# Implementación con py-raildriver (controles, señales*, eventos, paradas, paso por vía)

> Base: `raildriver/library.py` y `raildriver/events.py` (proyecto **py-raildriver**). Objetivo: un **colector** que unifica controles de cabina + "virtuales" de RailDriver y, opcionalmente, eventos/limitaciones que obtenemos por **LUA** (próximo límite, paradas de pasajeros, paso por marcadores).
>
> * Sobre "señales": la DLL de RailDriver no da el aspecto de la señal directamente. En su lugar, capturamos **sistemas de seguridad** (PZB/SIFA/LZB/AFB) si la loco los expone como controles, y/o usamos un **bridge LUA** para próximos límites/eventos de vía.

---

## 0) Estructura propuesta

```
/tsc-ai
  /ingestion
    rd_client.py          # Wrapper py-raildriver (controles + especiales +
listener)
    lua_eventbus.py       # Lector de eventos LUA (JSONL tail)
  /runtime
    collector.py          # Bucle que fusiona RD + LUA y vuelca CSV/JSONL
    csv_logger.py         # Cabecera dinámica, ';', ISO8601
    events_bus.py         # Normalizador de eventos (speed_limit_change,
stop_begin, ...)
  /lua
    tsc_eventbus.lua      # Script LUA opcional para eventos/limit y paradas
  /profiles
    BR146.json            # Mapeo controles → alias + escalados; min/max por
control
  /data
    runs/*.csv            # Telemetría de cada sesión (10–20 Hz)
    events/*.jsonl        # Eventos discretos (cambios, paradas, marcadores)
```

---

## 1) rd_client.py (py-raildriver wrapper)

```python
# /ingestion/rd_client.py
from __future__ import annotations
import time
from typing import Dict, Any, Iterable
from raildriver import RailDriver
from raildriver.events import Listener

SPECIAL_KEYS = [
```

```
    "!Coordinates", "!FuelLevel", "!Gradient", "!Heading",
    "!IsInTunnel", "!Time", "!LocoName"
]

class RDClient:
    def __init__(self, poll_hz: float = 10.0) -> None:
        self.rd = RailDriver()
        self.rd.setRailSimConnected(True)
        self.rd.setRailDriverConnected(True)
        self.poll_dt = 1.0 / poll_hz
        # Descubrir controles disponibles (nombre → idx) y rangos
        self.controllers = list(self.rd.get_controller_list())
# iterable de (index, name)
        self.ctrl_index_by_name = {name: idx for idx, name in
self.controllers}
        self.minmax = {
            name: (
                self.rd.get_min_controller_value(idx),
                self.rd.get_max_controller_value(idx),
            )
            for name, idx in self.ctrl_index_by_name.items()
        }
        # Listener para cambios
        self.listener = Listener(self.rd, interval=self.poll_dt)
        for sk in SPECIAL_KEYS:
            self.listener.add(sk)
        for name in self.ctrl_index_by_name.keys():
            self.listener.add(name)

    # --- Lecturas puntuales ---
    def read_specials(self) -> Dict[str, Any]:
        # py-raildriver expone helpers; aquí usamos listener snapshot para
unificar
        snap = self.listener.snapshot()
        out: Dict[str, Any] = {}
        # LocoName → [Provider, Product, Engine]
        if "!LocoName" in snap:
            loco = snap["!LocoName"] or []
            if isinstance(loco, (list, tuple)) and len(loco) >= 3:
                out.update({
                    "provider": loco[0],
                    "product": loco[1],
                    "engine": loco[2],
                })
        # Coordenadas/tiempo/rumbo/pendiente…
        coords = snap.get("!Coordinates")
        if coords and isinstance(coords, (list, tuple)) and len(coords) >= 2:
            out["lat"], out["lon"] = coords[0], coords[1]
        if "!Heading" in snap: out["heading"] = snap["!Heading"]
        if "!Gradient" in snap: out["gradient"] = snap["!Gradient"]
        if "!FuelLevel" in snap: out["fuel_level"] = snap["!FuelLevel"]
```

```
        if "!IsInTunnel" in snap: out["is_in_tunnel"] = bool(snap["!
IsInTunnel"])
        if "!Time" in snap:
            # !Time suele venir como datetime.time o [h,m,s]
            tval = snap["!Time"]
            if isinstance(tval, (list, tuple)) and len(tval) >= 3:
                out["time_ingame_h"], out["time_ingame_m"],
out["time_ingame_s"] = tval[:3]
            else:
                out["time_ingame"] = str(tval)
        return out

    def read_controls(self, names: Iterable[str]) -> Dict[str, float]:
        snap = self.listener.snapshot()
        res = {}
        for n in names:
            if n in snap:
                res[n] = float(snap[n])
            else:
                # fallback lectura directa
                idx = self.ctrl_index_by_name.get(n)
                if idx is not None:
                    res[n] = float(self.rd.get_current_controller_value(idx))
        return res

    def stream(self) -> Iterable[Dict[str, Any]]:
        """Genera dicts con specials + subset de controles comunes."""
        common_ctrls = self._common_controls()
        while True:
            row = self.read_specials()
            row.update(self.read_controls(common_ctrls))
            # Derivar metrías útiles
            v = row.get("SpeedometerKPH") or row.get("SpeedometerMPH")
            if v is not None:
                if "SpeedometerMPH" in row:
                    v_ms = float(v) * 0.44704
                else:
                    v_ms = float(v) / 3.6
                row["v_ms"], row["v_kmh"] = v_ms, v_ms * 3.6
            yield row
            time.sleep(self.poll_dt)

    def _common_controls(self) -> Iterable[str]:
        names = set(self.ctrl_index_by_name.keys())
        prefer = [
            "SpeedometerKPH", "SpeedometerMPH",
            "Regulator", "Throttle",
            "TrainBrakeControl", "LocoBrakeControl",
            "DynamicBrake", "Reverser",
            # seguridad (si existen)
            "SIFA", "VigilEnable", "PZB_85", "PZB_70", "PZB_55",
```

```
            "PZB_1000", "PZB_500", "PZB_40",
            "AFB_Speed", "LZB_V_SOLL", "LZB_V_ZIEL", "LZB_DISTANCE",
        ]
        return [n for n in prefer if n in names]
```

---

## 2) lua_eventbus.py (tail de eventos)

```
# /ingestion/lua_eventbus.py
from __future__ import annotations
import json, time, os
from typing import Dict, Any, Iterable, Optional

class LuaEventBus:
    def __init__(self, path: str, from_end: bool = True) -> None:
        self.path = path
        self.pos = 0
        if from_end and os.path.exists(self.path):
            self.pos = os.path.getsize(self.path)

    def poll(self) -> Optional[Dict[str, Any]]:
        if not os.path.exists(self.path):
            time.sleep(0.1)
            return None
        with open(self.path, "r", encoding="utf-8", errors="ignore") as f:
            f.seek(self.pos)
            line = f.readline()
            if not line:
                return None
            self.pos = f.tell()
            try:
                return json.loads(line)
            except json.JSONDecodeError:
                return None

    def stream(self) -> Iterable[Dict[str, Any]]:
        while True:
            evt = self.poll()
            if evt:
                yield evt
            else:
                time.sleep(0.05)
```

---

## 3) events_bus.py (normalizador de eventos)

```
# /runtime/events_bus.py
from __future__ import annotations
from typing import Dict, Any

# Convierte eventos crudos (LUA o heurísticas) a un modelo estable
# type: "speed_limit_change" | "stop_begin" | "stop_end" | "marker_pass" |
"custom"

def normalize(evt: Dict[str, Any]) -> Dict[str, Any]:
    t = evt.get("type")
    base = {"t_ingame": evt.get("time"), "lat": evt.get("lat"), "lon":
evt.get("lon")}
    if t == "speed_limit_change":
        base.update({
            "type": t,
            "limit_prev_kmh": evt.get("prev"),
            "limit_next_kmh": evt.get("next"),
            "dist_est_m": evt.get("dist"),
        })
    elif t in ("stop_begin", "stop_end"):
        base.update({"type": t, "station": evt.get("station")})
    elif t == "marker_pass":
        base.update({"type": t, "marker": evt.get("name")})
    else:
        base.update({"type": "custom", "payload": evt})
    return base
```

---

## 4) csv_logger.py

```
# /runtime/csv_logger.py
from __future__ import annotations
import csv, os
from typing import Dict, Any, Iterable

class CsvLogger:
    def __init__(self, path: str, delimiter: str = ";") -> None:
        self.path = path
        self.delimiter = delimiter
        self.fieldnames = None
        os.makedirs(os.path.dirname(path), exist_ok=True)

    def write_row(self, row: Dict[str, Any]) -> None:
        if self.fieldnames is None:
            self.fieldnames = sorted(row.keys())
            newfile = not os.path.exists(self.path)
```

```
            with open(self.path, "w", newline="", encoding="utf-8") as f:
                w = csv.DictWriter(f, fieldnames=self.fieldnames,
delimiter=self.delimiter)
                w.writeheader()
                if newfile:
                    w.writerow({k: row.get(k, "") for k in self.fieldnames})
                else:
                    w.writerow({k: row.get(k, "") for k in self.fieldnames})
        else:
            missing = [k for k in row.keys() if k not in self.fieldnames]
            if missing:
                # reescribir con cabecera extendida (simple y robusto para
MVP)
                self.fieldnames.extend(sorted(missing))
                with open(self.path, "r", encoding="utf-8") as f:
                    lines = f.read().splitlines()
                with open(self.path, "w", newline="", encoding="utf-8") as f:
                    w = csv.DictWriter(f, fieldnames=self.fieldnames,
delimiter=self.delimiter)
                    w.writeheader()
                    for line in lines[1:]:
                        f.write(line + "\n")
            with open(self.path, "a", newline="", encoding="utf-8") as f:
                w = csv.DictWriter(f, fieldnames=self.fieldnames,
delimiter=self.delimiter)
                w.writerow({k: row.get(k, "") for k in self.fieldnames})
```

---

## 5) collector.py (fusiona RD + LUA → CSV + eventos)

```
# /runtime/collector.py
from __future__ import annotations
import os, time, json
from typing import Dict, Any
from ingestion.rd_client import RDClient
from ingestion.lua_eventbus import LuaEventBus
from runtime.csv_logger import CsvLogger
from runtime.events_bus import normalize

CSV_PATH = os.path.join("data", "runs", "run.csv")
EVT_PATH = os.path.join("data", "events", "events.jsonl")
LUA_BUS  = os.path.join("data", "lua_eventbus.jsonl")  # ajusta ruta si usas
otra


def run(poll_hz: float = 10.0) -> None:
    os.makedirs(os.path.dirname(CSV_PATH), exist_ok=True)
    os.makedirs(os.path.dirname(EVT_PATH), exist_ok=True)
    rd = RDClient(poll_hz=poll_hz)
```

```
    csvlog = CsvLogger(CSV_PATH)
    bus = LuaEventBus(LUA_BUS)

    last_evt_write = 0.0
    while True:
        row = next(rd.stream())
        row["t_wall"] = time.time()
        csvlog.write_row(row)
        # drenar uno o varios eventos LUA si los hay
        for _ in range(10):
            evt = bus.poll()
            if not evt:
                break
            nrm = normalize(evt)
            with open(EVT_PATH, "a", encoding="utf-8") as f:
                f.write(json.dumps(nrm, ensure_ascii=False) + "\n")
        # ritmo base
        time.sleep(max(0.0, 1.0/poll_hz - 0.0005))

if __name__ == "__main__":
    run(10.0)
```

---

## 6) tsc_eventbus.lua (opcional, para límites/paradas/marcadores)

Colócalo como **script de escenario** y adapta `EVENTBUS_PATH`. Emite **una línea JSON por evento** (JSONL). Ejemplos de eventos: cambio de próximo límite, inicio/fin de parada (heurístico por velocidad ≈0 durante X s), y marcador (si decides invocarlo desde un marcador personalizado).

```
-- /lua/tsc_eventbus.lua (ejemplo mínimo)
local EVENTBUS_PATH = "C:/Users/Public/Documents/tsc_eventbus.jsonl"  --
ajusta si lo prefieres dentro de RailWorks
local last_speed_ms = 0.0
local stopped = false
local stop_t0 = nil
local last_limit = nil

local function json_escape(s)
  s = string.gsub(s, "\\", "\\\\")
  s = string.gsub(s, '"', '\\"')
  return s
end

local function emit(evt)
  local f = io.open(EVENTBUS_PATH, "a")
  if not f then return end
  f:write(evt .. "\n")
  f:close()
```

```
end

local function emit_json(tbl)
  local parts = {}
  for k,v in pairs(tbl) do
    local vs
    if type(v) == "string" then
      vs = '"' .. json_escape(v) .. '"'
    else
      vs = tostring(v)
    end
    table.insert(parts, '"'..k..'":'..vs)
  end
  emit('{'.. table.concat(parts, ",") ..'}')
end

function Update(time)
  -- Velocidad y hora in-game
  local v = SysCall("PlayerEngine:GetSpeed") or 0.0 -- m/s
  local hours = SysCall("PlayerEngine:GetTime") or 0.0
  -- Próximo límite (si disponible en este contexto)
  local next_limit = SysCall("PlayerEngine:GetNextSpeedLimit", 0, 0)

  -- Evento: cambio de próximo límite
  if next_limit and next_limit ~= last_limit then
    emit_json({ type = "speed_limit_change", prev = last_limit or -1, next =
next_limit, time = hours })
    last_limit = next_limit
  end

  -- Heurística de parada: velocidad < 0.2 m/s sostenida ≥ 4 s
  local now = os.time()
  if v < 0.2 then
    if not stopped then
      stop_t0 = stop_t0 or now
      if now - stop_t0 >= 4 then
        stopped = true
        emit_json({ type = "stop_begin", time = hours })
      end
    end
  else
    stop_t0 = nil
    if stopped then
      stopped = false
      emit_json({ type = "stop_end", time = hours })
    end
  end

  last_speed_ms = v
end
```

```lua
-- Llamable opcional desde un marcador personalizado
function MarkerPassed(name)
  emit_json({ type = "marker_pass", name = tostring(name) })
end
```

**Notas LUA**: 1) `GetNextSpeedLimit` puede no estar disponible en todos los contextos; si el script de escenario no lo ve, muévelo a un script de **locomotora** o añade un proxy. 2) Para “paso por vía”/marcadores, crea un pequeño script de marcador que llame a `MarkerPassed("<nombre>")` al activarse. 3) Para “paradas de pasajeros”, puedes reforzar la heurística con lectura de **DoorsOpen** si tu locomotora expone ese control, o vinculando un marcador de andén.

---

## 7) Qué información concreta obtendrás

### De py-raildriver (directo)

- **Identidad** de la loco (provider/product/engine).
- **GPS/rumbo/pendiente/hora in-game/túnel/combustible** (campos especiales).
- **Controles de cabina** con lectura y rango **min/max**: velocímetro (KPH/MPH), **Regulator/Throttle**, **TrainBrake**, **LocoBrake**, **DynamicBrake**, **Reverser**, y, si existen, **SIFA/PZB/LZB/AFB** y periféricos (sander, luces, bocina…).

### De LUA (eventbus)

- **Cambio de próximo límite** (`speed_limit_change` con `prev/next`).
- **Paradas de pasajeros** (heurística `stop_begin` / `stop_end`; opcionalmente con nombre de estación si llamas desde marcador de andén).
- **Paso por vía/marcadores** (`marker_pass` con `name`).

Todo queda: - En **CSV** continuo (`data/runs/*.csv`) a 10 Hz (configurable) con métricas de conducción. - En **JSONL** de eventos (`data/events/*.jsonl`) para hitos discretos.

---

## 8) Roadmap corto de integración

1. Ejecutar `collector.py` y comprobar que `runs/run.csv` recibe **v_ms** y controles.
2. Añadir `tsc_eventbus.lua` al escenario y verificar que se crean eventos (`events.jsonl`).
3. Enriquecer `profiles/*.json` con alias de controles por locomotora (p. ej., `Throttle` vs `Regulator`).
4. Añadir detección de **PZB/SIFA** desde los nombres de controles presentes.
5. Integrar **braking_curves.py** y planificador por **próximos límites**.
6. Extender LUA para etiquetar **paradas por estación** (marcador de andén) y enviar **distancia al límite** si el contexto lo permite.