Signal Processing Visualizer — Production Document

Project: Signal Processing Visualizer (SPV)

Owner: Nishant R. Kadu

Date: 05 Sep 2025

---

1. Executive Summary

Signal Processing Visualizer (SPV) is a production-ready web application that allows students, educators, and engineers to upload or generate signals and interactively explore time-domain and frequency-domain representations, apply filters, perform transformations (DFT/FFT), convolution, and other DSP operations. The goal is to provide a polished, educational, and engineering-grade tool that showcases your ECE + full-stack skills and can be deployed, scaled, and maintained as a production service.

Key selling points for your resume: - Combines ECE (DSP) expertise with full-stack engineering. - Includes a polished frontend with interactive visualizations. - Demonstrates backend design, API development, and optional DevOps/CI-CD pipeline. - Optional hardware integration (record from microphone or from Arduino/ESP32 sensors).

---

2. Goals & Success Metrics

Primary goals - Build an interactive, fast, and accurate DSP visualizer supporting typical DSP operations. - Make the tool intuitive for students while also useful for engineers. - Deploy to a cloud provider with a CI/CD pipeline and monitoring.

Success metrics - 95% of unit tests passing. - Page load time under 1.5s for the dashboard. - Real-time visualization response latency under 200ms for typical datasets ($\leq$ 100k samples). - 100+ GitHub stars & 50+ unique demo users within the first 3 months (optional marketing metric).

---

3. Target Users & Personas

1. **ECE Students** — want a learning tool to visualize transforms and filters.
2. **Instructors/TAs** — want to demo concepts in class quickly.
3. **Embedded Engineers** — want to inspect real sensor data and pre-process it.
4. **Hobbyists** — experimenting with audio signals and filters.

---

4. Feature List (MVP & Beyond)

MVP Features (must-have)

- Upload signals (CSV, WAV, TXT) and generate built-in demo signals (sine, square, sawtooth, noise).
- Time-domain plot (zoom, pan, cursor readouts, sample index/time readout).
- Frequency-domain plot (DFT/FFT magnitude and phase), slider for FFT size and windowing.
- Basic filters: FIR (windowed), IIR (biquad), lowpass/highpass/bandpass/bandstop with adjustable cutoff and order.
- Convolution (user selects two signals and computes convolution) with visualization.
- Export processed signal (WAV/CSV).
- Responsive UI (desktop & tablet) and accessible controls.

Secondary Features (post-MVP)

- Real-time audio capture via microphone, live visualization.
- Integration with hardware: stream ADC samples from ESP32/Arduino via WebSocket.
- Spectrogram display and STFT controls (window type, overlap, hop size).
- Filter designer (specify passband/stopband and get filter coefficients).
- Compare multiple signals with synchronized cursors.
- Shareable permalink for sessions (store in DB or encode state in URL).

---

5. Tech Stack

Frontend - React (Vite) + TypeScript - Plotting: Recharts / Chart.js for basic plots, but for interactive DSP visualizations **prefer**: Plotly.js or custom WebGL canvas (using WebGL or Canvas2D) for performance. Recommendation: **Plotly.js** for quick interactivity; move to WebGL if needed. - UI: Tailwind CSS (you already use it), Headless UI for components. - State management: React Query for async + local Zustand for local state (optional)

Backend - Node.js + Express (TypeScript) - Processing microservice: Python (FastAPI) for heavy DSP operations (NumPy, SciPy) — optional but recommended for numerical accuracy and convenience. - Auth: JWT + refresh tokens (if user accounts are needed).

Data & Storage - MongoDB (Atlas) for user accounts, saved sessions/meta data. - Files: S3-compatible storage (AWS S3 / DigitalOcean Spaces) for uploaded files.

Realtime / Streaming - WebSocket via Socket.IO if real-time audio/ADC streaming is implemented.

CI / CD / DevOps - GitHub Actions for CI (lint, tests, build, image build). - Docker for containerization. - Container registry: GitHub Container Registry or Docker Hub. - Cloud deployment: Render / Vercel (frontend), AWS ECS/Fargate or DigitalOcean App Platform for backend; or deploy the whole stack on a single VPS for simplicity. - Monitoring: Sentry (errors) + Prometheus + Grafana or lightweight: LogDNA/ LogRocket.

Extras - Reverse proxy & TLS: Nginx or managed TLS via platform.

---

6. System Architecture

High-level components: 1. **Frontend (React)** — UI, visualizations, file upload, audio capture. 2. **API Gateway (Express)** — handles authentication, user requests, job submission. 3. **DSP Processor (Python FastAPI)** — executes FFT, filters, convolution, spectrograms, returns results (JSON or binary arrays). Optionally this runs as a separate microservice or within same Node runtime via worker processes. 4. **Storage (S3 + MongoDB)** — store uploaded files, processed outputs, user sessions. 5. **Realtime Layer (Socket.IO)** — stream live audio/sensor data to backend + broadcast processed results back. 6. **CI/CD** — GitHub Actions build/test/deploy pipeline.

Diagram (textual):

```
[User Browser] <-> [React Frontend] <-> [Express API] <-> [DSP Service (FastAPI/
Python)]
                                    |-> [MongoDB]
                                    |-> [S3 Storage]
                                    |-> [WebSocket (Socket.IO)]
```

---

7. Data Flow & API Contracts

1) Upload File - POST /api/upload - Body: multipart/form-data: file - Response: `{ fileId, url, samples, sampleRate, channels }`

2) Generate Demo Signal - POST /api/signals/generate - Body: `{ type: "sine", frequency: 440, amplitude:1, phase:0, sampleRate:44100, duration:2 }` - Response: `{ signalId, samples[], sampleRate }`

3) Compute FFT - POST /api/dsp/fft - Body: `{ signalId, window: "hann", nfft: 8192 }` or raw samples - Response: `{ freq[], magnitude[], phase[], metadata }`

4) Apply Filter - POST /api/dsp/filter - Body: `{ signalId, filterType: "butter", kind:"lowpass", cutoff:1000, order:4 }` - Response: `{ outSignalId, samples[], metadata }`

5) Convolution - POST /api/dsp/convolve - Body: `{ signalAId, signalBId, mode: "full" }` - Response: `{ outSignalId, samples[] }`

All DSP endpoints accept either IDs referencing uploaded/back-end stored samples or raw samples (arrays) when small.

---

8. DSP Implementation Details

Prefer implementing numerical DSP tasks in Python using NumPy + SciPy. Reasons: - Familiar, well-tested numerical libraries. - SciPy provides filtering, windowing, STFT, and spectral tools.

Core algorithms to implement - **FFT/DFT**: `numpy.fft` for FFT; add scaling and windowing, return one-sided spectrum when appropriate. - **Filters**: - FIR: `scipy.signal.firwin` with window options (rect, hann, hamming, blackman) - IIR: design via `scipy.signal.butter`, `scipy.signal.cheby1` and apply with `lfilter` or `sosfilt` for stability. - **Convolution**: `numpy.convolve` or `scipy.signal.fftconvolve` for large signals. - **STFT / Spectrogram**: `scipy.signal.stft` or `spectrogram` with adjustable window and overlap. - **Resampling**: `scipy.signal.resample` or `resample_poly`.

Numerical considerations - Use single precision floats for memory/perf but keep double where precision matters. - Normalize FFT outputs and document conventions (Hz bins, one-sided vs two-sided, dB vs linear magnitude).

---

9. Frontend UX / Wireframes

Pages & Components: - Landing / Demo page (quick demo signals and `Try Now`). - Dashboard / Workspace: - Left: Controls panel (upload, generate, operations list) - Center: Main plot area with tabs (Time, Frequency, Spectrogram) - Right: Signal list with metadata, buttons for operations (filter, fft, conv, export). - Modal dialogs: filter designer, export options, real-time streaming.

UX Notes: - Provide contextual help/tooltips for DSP parameters. - Show computational cost estimates (e.g. `FFT 8192 samples — ~10 ms`) using sample benchmarking. - Keyboard shortcuts for zoom/pan.

---

10. Database Schema (MongoDB)

users

```
{ _id, name, email, passwordHash, createdAt, profile }
```

files

```
{ _id, ownerId, filename, s3Url, sampleRate, channels, length, createdAt }
```

signals

```
{ _id, ownerId, name, sourceFileId, samplesRef, sampleRate, createdAt,
metadata }
```

sessions (optional)

```
{ _id, ownerId, name, signals: [signalIds], operations: [ {op, params,
timestamp} ], shareToken }
```

---

11. Testing Plan

Unit tests - DSP funcs: verify FFT magnitude symmetry, inverse FFT correctness, filter frequency response meets spec. - API routes: schema validation, edge cases (very short signals, empty uploads).

Integration tests - Full workflow: upload -> apply filter -> FFT -> export.

E2E tests - Use Playwright for critical UI flows: upload file, generate demo, apply operation, export.

Performance tests - Benchmark FFT sizes (1k, 10k, 100k samples) and measure latency. - Load test backend with multiple simultaneous requests using k6 or Locust.

---

12. CI/CD & Deployment

CI (GitHub Actions) - `on: [push, pull_request]` - Jobs: - lint-frontend (ESLint, TypeScript) - lint-backend (ESLint/flake8) - unit-tests (npm test / pytest) - build artifacts (frontend static bundle) - docker-build (build backend & dsp service images)

CD - On push to `main` -> run deployment workflow. - Options: - Vercel for frontend automatic deploy from main branch. - Backend: deploy Docker image to AWS ECS/Fargate or Render. - Use GitHub Container Registry for images.

Infrastructure as Code (optional) - Use Terraform for cloud infra (VPC, ECS, RDS/Atlas, S3 buckets).

---

13. Observability & Monitoring

- **Logs**: Structured JSON logs to stdout (captured by platform). Use Winston (Node) and Python `structlog`.
- **Errors**: Sentry for exception tracking.
- **Metrics**: Prometheus exporter (or platform metrics) for request latency, error rates, CPU/memory.
- **Alerts**: PagerDuty / email for high error rates or resource exhaustion.

14. Security Considerations

- Validate uploads (size limits, MIME type checks) to avoid denial-of-service.
- Rate limit DSP endpoints and use per-user quotas for heavy operations.
- Sanitize and validate all inputs to DSP functions — avoid eval or dynamic code execution.
- Use HTTPS everywhere; secure cookies and HTTP-only refresh tokens.
- If storing user data, provide a privacy policy and allow deletion of data.

---

15. Deployment Costs & Hosting Recommendations (Estimate)

Low-cost dev/prod setup: - Frontend: Vercel Hobby (free tier) — $0 - Backend: DigitalOcean App Platform / Render Basic — $7–20/month - Python DSP worker: same host or separate small instance — $5–15/month - MongoDB Atlas free tier (developer) — $0, paid tiers $9+/month - S3 storage: negligible for small files — $1–5/month

Total estimated: $15–40/month for a usable small-scale deployment.

---

16. Roadmap & Milestones (6–8 weeks realistic plan)

Week 1 (Design & Setup) - Finalize feature list and wireframes. - Setup repo, monorepo or separate repos (frontend, backend, dsp). - Basic CI pipeline and linting.

Week 2 (MVP Backend) - Implement upload API, file storage, signal model. - Implement FFT endpoint and basic filter endpoint in DSP service.

Week 3 (MVP Frontend) - Build dashboard shell, file upload UI, demo generator. - Wire up FFT visualization and basic controls.

Week 4 (Operations & Export) - Implement convolution and export features. - Add tests and fix critical bugs.

Week 5 (Polish & Deployment) - Add authentication (optional), orchestrate CI/CD, Dockerize services and deploy. - Add monitoring and logging.

Week 6 (Extras & Buffer) - Real-time audio capture, spectrograms, hardware integration (optional). - Final QA, README, and project write-up for resume.

---

17. Repo Structure Suggestion

Monorepo (recommended) or two repos.

```
/spv
  /frontend (react + ts)
  /api (node/express)
  /dsp (python fastapi)
  /infra (terraform or deployment scripts)
  /ci (github actions workflows)
  README.md
```

---

18. README Template (for GitHub)

Title: Signal Processing Visualizer (SPV)

Short description: Interactive DSP visualizer built with React + FastAPI + NumPy/SciPy. Upload or generate signals, apply filters, compute FFT/DFT, convolution, spectrograms, and export results.

Highlights: - Interactive time & frequency visualization - SciPy-based DSP backend - Real-time audio capture (optional) - Dockerized, CI/CD ready

Quickstart 1. `git clone ...` 2. `cd spv && cd frontend && npm install && npm run dev` 3. `cd ../api && npm install && npm run dev` 4. `cd ../dsp && pip install -r requirements.txt && uvicorn dsp:app --reload`

Contributing - Branch naming, PR template, code of conduct.

---

19. UI Copy & Microcopy Examples

- Upload hint: "Upload WAV/CSV/TXT (max 10 MB). Sample rate must be specified for CSV/TXT."
- FFT tooltip: "FFT size controls frequency resolution. Larger N gives finer bins but slower computation."

---

20. Appendix: Example API Request (FFT)

```
POST /api/dsp/fft
Content-Type: application/json
{
  "sampleRate": 44100,
  "samples": [0.0, 0.1, 0.3, ...],
  "window": "hann",
```

```
  "nfft": 8192
}
```

Response:

```
{
  "freq": [0.0, 5.383, 10.766, ...],
  "magnitude": [0.12, 0.31, 0.05, ...],
  "phase": [...],
  "metadata": { "nfft":8192, "sampleRate":44100 }
}
```

---

21. Next Steps (what I can do for you right now)

- Generate starter code scaffolding for frontend/backend/dsp (boilerplate + simple FFT demo).
- Create GitHub Actions workflows and Dockerfiles.
- Sketch React component structure and key UI components.

Tell me which of these you want me to start now and I'll create the files (or a canvas with the code) for you.

End of document.