

User: could you generate a txt file with all the context needed by another llm agentic mode to generate the project for me please ?

Kimi: Below is a single, self-contained context.txt file you can feed to any LLM in “agentic” mode

(e.g., Cursor Composer, GitHub Copilot Workspace, Aider, etc.).

It contains every instruction, file path, and code block required to reproduce the complete CS2 demo-analysis & ML training project in Rust.

Simply copy everything between the markers into `context.txt`, then ask your agent to

“generate the project from context.txt”.

-----BEGIN CONTEXT.TXT-----

Project: CS2 Demo Analysis & AI Training System (Rust)

OVERVIEW

We need a fully-working Rust workspace that:

1. Parses Counter-Strike 2 demo files (.dem) using the Git version of `demoparser2`.
2. Converts player snapshots into numerical “behavior vectors”.
3. Exports a Parquet dataset for ML training.
4. Trains a feed-forward behavior-cloning network (tch-rs/LibTorch).
5. Serves the trained policy over TCP so an in-game bot can request pro-like actions.

REPO LAYOUT

```
cs2-ml/
├── Cargo.toml
├── src/
│   ├── main.rs      # CLI (prepare / train / serve)
│   ├── data.rs     # demo → vector → Parquet
│   ├── model.rs    # tch-rs network + training loop
│   └── server.rs   # TCP policy server
└── README.md      # quick-start
```

GLOBAL CONSTRAINTS

- macOS / Linux / WSL compatible
- Rust 1.75+
- Requires LibTorch (CPU is OK) and Python 3 headers (for demoparser2)

STEP-BY-STEP INSTRUCTIONS

1. Create the workspace

```
cargo new cs2-ml --bin  
cd cs2-ml
```

2. Replace Cargo.toml with the following:

```
[package]  
name = "cs2-ml"  
version = "0.1.0"  
edition = "2021"  
  
[[bin]]  
name = "cs2-ml"  
path = "src/main.rs"  
  
[dependencies]  
  
git version of the parser  
demoparser2 = { git = "https://github.com/LaihoE/demoparser.git", rev = "main" }  
  
tch = "0.14"           # LibTorch bindings  
parquet = "52"  
arrow = "52"  
ndarray = "0.15"  
clap = { version = "4", features = ["derive"] }  
serde = { version = "1", features = ["derive"] }  
glob = "0.3"  
tracing = "0.1"  
tracing-subscriber = "0.3"  
bytemuck = "1.14"  
anyhow = "1.0"  
  
[build-dependencies]  
pyo3-build-config = "0.20"
```

3. Place each file below under src/ with exact names.

```
----- src/data.rs -----  
use demoparser2::{DemoParser, PlayerMeta};  
use parquet::file::writer::{SerializedFileWriter};  
use arrow::datatypes::{DataType, Field, Schema};
```

```
use arrow::array::{Float32Array, UInt32Array, UInt16Array};
use arrow::record_batch::RecordBatch;
use std::path::{Path, PathBuf};
use anyhow::Result;
```

```
#[derive(Debug, serde::Serialize, serde::Deserialize)]
pub struct BehavioralVector {
    pub tick: u32,
    pub steamid: u64,
    pub health: f32,
    pub armor: f32,
    pub pos_x: f32,
    pub pos_y: f32,
    pub pos_z: f32,
    pub vel_x: f32,
    pub vel_y: f32,
    pub vel_z: f32,
    pub yaw: f32,
    pub pitch: f32,
    pub weapon_id: u16,
    pub ammo: f32,
    pub is_airborne: f32,
    pub delta_yaw: f32,
    pub delta_pitch: f32,
}
```

```
pub fn vectors_from_demo(path: impl AsRef) -> Result<Vec> {
    let parser = DemoParser::new();
    let bytes = std::fs::read(path)?;
    let parsed = parser.parse(&bytes)?;
    let mut out = Vec::new();
    let ticks = parsed.ticks();
    for w in ticks.windows(2) {
        let cur = &w[0];
        let nxt = &w[1];
        for (cur_p, nxt_p) in cur.players().zip(nxt.players()) {
            let c = PlayerMeta::from(cur_p);
            let n = PlayerMeta::from(nxt_p);
            let weap_id = c.active_weapon_name.as_deref().unwrap_or("none").chars().fold(0u16, |a, b|
            a.wrapping_add(b as u16));
            out.push(BehavioralVector {
                tick: cur.number() as u32,
                steamid: c.steamid,
                health: c.props.get("m_iHealth").and_then(|v| v.parse().ok()).unwrap_or(0) as f32,
```

```

armor: c.props.get("m_ArmorValue").and_then(|v| v.parse().ok()).unwrap_or(0) as f32,
pos_x: c.props.get("m_vecOrigin[0]").and_then(|v| v.parse().ok()).unwrap_or(0.0),
pos_y: c.props.get("m_vecOrigin[1]").and_then(|v| v.parse().ok()).unwrap_or(0.0),
pos_z: c.props.get("m_vecOrigin[2]").and_then(|v| v.parse().ok()).unwrap_or(0.0),
vel_x: c.props.get("m_vecVelocity[0]").and_then(|v| v.parse().ok()).unwrap_or(0.0),
vel_y: c.props.get("m_vecVelocity[1]").and_then(|v| v.parse().ok()).unwrap_or(0.0),
vel_z: c.props.get("m_vecVelocity[2]").and_then(|v| v.parse().ok()).unwrap_or(0.0),
yaw: c.props.get("m_angEyeAngles[1]").and_then(|v| v.parse().ok()).unwrap_or(0.0),
pitch: c.props.get("m_angEyeAngles[0]").and_then(|v| v.parse().ok()).unwrap_or(0.0),
weapon_id: weap_id,
ammo: c.ammo_clip.unwrap_or(0) as f32,
is_airborne: if c.props.get("m_hGroundEntity").map_or(true, |v| v == "-1") { 1.0 } else { 0.0 },
delta_yaw: n.props.get("m_angEyeAngles[1]").and_then(|v| v.parse().ok()).unwrap_or(0.0) -
c.props.get("m_angEyeAngles[1]").and_then(|v| v.parse().ok()).unwrap_or(0.0),
delta_pitch: n.props.get("m_angEyeAngles[0]").and_then(|v| v.parse().ok()).unwrap_or(0.0) -
c.props.get("m_angEyeAngles[0]").and_then(|v| v.parse().ok()).unwrap_or(0.0),
});
}
}
Ok(out)
}

```

```

pub fn write_parquet(vecs: &[BehavioralVector], out_path: &Path) -> Result<()> {
let schema = Schema::new(vec![
    Field::new("tick", DataType::UInt32, false),
    Field::new("steamid", DataType::UInt64, false),
    Field::new("health", DataType::Float32, false),
    Field::new("armor", DataType::Float32, false),
    Field::new("pos_x", DataType::Float32, false),
    Field::new("pos_y", DataType::Float32, false),
    Field::new("pos_z", DataType::Float32, false),
    Field::new("vel_x", DataType::Float32, false),
    Field::new("vel_y", DataType::Float32, false),
    Field::new("vel_z", DataType::Float32, false),
    Field::new("yaw", DataType::Float32, false),
    Field::new("pitch", DataType::Float32, false),
    Field::new("weapon_id", DataType::UInt16, false),
    Field::new("ammo", DataType::Float32, false),
    Field::new("is_airborne", DataType::Float32, false),
    Field::new("delta_yaw", DataType::Float32, false),
    Field::new("delta_pitch", DataType::Float32, false),
]);
let file = std::fs::File::create(out_path)?;
let mut writer = SerializedFileWriter::new(file, std::sync::Arc::new(schema), Default::default())?;

```

```

let cols: Vec<Box> = vec![
    Box::new(UInt32Array::from_iter(vecs.iter().map(|v| v.tick))),
    Box::new(UInt64Array::from_iter(vecs.iter().map(|v| v.steamid))),
    Box::new(Float32Array::from_iter(vecs.iter().map(|v| v.health))),
    Box::new(Float32Array::from_iter(vecs.iter().map(|v| v.armor))),
    Box::new(Float32Array::from_iter(vecs.iter().map(|v| v.pos_x))),
    Box::new(Float32Array::from_iter(vecs.iter().map(|v| v.pos_y))),
    Box::new(Float32Array::from_iter(vecs.iter().map(|v| v.pos_z))),
    Box::new(Float32Array::from_iter(vecs.iter().map(|v| v.vel_x))),
    Box::new(Float32Array::from_iter(vecs.iter().map(|v| v.vel_y))),
    Box::new(Float32Array::from_iter(vecs.iter().map(|v| v.vel_z))),
    Box::new(Float32Array::from_iter(vecs.iter().map(|v| v.yaw))),
    Box::new(Float32Array::from_iter(vecs.iter().map(|v| v.pitch))),
    Box::new(UInt16Array::from_iter(vecs.iter().map(|v| v.weapon_id))),
    Box::new(Float32Array::from_iter(vecs.iter().map(|v| v.ammo))),
    Box::new(Float32Array::from_iter(vecs.iter().map(|v| v.is_airborne))),
    Box::new(Float32Array::from_iter(vecs.iter().map(|v| v.delta_yaw))),
    Box::new(Float32Array::from_iter(vecs.iter().map(|v| v.delta_pitch))),
];
let batch = RecordBatch::try_new(std::sync::Arc::new(schema.clone()), cols)?;
writer.write(&batch)?;
writer.close()?;
Ok(())
}

```

----- [src/model.rs](#) -----

```

use tch::{nn, nn::Module, Tensor};

```

```

pub struct BehaviorNet {
    layers: Vec<>,
}

```

```

impl BehaviorNet {
    pub fn new(vs: &nn::Path, in_dim: i64, out_dim: i64) -> Self {
        let layers = vec![
            nn::linear(vs / "l1", in_dim, 128, Default::default()),
            nn::linear(vs / "l2", 128, 64, Default::default()),
            nn::linear(vs / "l3", 64, out_dim, Default::default()),
        ];
        BehaviorNet { layers }
    }
}

```

```

    pub fn forward(&self, xs: &Tensor) -> Tensor {
        let mut x = xs.shallow_clone();
    }
}

```

```

    for (i, l) in self.layers.iter().enumerate() {
        x = l.forward(&x);
        if i < self.layers.len() - 1 {
            x = x.relu();
        }
    }
    x
}

pub fn train(
    vs: &nn::Path,
    dataset: Vec<(Vec<f32>, Vec<f32>)>,
    epochs: i64,
) -> anyhow::Result<()> {
    let net = BehaviorNet::new(vs, 14, 2);
    let mut opt = nn::Adam::default().build(vs, 1e-3)?;
    let xs: Vec<f32> = dataset.iter().flat_map(|(x, _)| x.clone()).collect();
    let ys: Vec<f32> = dataset.iter().flat_map(|(_, y)| y.clone()).collect();
    let xs = Tensor::from_slice(&xs).reshape([dataset.len() as i64, 14]);
    let ys = Tensor::from_slice(&ys).reshape([dataset.len() as i64, 2]);

    for epoch in 1..=epochs {
        let pred = net.forward(&xs);
        let loss = pred.mse_loss(&ys, tch::Reduction::Mean);
        opt.zero_grad();
        loss.backward();
        opt.step();
        if epoch % 100 == 0 {
            println!("epoch {epoch} loss {}", f64::from(&loss));
        }
    }
    Ok(())
}
}

```

----- [src/server.rs](#) -----

```

use std::net::{TcpListener, TcpStream};
use std::io::{Read, Write};
use tch::{nn, Tensor};

pub fn serve(model_path: &str, port: u16) -> anyhow::Result<()> {
    let vs = nn::VarStore::new(tch::Device::Cpu);
    vs.load(model_path)?;
}

```

```

let net = crate::model::BehaviorNet::new(&vs.root(), 14, 2);
let listener = TcpListener::bind(format!("0.0.0.0:{}", port))?;
println!("Policy server listening on port {}", port);
for stream in listener.incoming() {
let mut stream = stream?;
let mut buf = [0u8; 14 * 4];
stream.read_exact(&mut buf)?;
let vec: Vec = bytemuck::cast_slice(&buf).to_vec();
let input = Tensor::from_slice(&vec).reshape([1, 14]);
let output = net.forward(&input);
let mut out_bytes = [0u8; 8];
output.detach().to_device(tch::Device::Cpu).copy_data(&mut out_bytes, 2);
stream.write_all(&out_bytes)?;
}
}

```

----- src/[main.rs](#) -----

```

use clap::{Parser, Subcommand};
use std::path::{Path, PathBuf};

```

```

mod data;
mod model;
mod server;

```

```

#[derive(Parser)]
#[command(name = "cs2-ml")]
#[command(about = "CS2 behavior-cloning ML pipeline")]
struct Cli {
#[command(subcommand)]
command: Commands,
}

```

```

#[derive(Subcommand)]
enum Commands {
/// Convert demos → Parquet
Prepare {
demo_glob: String,
output_dir: PathBuf,
},
/// Train the policy network
Train {
parquet: PathBuf,
model_out: PathBuf,
#[arg(long, default_value = "1000")]

```

```

epochs: i64,
},
/// Serve the trained policy
Serve {
model: PathBuf,
#[arg(long, default_value = "8123")]
port: u16,
},
}

```

```

fn main() -> anyhow::Result<()> {
tracing_subscriber::fmt::init();
let cli = Cli::parse();
match cli.command {
Commands::Prepare { demo_glob, output_dir } => {
std::fs::create_dir_all(&output_dir)?;
for entry in glob::glob(&demo_glob)? {
let demo = entry?;
let vecs = data::vectors_from_demo(&demo)?;
let out = output_dir.join(demo.file_stem().unwrap()).with_extension("parquet");
data::write_parquet(&vecs, &out)?;
println!("Wrote {}", out.display());
}
}
Commands::Train { parquet, model_out, epochs } => {
use parquet::file::reader::SerializedFileReader;
let reader = SerializedFileReader::new(std::fs::File::open(parquet)?)?;
let mut rows = reader.get_row_iter(None)?;
let mut dataset = Vec::new();
for row in rows {
let vec: Vec = (0..14).map(|i| row.get_float(i).unwrap() as f32).collect();
let label = vec![row.get_float(14).unwrap() as f32, row.get_float(15).unwrap() as f32];
dataset.push((vec, label));
}
let vs = tch::nn::VarStore::new(tch::Device::Cpu);
model::BehaviorNet::train(&vs.root(), dataset, epochs)?;
vs.save(model_out)?;
println!("Model saved to {}", model_out.display());
}
Commands::Serve { model, port } => {
server::serve(model.to_str().unwrap(), port)?;
}
}
Ok(())

```



```
}
```

----- README.md -----

CS2-ML (prototype)

Quick start

```
```bash
```

```
1. install LibTorch CPU (macOS example)
```

```
curl -L https://download.pytorch.org/libtorch/cpu/libtorch-macos-2.1.0.zip -o libtorch.zip
```

```
unzip libtorch.zip -d /opt
```

```
2. build
```

```
export LIBTORCH=/opt/libtorch
```

```
export DYLD_LIBRARY_PATH=$LIBTORCH/lib:$DYLD_LIBRARY_PATH
```

```
cargo build --release
```

```
3. prepare dataset
```

```
./target/release/cs2-ml prepare "demos/*.dem" ./data
```

```
4. train
```

```
./target/release/cs2-ml train ./data/*.parquet ./policy.ot --epochs 1000
```

```
5. serve
```

```
./target/release/cs2-ml serve ./policy.ot --port 8123
```

```
```
```

The TCP server returns 8 bytes: two little-endian f32 (delta_yaw, delta_pitch) for each 14-f32 input vector.