Quick recommendations (short)

- **Backend:** Python + **FastAPI** (lightweight, async, great dev DX). Use `librosa` + `soundfile` for audio analysis & server-side time-stretching (preserves pitch).
- **Frontend:** JS framework of your choice (React/Vue/Svelte). I recommend **React + Vite** for speed and ecosystem — but **Svelte** is simpler if you want minimal code.
- **Audio engine (frontend): Tone.js** for drum/chord sequencing, scheduling, transport, bpm/ time signature control. Use **wavesurfer.js** for waveform & loop/cut UI.
- **Tuner / pitch detection (frontend):** WebAudio `getUserMedia` + a pitch detection library (e.g., `pitchfinder` — YIN or AMDF).
- **CSS:** Tailwind CSS (you already chose).
- **Persisting presets/configs:** JSON files. Simple, load/save to disk via file-download / file-input.

High-level architecture

- Frontend handles interactive audio sequencing and immediate playback (Tone.js).
- Backend handles heavier analysis and deterministic audio processing (BPM detection, time-stretch with pitch-preservation, and producing new audio files on request).
- All data stored locally in repo during development (no DB required). Presets are JSON files inside repo or downloadable to user machine.

Main functional pieces & components

Frontend components

(Each bullet = small React component or Svelte component.)

- `App` — top-level state (current BPM, time signature, tuning, transport running, current preset).
- `TransportBar` — start/stop, count-in toggle, master BPM control, tempo input, metronome on/off.
- `DrumMachine` — grid sequencer for Kick/Snare/HiHat/OpenHat. Uses Tone.js Sampler or Player to trigger samples.

  - props: `pattern` (array), `bpm`, `timeSignature`.
  - emits: `patternChanged`.
- `ChordMachine` — grid sequencer for chord slots, chord selection UI, chord-sound selector, automation options (arpeggio/strum patterns).
  - supports tuning offset (A4 frequency) — apply via detune or calculate shifted sample synthesis.
- `SamplerSelector` — choose sample/instrument for chord playback (list of sample packs).
- `WaveformEditor` — uses wavesurfer.js to show waveform, let user set start/end loop, play loop, cut, export selection.
- `AudioUploader` — upload file to backend (or client decode) + show detected BPM.
- `BpmAnalyzer` — UI that displays backend BPM results + manual override.
- `Tuner` — live mic tuner using WebAudio + pitch detection, shows note name, cents offset relative to selected A4.
- `LoudnessControl` — master gain slider (maps to Tone.js `Master` node or WebAudio GainNode).
- `PresetManager` — save preset to JSON, load preset from file, import/export.
- `Settings` — set tuning (A4 value), time signature, sample pack choices, export options.

Backend endpoints (FastAPI)

- `POST /upload` — accepts audio file, returns `{ filename, duration, sample_rate }`.
- `POST /analyze/bpm` — runs `librosa.beat.tempo` and returns detected BPM and confidence / beat times.
- `POST /process/time-stretch` — body: `{ filename, target_bpm }` or `{ rate }`; returns processed audio file (WAV/OGG) URL or streamed file. Implementation uses `librosa.effects.time_stretch`.
- `GET /presets/{name}` — return preset JSON (optional).
- `POST /presets` — upload preset JSON to server or store in repo (optional).
- Serve processed files from `/processed/<file>` via static file route.

Security note: this is a private repo + local dev; keep CORS permissive only for your local dev host.

Data model — preset JSON (example)

```
{
  "meta": {
    "name": "D-A-G-Em-C_example",
    "created": "2025-11-06T12:00:00Z",
    "version": 1
  },
  "transport": {
    "bpm": 120,
    "timeSignature": [4,4],
    "countIn": true,
    "tuning": 440.0
  },
  "drumMachine": {
    "stepCount": 16,
    "pattern": {
      "kick":  [1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0],
      "snare": [0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0],
      "hh":    [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
      "oh":    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
    },
    "samples": {
      "kick": "samples/kick-01.wav",
      "snare": "samples/snare-01.wav"
    }
  },
  "chordMachine": {
    "stepCount": 8,
    "pattern": ["D","-","A","G","-","Em","C","-"],
    "chordSounds": {
      "slot0": "piano-1",
      "slot1": "pad-2"
    },
    "arpeggio": {
      "enabled": true,
      "type": "up-down",
      "rate": "8n"
    }
  }
}
```

- – means no chord on that slot.
- Save & load this JSON (user downloads file or uploads to restore state).

Libraries & why

- **Frontend**
  - **Tone.js** — scheduling, samplers, transport (easy tempo/time signature changes).
  - **WaveSurfer.js** — waveform display and range selection (loop/cut UI).
  - **pitchfinder** (or `ml5` pitch detection) — tuner detection (YIN recommended).
  - **FileSaver.js** for preset/file downloads.
- **Backend**
  - **FastAPI** — simple, async endpoints.
  - **librosa** — bpm detection (`librosa.beat.beat_track` or `librosa.beat.tempo`) and time-stretch (`librosa.effects.time_stretch`) which preserves pitch.
  - **soundfile (pysoundfile)** — read/write WAV/FLAC.
  - Optional: **ffmpeg** (system binary) if you want format conversion or to rely on `pydub` for format handling.

Implementation details & tips

Drum & chord machine (frontend)

- Use Tone.Transport as the master clock. Set `Tone.Transport.bpm.value = bpm`.
- Create a `Tone.Part` or `Tone.Sequence` for each instrument with an array of booleans/steps.
- For chords: use either sampler samples for each chord or a synth (Tone.PolySynth) that triggers chord notes computed from chord name and tuning offset (calculate MIDI notes from chord labels).
- Tuning: if using synths, set `detune` for nodes or compute note frequencies with `A4` base frequency (e.g., for A4=442). If using samples, you can pitch-shift samples via playbackRate + resampling (but that can be messy). Synths make tuning changes trivial.

BPM / time signature

- Transport supports only BPM; for time signature you can render grid lengths dynamically (e.g., `stepCount = beatsPerBar * pulsesPerBeat`).

- When switching from 4/4 to 3/4, update UI and Tone.Sequence lengths.

Count-in & metronome

- If count-in enabled, schedule 4 metronome clicks before starting Transport. Tone.js supports scheduling events relative to Transport.

Tuner

- Use microphone via `navigator.mediaDevices.getUserMedia({ audio: true })`, feed into AnalyserNode, implement YIN using `pitchfinder` or port the algorithm. Show note and cent offset relative to selected A4.

Waveform / upload / loop / cut

- Use wavesurfer.js to show waveform and allow range selection. For playback of the uploaded audio while preserving pitch when BPM changed:
    - Option A (fast): change playbackRate in WebAudio — this changes pitch (maybe acceptable for quick tests).
    - Option B (correct): send request to backend to time-stretch the audio using `librosa.effects.time_stretch(y, rate)` and return processed file. This preserves pitch.
- Offer both: client-side fast preview (playbackRate) and server-side render for exports.

BPM detection algorithm (backend)

- Use `librosa.load`, then `librosa.beat.beat_track` or `librosa.beat.tempo` to estimate BPM. Return beat positions to the frontend for visually aligning loops.

File sizes & formats

- Accept common audio formats (.wav, .mp3, .ogg). Convert mp3/ogg to WAV on the backend using `ffmpeg` before processing if needed.
- Limit uploads (e.g., 50 MB) or warn.

Milestones (implementation plan — do in this order)

1. **Scaffold repo**: Vite React app + Tailwind + FastAPI backend skeleton. Confirm CORS.
2. **Basic Transport + DrumMachine using Tone.js**: implement grid UI (static samples), start/stop, bpm change.

3. **ChordMachine**: implement chord selection, chord-to-notes mapping, play chords with Tone.PolySynth. Implement tuning change (A4).
4. **Preset save/load**: export/import JSON (PresetManager).
5. **Waveform upload & display**: integrate wavesurfer.js, implement upload endpoint, show waveform.
6. **BPM detection**: implement `/analyze/bpm` using librosa and show result in UI.
7. **Loop/cut UI**: user selects segment in wavesurfer, play loop.
8. **Time-stretch processing**: quick client preview via `playbackRate`; production-quality export via POST to `/process/time-stretch` (librosa).
9. **Tuner**: implement mic-based pitch detection and tuning display.
10. **Polish UI, add metronome and count-in, gain control, sample packs**.
11. **Testing & docs**: create README, dev run instructions, sample presets.

Edge cases & gotchas

- **librosa time-stretch quality**: Good for moderate tempo changes. For extreme time-stretch you may prefer `rubberband` but it is more complex to set up (native binary).
- **Latency**: Browser audio scheduling (Tone.js) is very good — but sample loading must be preloaded to avoid glitch.
- **CORS & file URLs**: When running frontend & backend separately, configure CORS on FastAPI and use absolute URLs for processed file downloads.
- **Formats**: Client browsers may not decode every format; server-side conversion via `ffmpeg` is safest.
- **Browser autoplay**: User gesture needed to unlock audio context in many browsers (call `Tone.start()` on a user interaction).
- **Large uploads**: Set server limits and provide progress UI.

Example preset save/load flow

- User clicks "Save Preset" → frontend compiles state into JSON (use schema above) → triggers file download `preset.json`.

- User clicks "Load Preset" → file input reads JSON and applies state to components (update Tone.js sequences, sample choices, bpm, tuning).

Testing & dev notes

- Dev locally: run FastAPI on `http://localhost:8000`, React on `http://localhost:5173`. Configure CORS accordingly.
- For reproducible audio processing, always include sample rate in responses and re-sample on backend if needed.
- Keep an example preset and sample packs in repo so you can demo offline.