Step 1 → spin up a fresh “Python (Poetry)” Replit so you get a virtual-env and the Nix compiler tool-chain. Add the Rust-backed **demoparser** wheel (poetry add demoparser) together with duckdb pyarrow shapely fastapi uvicorn python-socketio pandas – Rust code is pre-compiled in the wheel, so Replit’s container can import it straight away [GitHub](https://github.com/LaihoE/demoparser?utm_source=chatgpt.com).

Step 2 → create two folders at project root: /replays for raw .dem files and /lake for the Parquet you will write.

Parsing and persisting

Step 3 → write parse.py. The essential loop is

import demoparser, pyarrow as pa, pyarrow.parquet as pq, pathlib

demo = demoparser.parse("replays/match.dem",

props=["tick","time","round","steamid","team",

"pos\_x","pos\_y","pos\_z",

"view\_x","view\_y","vel\_x","vel\_y",

"is\_flashed","has\_c4","grenade","event"])

tbl = pa.Table.from\_pylist(demo) # ~100 k rows

pq.write\_table(tbl, "lake/match.parquet")

That single table now holds every tick for every entity in the match – the same numbers your 2-D canvas already consumes, only persisted for analysis.

Query engine

Step 4 → open a DuckDB connection from a Python REPL or a script:

python

CopyEdit

import duckdb, pathlib

con = duckdb.connect()

con.execute("""

CREATE VIEW ticks AS

SELECT \* FROM parquet\_scan('lake/\*.parquet')

""")

DuckDB scans the Parquet lazily, so memory stays tiny even on the free Replit plan [DuckDB](https://duckdb.org/docs/stable/clients/python/overview.html?utm_source=chatgpt.com).

Metric layer

Step 5 → drop a file models.py that defines one SQL view (or Python UDF) per metric in your CSDK sheet. Three examples show the pattern:

-- hot-zone share (B-site)

CREATE VIEW hot\_zone\_B AS

SELECT steamid,

SUM(frame\_ms) \* 1.0 / SUM(SUM(frame\_ms)) OVER (PARTITION BY round) AS pct\_time

FROM ticks

WHERE pos\_x BETWEEN 1800 AND 2600

AND pos\_y BETWEEN -800 AND 200

GROUP BY round, steamid;

-- rotation latency (bomb planted → arrive within 500 units of site)

CREATE VIEW rotation\_latency AS

SELECT t.steamid,

MIN(t.time) - p.time AS ms\_after\_plant

FROM ticks t

JOIN (SELECT round,time FROM ticks WHERE event='bomb\_planted') p

ON t.round = p.round

WHERE sqrt((t.pos\_x-2000)^2 + (t.pos\_y+500)^2) < 500

GROUP BY t.round,t.steamid;

-- utility effectiveness (flash assist)

CREATE VIEW flashes AS

SELECT flash.thrower AS thrower,

victim.steamid AS blinded,

COUNT(\*) AS flashes\_landed

FROM ticks flash

JOIN ticks victim

ON flash.event='flash\_thrown'

AND victim.event='player\_blinded'

AND victim.time BETWEEN flash.time AND flash.time+2500

GROUP BY thrower, blinded;

Because every metric is a view over ticks, you never duplicate data and you can add new ideas in minutes.

API bridge

Step 6 → stand up api.py with FastAPI:

python

CopyEdit

from fastapi import FastAPI

from fastapi.responses import StreamingResponse

import duckdb, json, asyncio

con = duckdb.connect()

app = FastAPI()

@app.get("/metrics/{metric}")

def metric(metric: str):

df = con.execute(f"SELECT \* FROM {metric}").fetchdf()

return df.to\_dict(orient="records")

@app.websocket("/ticks/ws")

async def ticks\_ws(ws):

await ws.accept()

for row in con.execute("SELECT time,pos\_x,pos\_y,steamid FROM ticks ORDER BY time"):

await ws.send\_text(json.dumps(row))

await asyncio.sleep(0.016) # ~60 fps playback

Run locally with uvicorn api:app --reload; Replit’s “Run” button will do the same.

Viewer hook-up

Step 7 → inside your existing JavaScript canvas, swap the file-based data feed for a WebSocket:

js

CopyEdit

const ws = new WebSocket("wss://your-repl.tld/ticks/ws");

ws.onmessage = e => {

const {time,pos\_x,pos\_y,steamid} = JSON.parse(e.data);

drawDot(steamid, pos\_x, pos\_y);

};

Fetch metric panels via fetch('/metrics/hot\_zone\_B') and render them in the sidebar; because both endpoints query DuckDB you are always looking at identical ground-truth.

Iterate and extend

Step 8 → populate the rest of the CSDK sheet: angle exposure, entry success, post-plant clutch win-rate, whatever you need. For each new request the loop is the same: add a SQL view → expose it in the API → paint or list it in the UI.  
 Step 9 → drop new demos into /replays and rerun parse.py; no other code changes are required because the tick schema is identical across matches.  
 Step 10 → when you want to share, enable Replit’s web view or export the container; nothing else to deploy.

Follow these ten steps and your basic 2-D demo viewer becomes a full scouting console powered by the same positional feed it already understands, now enriched with every analytical dimension defined in the CSDK analysis metric set – all without asking an AI model to watch the replay.

Once the parser has turned a demo into a single Parquet / DuckDB table, everything else in Replit is just **queries** and **function calls** that fill the metric objects you already defined.

### **How the pieces hook together**

pgsql

CopyEdit

.dem ┌─────────────┐

replay ───────────► │ parse.py │ ←– Rust demoparser

└─────────────┘

│ writes Parquet

▼

┌─────────────┐

│ DuckDB view │ ←– 1 row ≈ 1 tick

└─────────────┘

│ SELECT / UDF

┌──────── your existing metric-functions ───────┐

│ │

hot-zone share ▸ SQL density aggregate │

rotation time ▸ SQL window fn │

territory ▸ Python Voronoi UDF │

util value ▸ SQL+Python join on grenade events │

… every row in the CSDK sheet uses the same table … │

└───────────────────────────────────────────────┘

│ .fetchdf() /.fetchall()

▼

┌──────────────┐

│ FastAPI JSON │ ←– /metrics/\*

└──────────────┘

│ HTTP / WebSocket

▼

your 2-D viewer

1. **Ingest** – parse.py match.dem drops match.parquet in /lake.
2. **Analyse** – When your code (or a REST call) asks for SELECT \* FROM hot\_zone\_B, DuckDB evaluates the view on the fly and returns the numbers.
3. **Present** – Your front-end (or a Google-Sheets webhook, or a CLI) simply consumes those JSON / DataFrame results.

### **Minimal glue you still need**

**Wire your metric functions to the DB** In models.py expose each metric as either  
  
 python  
CopyEdit  
def hot\_zone\_B(match\_id):

return con.execute("SELECT … WHERE match=?", [match\_id]).fetchdf()

* or a plain SQL view if no Python logic is required.
* **Replace hard-coded CSV inputs** Anywhere your current rating code expects a manual CSV upload, swap the read-csv call for a fetchdf() from DuckDB.

**(Optional) persist snapshots** If you want a frozen record for sheets/BI tools, run  
  
 python  
CopyEdit  
con.execute("""

COPY (SELECT \* FROM hot\_zone\_B WHERE match=?)

TO 'exports/hot\_zone\_B.csv' WITH (HEADER)

""", [match\_id])

* on a schedule or after each ingest.

That’s it. The parser gives you authoritative XY-Z, DuckDB turns it into whatever metric the CSDK spec calls for, and you surface the result wherever you like inside the same Replit workspace. No additional data sources, no manual uploads—just stream demos in and read scouting metrics out.

You only need one extra move: pull the **grenade-specific game-events into their own table and give each row a grenade\_type flag**. The tick stream you are already collecting tells you who is holding a grenade (grenade bool or inventory info) but it does **not** record when a HE explodes, a smoke blooms, or a molly starts burning; those are emitted as separate events such as hegrenade\_detonate, smokegrenade\_detonate, flashbang\_detonate, inferno\_startburn, decoy\_detonate. Demoparser (and demoinfocs-golang if you prefer Go) can query them directly:

from demoparser2 import DemoParser

import pyarrow as pa, pyarrow.parquet as pq

p = DemoParser("replays/match.dem")

# one call fetches every detonation / start-burn event and tags it

events = p.parse\_event(

["hegrenade\_detonate",

"flashbang\_detonate",

"smokegrenade\_detonate",

"inferno\_startburn",

"decoy\_detonate"],

player = ["steamid"],

position = ["x","y","z"],

other = ["round","time"]) # :contentReference[oaicite:0]{index=0}

tbl = pa.Table.from\_pylist(

[{\*\*row, "grenade\_type": row["event"].split("\_")[0]} for row in events]

)

pq.write\_table(tbl, "lake/grenade\_events.parquet")

Now you have two parquet tables:

* ticks.parquet all per-tick X Y Z and player state (your original snippet)
* grenade\_events.parquet one row per nade with a clean grenade\_type

DuckDB can join them on time or round to compute every utility-driven metric:

sql

CopyEdit

-- flash effectiveness (enemies inside 875-unit radius and line-of-sight)

CREATE VIEW flash\_effect AS

SELECT g.steamid AS thrower,

v.steamid AS victim,

g.time,

distance(g.x,g.y, v.x,v.y) AS dist

FROM grenade\_events g

JOIN ticks v

USING (round)

WHERE g.grenade\_type = 'flashbang'

AND v.team <> g.team

AND v.time BETWEEN g.time AND g.time + 2500 -- 2.5 s window

AND dist < 875;

Exactly the same pattern gives you

* smoke coverage – calculate whether a point is inside the smoke radius between detonate and expire
* molotov area denial – integrate seconds a CT or T would have to path through the burn polygon
* HE damage and entry-nade success – join to player\_hurt events
* decoy information gain – count enemies that turned to face the sound cue

Because grenade\_type is now a dimension column, dashboards and rating formulas in Replit can pivot on it without code changes: SELECT grenade\_type, SUM(damage) … GROUP BY grenade\_type.

So, yes, you **do** factor each grenade, but you do it once—by writing a tidy grenade\_events table and a three-line SQL mapping—then every downstream metric (and your 2-D viewer’s coloured trajectories) can key on that field. No other changes to the ingest pipeline are required, and the rest of your CSDK metric logic stays exactly as already wired into Replit.

That is the whole adjustment.

Data-first approach: Always examine the actual data files first before building any features. Extract and catalog all unique values and field names to understand the true data structure.

Create data validation early: Implement validation for incoming data that checks for valid participant IDs and expected data fields, raising clear error messages when unexpected values appear.

Use runtime type checking: When working with external data sources, implement runtime type checking to verify that the data being processed matches expectations.

Build comprehensive test cases: Create test cases specifically for handling edge cases like missing or invalid data.

Avoid hardcoded values: Instead of hardcoding any values in the application code, extract them to configuration objects or constants based on the actual dataset.

Create data dictionaries: Document all expected data fields, their formats, and valid values in a data dictionary that's used as a reference throughout development.

Regular data integrity checks: Add logging and verification steps that monitor the data being processed to catch inconsistencies early.

Graceful fallbacks: Design the UI to handle missing or invalid data gracefully, showing appropriate error messages rather than failing silently.

Iterative validation: Verify at each stage of data processing that the data maintains integrity and matches expectations.

Ask clarifying questions: If there's any uncertainty about data structure or valid values, I should ask you specific questions rather than making assumptions.





