Audio Systems Learning Map (Local-First, Python-Centric)

This document is a **learning-oriented map of audio projects** worth studying if you want to design **real-time, offline, programmable audio systems**. The emphasis is on **architecture, signal flow, buffers, and extensibility**, not just end-user tools.

Python is prioritized, but strategically chosen C/C++ projects are included when they teach *core engine design* that transfers cleanly to Python bindings.

---

1. Core Mental Models (Read This First)

These concepts show up in *every serious audio system*, regardless of language or framework. Understanding them deeply will save you months.

1.1 Push vs Pull Audio Graphs

- **Pull-based**: audio driver asks for the next buffer (most common: PortAudio, JACK)
- **Push-based**: producer pushes audio downstream (less common, more fragile)

Why it matters: - Pull-based systems naturally enforce realtime constraints - Push-based systems often drift or glitch under load

Look for this concept in: - JACK client callbacks - JUCE AudioProcessor::processBlock

---

1.2 Ring / Circular Buffers

Key ideas: - Fixed-size memory - Separate read/write heads - No allocations in realtime paths

Intriguing questions: - How do you handle underruns vs overruns? - Do you drop data or block?

This concept connects audio ↔ MIDI ↔ UI safely.

1.3 Clock Domains

Typical clocks: - Audio sample clock (e.g. 48kHz) - MIDI/event clock (timestamped) - UI / render clock (vsync or timer-driven)

Key insight:

> Most bugs are *clock-crossing bugs*, not DSP bugs.

1.4 Event vs Signal Pipelines

- **Signals**: continuous (audio buffers)
- **Events**: discrete (notes, beats, UI actions)

Clean systems never confuse the two.

---

2. Audio I/O, Buffers, and Realtime Foundations

These projects teach how audio *enters and leaves* your program.

🔹 sounddevice (Python)

https://github.com/spatialaudio/python-sounddevice

Why it's interesting: - Thin wrapper over PortAudio - Exposes callback-based audio I/O

Concepts to study: - Audio callback lifetimes - Block sizes and latency - Thread priority and dropouts

Intriguing question:

> What *cannot* happen inside an audio callback?

🔹 JACK Audio (Reference Architecture)

https://github.com/jackaudio/jack2

Why it's interesting: - Explicit audio graph scheduling - Sample-accurate synchronization

Concepts to study: - Client registration - Graph execution order - Realtime safety guarantees

Even if you never run JACK, copy its *mental model*.

🔹 JACK Audio (Architecture Reference)

(Language: C, bindings everywhere)

What to extract: - Graph-scheduled audio processing - Sample-accurate synchronization - Client registration model

Even if you never use JACK directly, its *mental model* is gold.

---

3. Music Information Retrieval & Feature Extraction

These projects turn raw audio into *structured data*.

🔹 librosa (Python)

https://github.com/librosa/librosa

Why it's interesting: - Reference implementations of DSP

Concepts to focus on: - STFT windowing tradeoffs - Frame vs hop size - Phase reconstruction

Intriguing question:

> How much temporal resolution do you really need?

🔹 Essentia (C++ core + Python bindings)

https://github.com/MTG/essentia

Why it's interesting: - Descriptor graphs - Streaming vs batch analyzers

Concepts to focus on: - Feature dependency graphs - Deterministic pipelines

Think of this as a *feature engine*, not a library.

🔹 madmom (Python)

https://github.com/CPJKU/madmom

Why it's interesting: - Beat/downbeat tracking - Musical-time reasoning

Concepts to focus on: - Tempo hypotheses - Probabilistic timing

---

🔹 Essentia (C++ core + Python bindings)

Why study: industrial-grade MIR system

Extract ideas: - Descriptor graphs - Feature pipelines - Batch vs streaming analyzers

This is a *blueprint* for turning audio into structured data.

🔹 madmom (Python)

Why study: rhythm-aware systems

Study focus: - Beat/downbeat tracking - Synchronization to musical time - Probabilistic timing models

Excellent for drum trainers and sequencers.

---

4. Source Separation & Stem Architectures

These projects show how ML integrates with audio I/O.

🔹 Ultimate Vocal Remover (UVR)

https://github.com/Anjok07/ultimatevocalremovergui

Why it's interesting: - Chunked inference - Overlap-add reconstruction

Concepts to study: - Window stitching - Model orchestration

Ignore the UI first — read the processing pipeline.

🔹 Demucs

https://github.com/facebookresearch/demucs

Why it's interesting: - End-to-end waveform models - Streaming-friendly design

Concepts to study: - Receptive fields - Latency vs quality

🔹 Open-Unmix

https://github.com/sigsep/open-unmix-pytorch

Why it's interesting: - Modular ML separation

Concepts to study: - Model abstraction layers - Separation as a service

---

🔹 Open-Unmix

Why study: modular ML separation design

Extract: - Configurable stems - Model abstraction layers - Separation as a service concept

Good reference for *pluggable* ML audio blocks.

---

5. Voice, Pitch, and Performance Modeling

These systems convert sound into *control*.

🔹 CREPE

https://github.com/marl/crepe

Why it's interesting: - Neural pitch estimation

Concepts to study: - Frame confidence - Audio → control-rate conversion

---

🔹 WORLD Vocoder

https://github.com/mmorise/World

Why it's interesting: - Clean speech decomposition

Concepts to study: - Separation of pitch, timbre, noise - Deterministic synthesis

This explains how voice cloning actually works.

🔹 DDSP

https://github.com/magenta/ddsp

Why it's interesting: - Neural networks parameterize DSP

Concepts to study: - Hybrid systems - Controllability vs realism

---

🔹 WORLD Vocoder

Why study: clean speech decomposition

Extract: - Separation of pitch, timbre, noise - Deterministic synthesis

This teaches how voice cloning systems *actually work under the hood*.

🔹 DDSP

Why study: hybrid neural + DSP systems

Key insight: - Neural networks don't replace DSP — they *parameterize it*

Highly relevant if you care about controllability.

---

6. MIDI, Timing, and Event Systems

MIDI is not audio — treat it as events.

🔹 RtMidi

https://github.com/thestk/rtmidi

Why it's interesting: - Cross-platform MIDI I/O

Concepts to study: - Polling vs callbacks - Timestamping

🔹 mido

https://github.com/mido/mido

Why it's interesting: - Clean Python MIDI abstraction

Concepts to study: - Backend separation - Message routing

🔹 pretty_midi

https://github.com/craffel/pretty-midi

Why it's interesting: - MIDI ↔ time alignment

Concepts to study: - Symbolic vs real-time representations

🔹 pretty_midi

Why study: MIDI ↔ time alignment

Extract: - MIDI as structured score data - Mapping note events to wall-clock time

Excellent bridge between symbolic and signal domains.

---

7. Realtime DSP Graphs (Non-Python, Worth Studying)

🔹 SuperCollider

Why it matters: - Audio as message-passing - Explicit separation of control vs audio rate

This will change how you design Python engines.

🔹 Faust

Why it matters: - Pure functional DSP - Explicit signal graphs - Compile-time optimization

Faust teaches *what DSP code should look like before glue exists*.

🔹 JUCE

Why it matters: - Production-grade plugin architecture - Audio threading discipline

Study it for architecture, not syntax.

---

8. Visualization & Audio-UI Coupling

Recommended pairing ideas: - PyQtGraph + ring buffers - OpenGL textures fed from audio FFTs - Timeline-based waveform caches

Study goal: - Decouple rendering from audio time - UI reads snapshots, never live buffers

---

9. Suggested Learning Path (Practical)

Phase 1 — Fundamentals - sounddevice + ring buffer - librosa offline analysis

Phase 2 — Events & Timing - MIDI input + timestamps - madmom beat tracking

Phase 3 — ML Blocks - Demucs or UVR as a batch service - CREPE pitch → MIDI

Phase 4 — System Design - Audio + MIDI unified event bus - Feature extractors as producers - UI panels as observers

---

10. What to Build While Learning

Small but powerful projects:

- Realtime pitch → MIDI monitor
- Stem separation → per-stem FFT viewer
- Drum hit alignment trainer
- Audio feature recorder → Zarr store

Each one reinforces architecture, not just DSP.

---

11. Design Rule of Thumb

> If audio can block, it must not allocate. If UI can stall, it must never touch live audio. If ML is slow, treat it as an offline or chunked service.

---

12. Where This Naturally Leads

This learning map naturally supports: - Modular audio engines - AI-assisted music tools - Drum and vocal training systems - Audio-driven visualization engines - Audio-LLM hybrid pipelines

---

If you want next: - A **repo reading order** for any section - A **minimal Python audio engine skeleton** - A **unified audio + MIDI event model**

Say the word and we'll go there.