in the row of the zone.

When neither fits the user's view, a detector is taught for it. The user's word comes first: it takes hours of the GPU and, for marking by eye, the agent looks at many frames of the user's recordings (see "Rules that always hold"). The order:

1. **The exam first.** Choose two or three windows of 90 s the failing view is in, count them by eye (two narrow strips either side of the line, frame by frame) and with the general detector (`detector exam --model yolo26s.pt`). These windows are never taught on: every step below that takes frames gets them in `--skip`.

2. **Self-teaching** - when the detector sees the vehicles, only too weakly to follow them:

   ```
   main.py detector harvest --video REC --out weak\r1.csv --start S --end S --skip
   2940-3030
   main.py detector marks   --weak weak\r1.csv --out marks\r1.json
   main.py detector frames  --video REC --marks marks\r1.json --out frames --name r1
   --every 5
   main.py detector tiles   --frames frames --out sets\self
   ```

   Its boxes at a low confidence are joined into tracks; a track that lasts and was sure once is a vehicle in every frame of it, and a frame with a sure box outside every track is left out. The frames are cut into 640x640 tiles at their own scale. It does not help where the detector does not see a vehicle at all (a dark night): then mark by eye.

3. **Marking by eye**:

```
main.py detector pack  --video REC --out label\r1 --start S --end S --count 20
                       --models yolo26s.pt --skip 2940-3030
main.py detector check --pack label\r1 --frame fNNNNNN
main.py detector eye   --video REC --pack label\r1 --out sets\eye --name r1 --
copies 8
```

`pack` writes, for each frame, the road band brightened with a scale in frame pixels and the numbered guesses of the detectors (`fNNNNNN.jpg`, `fNNNNNN.json`) and the same band clean. The marker writes `fNNNNNN.marks.json`: `{"vehicles": [{"from": 2}, {"from": 4, "box": [x1, y1, x2, y2]}, {"box": [...], "kind": "truck"}], "unsure": [[x, y]]}` - a guess kept, a guess with its box corrected, a vehicle no guess has; a wrong guess is not listed; what cannot be told goes into "unsure" and is painted over in its tile. `check` draws the marks on the clean band: every vehicle has a box, every box sits on a vehicle. A vehicle seen only by its lights: the box around the pair of lights plus a sixth of their width on each side, about 0.7 of its width high for a car. Reflections on the road and on a barrier, beams and lamps are not vehicles. `eye` cuts the tiles from the frames as they are - dark, as the count sees them - and repeats them so a few hundred frames are not drowned by the rest.

4. **Teach, examine, ship**:

```
main.py detector join  --out sets\eye --sets sets\self --every 2
main.py detector teach --set sets\eye --name view-01 --epochs 8 --runs runs
main.py detector exam  --video REC --camera config\cameras\ID.json
                       --model runs\view-01\weights\best.pt --windows 2940-3030 --
out exam\view-01
main.py detector ship  --weights runs\view-01\weights\best.pt --name yolo26s-view-
01
```

`teach --from` goes on from a detector taught before. The exam is a real count of the windows of step 1 with the taught detector in `--zone` (every zone without it): it must come nearer the eye count than the general detector, and the lines it must not move (the near side, daylight) must stay. The numbers the teaching prints are not the result. `ship` puts the weights into `system_core\detectors` under a new name and never over a file.

When something goes wrong

Symptom	What to do
`camera lines are drawn for (640, 480), the video is (...)`	the recording has another size: a camera file of its own
`cannot open video`	a wrong path, or a file OpenCV cannot read: check it with `Tools\ffmpeg\bin\ffprobe.exe`
a far line counts almost nothing	the vehicles there are too small: a zone with `scale` 2, or a larger model (`--model yolo26m.pt`)
counts differ by a few vehicles between machines	different backends; compare runs of the same backend
`DirectML is not installed`	SETUP, the DIRECTML button, or `builder_main.cmd directml`
`teach` refuses: no labelled bus, truck	label more of them: `sheets --classes bus truck` on more pictures
`labels` refuses: letters for cells	the count of letters in a `.txt` differs from the count of cells of its sheet
a model makes the cars worse	it was taught on other cameras: label about two hundred vehicles of this one and teach again
the computer sleeps during a long cut	`harvest` and `teach` keep the machine awake by themselves; a closed lid sleeps all the same