# Oscilloscope Parser & Pipeline — Complete Fix, Missing Pieces, and Implementation Plan

This document explains exactly what is wrong, what is missing, and how to fix and implement the missing parts so your oscilloscope renders real NGSpice transient waveforms. It includes end-to-end code structure, server/client patches, a robust parser, and verification steps to prove correctness.

## Executive Summary

NGSpice is producing tens of thousands of transient points, but your custom parser extracts zero because the output lines contain concatenated scientific numbers without spaces. The client also reads results via a global and renders scalar values repeatedly. Fix by: (1) switching to a guaranteed-parseable output (WRDATA CSV) or adding a robust concatenated-number parser, (2) wiring transient arrays through the server interface, and (3) rendering full series on the client without globals.

## What’s Wrong (from your analysis)

- parseCustomTransientFormat assumes whitespace-delimited columns; actual NGSpice lines are concatenated (‘4999994.999830e-024.999830e-025.000000e+00’).

- Indices and floats merge into one token → split(/\s+/) returns one element; downstream indices are undefined.

- Oscilloscope accesses results from window global instead of the returned simulation object; state becomes stale and non-deterministic.

- Net-to-node mapping is inconsistent; extraction by numeric node is brittle if names are available and stable.

## Fix Strategy Overview

- Prefer WRDATA CSV output from NGSpice to avoid custom parsing entirely; parse the CSV with a simple splitter.

- If you must parse the concatenated format, use a regex tokenizer that finds scientific numbers and an index peel-off for the leading integer.

- Stop using window.cachedNgSpiceResult; pass the NGSpice result through a proper function/store and render series arrays (t, v).

- Standardize net names in the netlist and record named voltages (v(Q1E), v(NOISE), v(SCOPE\_OUT)), avoiding fragile node-number coupling.

## A) Server — Output a Clean, Parseable CSV (Preferred)

Modify the .control block to write an ASCII CSV that lists exactly the nets you need, in known order.

// server/utils/ngspice-interface.ts (when building the netlist/control)  
const netsToRecord = ["Q1E", "NOISE", "SCOPE\_OUT"]; // add/remove as needed  
const csvPath = tmp.join("tran.csv"); // choose your temp path  
  
const control = `  
.control  
set filetype=ascii  
set wr\_singlescale  
tran 0.1us 50ms uic  
wrdata ${csvPath} time ${netsToRecord.map(n => `v(${n})`).join(' ')}  
quit  
.endc  
`;

Then parse the CSV:

// server/utils/ngspice-interface.ts  
import \* as fs from "fs/promises";  
  
async function readWrdataCsv(csvPath: string, nets: string[]) {  
 const raw = await fs.readFile(csvPath, "utf8");  
 const time = [];  
 const voltages: Record<string, number[]> = Object.fromEntries(nets.map(n => [n, []]));  
  
 for (const line of raw.split(/\r?\n/)) {  
 const t = line.trim();  
 if (!t || t.startsWith("\*")) continue;  
 const cols = t.split(/\s+/);  
 if (cols.length < 1 + nets.length) continue;  
 time.push(Number(cols[0]));  
 for (let i = 0; i < nets.length; i++) {  
 voltages[nets[i]].push(Number(cols[1 + i]));  
 }  
 }  
 return { time, voltages };  
}

## B) Server — Robust Concatenated-Line Parser (If CSV not possible)

If you must parse lines like ‘4999994.999830e-024.999830e-025.000000e+00’, use a two-stage tokenizer: peel off the leading index (integer), then tokenize the remaining string as scientific numbers.

// server/utils/ngspice-interface.ts  
function tokenizeConcatenatedLine(line: string): { index: number, values: number[] } | null {  
 const s = line.trim();  
 if (!s) return null;  
  
 // 1) Peel leading integer index (sequence of digits at start)  
 const m = s.match(/^(\d+)(.\*)$/);  
 if (!m) return null;  
 const index = Number(m[1]);  
 let rest = m[2];  
  
 // 2) Tokenize scientific numbers (with optional leading sign)  
 const numRe = /[+-]?(?:\d+\.\d+|\d+)(?:e[+-]?\d+)?/ig;  
 const matches = rest.match(numRe);  
 if (!matches || matches.length === 0) return { index, values: [] };  
  
 const values = matches.map(v => Number(v));  
 return { index, values };  
}  
  
// Example usage inside parseCustomTransientFormat:  
function parseCustomTransientFormat(block: string, colsPerLine: number) {  
 const time: number[] = [];  
 const voltages: Record<string, number[]> = {}; // e.g. {'Q1E':[], 'NOISE':[]}  
 const lines = block.split(/\r?\n/);  
 for (const raw of lines) {  
 const tok = tokenizeConcatenatedLine(raw);  
 if (!tok || tok.values.length < colsPerLine) continue;  
 // Expected order after index: [time, v(Q1E), v(NOISE), v(SCOPE\_OUT), ...]  
 time.push(tok.values[0]);  
 // fill voltages from tok.values[1..]  
 }  
 return { time, voltages };  
}

This approach is resilient to missing spaces and variable exponent widths.

## C) Server — Result Shape & Net Naming

Return named arrays and include netToNodeMap only if you also support numeric extraction. Named signals are simpler and less brittle.

export interface NgSpiceTransient {  
 time: number[];  
 voltages: Record<string, number[]>; // keys are net names: 'Q1E', 'NOISE', etc.  
 currents: Record<string, number[]>;  
}  
  
export interface NgSpiceResult {  
 success: boolean;  
 data?: {  
 operatingPoint: Record<string, number>;  
 transientData?: NgSpiceTransient;  
 };  
}

## D) Client — Ditch window globals; pass the result through

Replace global access with a returned object or a store selector.

// client/src/utils/circuit-simulation.ts  
export interface Series { t: number[]; v: number[]; }  
  
export function getSeriesNamed(result: NgSpiceResult, net: string): Series | null {  
 const td = result?.data?.transientData;  
 if (!td) return null;  
 const v = td.voltages?.[net];  
 if (!v || !td.time || v.length !== td.time.length) return null;  
 return { t: td.time, v };  
}

// client/src/components/circuit/oscilloscope-panel.tsx  
useEffect(() => {  
 let mounted = true;  
 (async () => {  
 const sim = await simulateCircuitProfessional(components); // returns NgSpiceResult  
 const series = getSeriesNamed(sim, monitoredNet);  
 setWaveformData(series ?? { t: [], v: [] });  
 })();  
 return () => { mounted = false; };  
}, [componentsHash, monitoredNet]);

## E) Optional: Differential channel by name

export function getDiffSeriesNamed(result: NgSpiceResult, pos: string, neg?: string): Series | null {  
 const a = getSeriesNamed(result, pos);  
 if (!a || !neg) return a;  
 const b = getSeriesNamed(result, neg);  
 if (!b || b.t.length !== a.t.length) return a;  
 const v = a.v.map((x, i) => x - b.v[i]);  
 return { t: a.t, v };  
}

## F) Verification & Logging

// server (after parsing)  
console.info("Parsed transient points:", data.transientData?.time?.length ?? 0);  
if (data.transientData) {  
 const keys = Object.keys(data.transientData.voltages);  
 console.info("Signals:", keys.join(", "));  
 console.info("First row:", data.transientData.time[0], ...keys.map(k => data.transientData.voltages[k][0]));  
}  
  
// client  
if (!series || series.t.length < 2) {  
 setBanner("No transient data — showing DC only");  
}

## G) Tests & Acceptance

Unit tests:

- tokenizeConcatenatedLine: parses index=499999 and values=[4.999830e-02, 4.999830e-02, 5.000000e+00].  
- parseCustomTransientFormat: returns time.length > 100 and equal-length voltage arrays.  
- readWrdataCsv: reads CSV and yields aligned arrays.

Integration/E2E:

- Netlist uses WRDATA; CSV parsed with >1000 points; time monotonic.  
- Changing R8 220k→22k changes NOISE amplitude at next render.  
- Switching monitored net updates waveform without reload.

Acceptance criteria:

- transientData.time.length > 100 and equals every voltages[key].length

- client renders non-flat waveform; responds to component changes

- no usage of window globals; result passed via function/store

- parser handles both CSV and concatenated formats (or CSV only)