The Voice Memo Graveyard

Building a Second Brain for Musicians with Python

Pablo López Diéguez
keep-audio.xyz

PyCon TW 26

Who am I?

- **Pablo** from northwest Spain
- **Senior ML Engineer** @ Micron Technology
- **8+ years** living in Taiwan
- **Amateur songwriter** in my free time
  (guitar player, learning piano / drums)



What we'll talk about today

01 / CONTEXT
The creative problem

- The voice memo graveyard
- Song Seeds
- Live demo (keep-audio.xyz)

02 / INTUITION INTO CODE
Rules vs. AI

- Turning musical intuition into Python rules
- Similarity vs. Complementarity
- Where AI belongs (and where it doesn't)

03 / THE BORING STACK
Production trade-offs

- Architecture: FastAPI + SQLite + local Whisper
- Lessons learned: How `async def` froze the event loop

What we'll talk about today

01 / CONTEXT
The creative problem

- The voice memo graveyard
- Song Seeds
- Live demo (keep-audio.xyz)

02 / INTUITION INTO CODE
Rules vs. AI

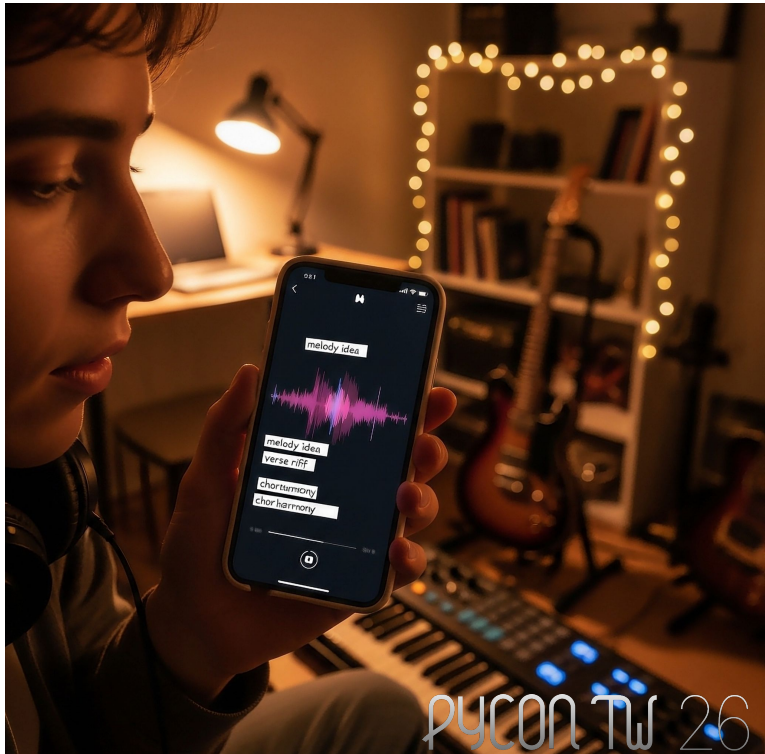
- Turning musical intuition into Python rules
- Similarity vs. Complementarity
- Where AI belongs (and where it doesn't)

03 / THE BORING STACK
Production trade-offs

- Architecture: FastAPI + SQLite + local Whisper
- Lessons learned: How `async def` froze the event loop

Musicians record ideas the moment they show up

- Ideas = Melodies, chord progressions, riffs, lyrics, etc.
- Recorded in seconds, before they disappear
- The persistent lie: "I'll come back to this later"



The ideas exist. They just never meet.

GOAL =
Make a (full) song

ANTI-GOAL =
Massive backlog of voice memos

CREATIVE BLOCK
Not knowing where to start (not a lack of ideas)

SAME PATTERN ELSEWHERE
Notes apps, bookmarked articles, photo rolls, video

Can technology help, without taking away the part I enjoy?

I didn't want:

- AI making the music (e.g. Suno)
- A black box deciding what "fits"
- A distributed system for a personal project (overengineering)

I wanted:

- Help finding a starting point
- Suggestions I can understand and argue with
- Cheap & simple pythonic stack (we love it)

How is a pop song structured?

Intro / Riff → Verse + Chorus ★ → Bridge → Chorus / Solo

01

Initial Riff

The Hook / Intro

"Seven Nation Army"
— The White Stripes

02

Verse + Chorus

Story & Climax

"Love Confession" (告白氣球)
— Jay Chou (周杰倫)

03

The Bridge

Novelty

"Shape of You"
— Ed Sheeran

04

Final Climax

Chorus / Solo

"Hotel California"
— Eagles (Outro Solo)

The answer: **Song Seeds***
and let Python do the searching

Riff + Verse / Chorus melody + Lyric line → 🌱 **Song Seed** (+ why they fit)

01

Fixed Rules

Define **"good"** with fixed rules for what makes ideas fit together

02

Targeted AI

AI strictly for **speech-to-text**

03

Minimal Stack

One Python app on a **small VM** until it breaks

Song Seed* = what a **good song start looks like**

Live Demo

keep-audio.xyz

upload idea → add tags → find matches

(🌱 Song Seed)

What we'll talk about today

01 / CONTEXT
`The creative problem`

- The voice memo graveyard
- Song Seeds
- Live demo (`keep-audio.xyz`)

02 / INTUITION INTO CODE
`Rules vs. AI`

- Turning musical intuition into Python rules
- Similarity vs. Complementarity
- Where AI belongs (and where it doesn't)

03 / THE BORING STACK
`Production trade-offs`

- Architecture: FastAPI + SQLite + local Whisper
- Lessons learned: How `async def` froze the event loop

What we'll talk about today

01 / CONTEXT
The creative problem

- The voice memo graveyard
- Song Seeds
- Live demo (keep-audio.xyz)

02 / INTUITION INTO CODE
Rules vs. AI

- Turning musical intuition into Python rules
- Similarity vs. Complementarity
- Where AI belongs (and where it doesn't)

03 / THE BORING STACK
Production trade-offs

- Architecture: FastAPI + SQLite + local Whisper
- Lessons learned: How `async def` froze the event loop

Step 1: Translating Intuition into Code

Reward



Penalty

My intuition	Rule in code
A song needs **different parts** (e.g. riff + verse + chorus)	**Reward++ complementary roles**
A song needs **different instruments**	**Reward+ different instruments**
Parts should feel **cohesive**, like the same song	**Reward** same **mood**, compatible **energy**
Verse can be quieter than chorus	Allow energy gap if verse + chorus are present Otherwise → **penalty**
Different keys are fixable, but not free	**Key mismatch** → **penalty+** (relative keys allowed)
Lyrics in **two languages** won't merge	**Language conflict** → **penalty++**

Similarity vs. Complementarity:

Embeddings find similar ideas, but a song needs different ones

Similarity (embeddings)	Complementarity (my rules)
 Riff → matches   Drums → matches   Piano → matches 	 Riff → matches   
    Groups 4 basslines	    Builds a group with distinct roles
These are **close in vector space**	Complementary **roles**, same **mood**, close **keys**

Deterministic Scoring:
The rules fit in a few lines of Python

Stage 1: Fast Candidate Pairing
Additive scoring to find candidate neighbors against target memo

```python
# app/application/services/grouping_service.py
def compute_score(idea_tags, idea, candidate_tags, candidate) -> int:
    """Deterministic complementarity score between two ideas."""
    score = 0
    if _roles_are_complementary(role(idea_tags), role(candidate_tags)):
        score += 5                          # most important: different parts
    if same(idea_tags, candidate_tags, "mood"):
        score += 2
    if energy_gap(idea_tags, candidate_tags) <= 1:
        score += 2
    if no_shared_instruments(idea_tags, candidate_tags):
        score += 3
    score += min(candidate.play_count or 0, 1)   # usage: tiebreaker only
    return score
```

Stage 2: Seed Validation & Penalties
Evaluates the 3-4 ideas as a musical composition

```python
# 1. Structural context: energy gap is OK IF verse + chorus create an arc
if energy_gap <= 2 and ("verse" in roles and "chorus" in roles):
    energy_score = 0.9  # natural dynamic contrast

# 2. Multiplicative penalty layer for objective musical conflicts
if key_mismatch and not relative_keys:  penalty *= 0.75  # needs transposition
if multiple_vocal_languages:            penalty *= 0.50  # incompatible lyrics

final_score = affinity_score * penalty  # penalties scale the whole seed down
```

Explainability & no-black-box:
Every suggestion comes with its reasons

Example 1:

MATCH RATIONALE — CONFIDENCE: **93%**

- Good role complementarity: riff + verse
- Consistent energy across all ideas: low
- Strong mood coherence: melancholic ×2

Example 2:

MATCH RATIONALE — CONFIDENCE: **68%** — PENALTY: -25%

- Good role complementarity: riff + verse
- Consistent energy across all ideas: low
- Strong mood coherence: melancholic ×2
- Close keys (D#, D) — minor transposition likely needed
- Key mismatch (D, D#) — transposition likely needed

Applying AI strictly where rules cannot be written

Raw Perception

Turn sung audio into text

Whisper (local base model) for speech-to-text

```python
from app.infrastructure.ai.whisper import transcribe
from app.infrastructure.db.session import SessionLocal
from app.infrastructure.db.models import IdeaModel
from datetime import datetime, timezone

def process_idea(idea_id: str):
    with SessionLocal() as db:
        idea = db.query(IdeaModel).get(idea_id)
        if not idea:
            return

        try:
            idea.status = "PROCESSING"
            db.commit()

            result = transcribe(idea.audio_path)

            idea.transcription = result["text"]
            idea.detected_language = result["language"]
            idea.status = "DONE"
            idea.last_processed_at = datetime.now(timezone.utc)
            db.commit()

        except Exception as e:
            idea.status = "ERROR"
            idea.last_processed_at = datetime.now(timezone.utc)
            db.commit()
            raise e
```

```python
import whisper

_model = None

def get_model():
    global _model
    if _model is None:
        _model = whisper.load_model("base")

    return _model

def transcribe(audio_path: str, language: str = None) -> dict:
    model = get_model()

    # Check if we can load audio first to avoid cryptic tensor errors
    audio = whisper.load_audio(audio_path)
    if audio.size == 0 or len(audio) == 0:
        raise ValueError(f"Audio file is empty or unreadable: {audio_path}")

    # Use provided language or auto-detect
    options = {"task": "transcribe"}
    if language:
        options["language"] = language

    result = model.transcribe(
        audio, # Pass loaded audio directly
        **options
    )

    detected_lang = result.get('language', language or 'unknown')

    return {
        "text": result["text"],
        "language": detected_lang
    }
```

Applying AI strictly where rules cannot be written



Raw Perception

Turn sung audio into text

Whisper (local base model) for speech-to-text



Explainable Decisions

Python rules for matching ideas taste



Creative Work

The **human** writes the song

What we'll talk about today

01 / CONTEXT
The creative problem

- The voice memo graveyard
- Song Seeds
- Live demo (keep-audio.xyz)

02 / INTUITION INTO CODE
Rules vs. AI

- Turning musical intuition into Python rules
- Similarity vs. Complementarity
- Where AI belongs (and where it doesn't)

03 / THE BORING STACK
Production trade-offs

- Architecture: FastAPI + SQLite + local Whisper
- Lessons learned: How `async def` froze the event loop

What we'll talk about today

01 / CONTEXT
The creative problem

- The voice memo graveyard
- Song Seeds
- Live demo (`keep-audio.xyz`)

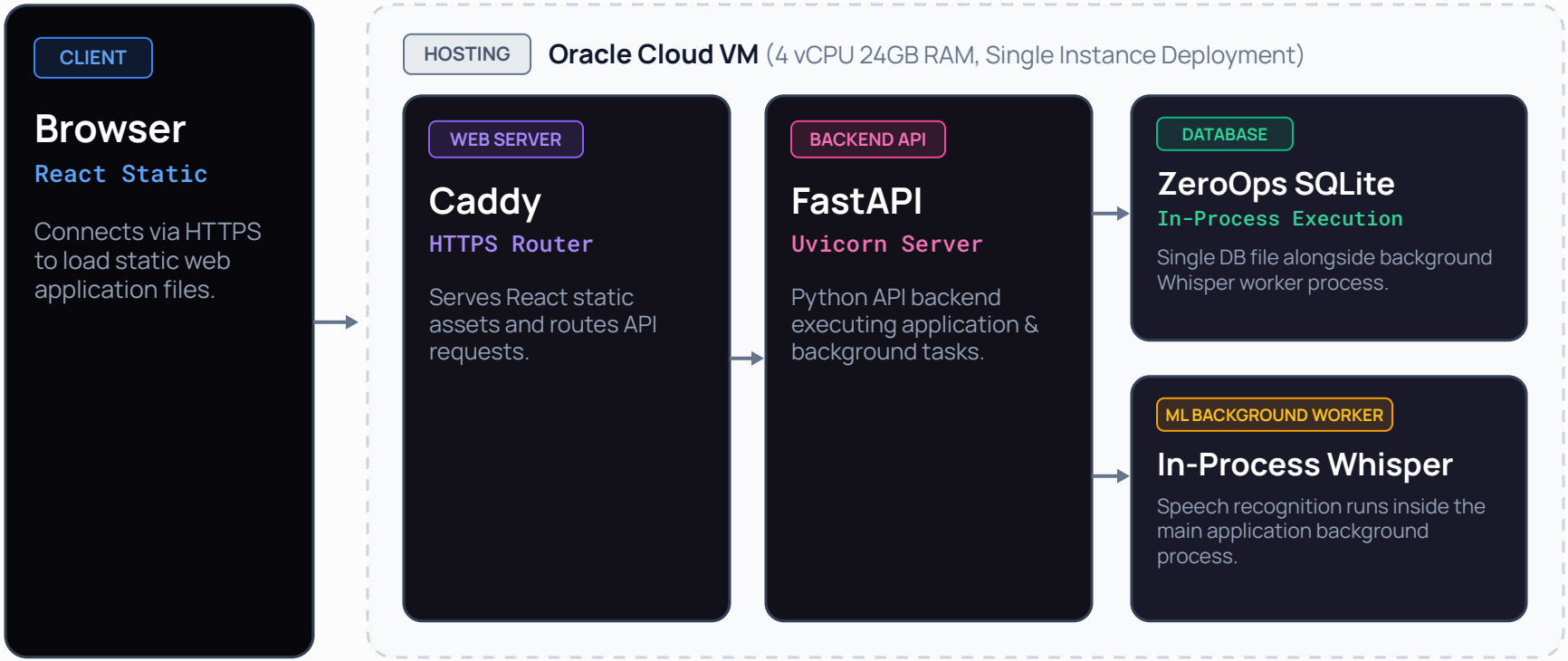
02 / INTUITION INTO CODE
Rules vs. AI

- Turning musical intuition into Python rules
- Similarity vs. Complementarity
- Where AI belongs (and where it doesn't)

03 / THE BORING STACK
Production trade-offs

- Architecture: FastAPI + SQLite + local Whisper
- Lessons learned: How `async def` froze the event loop

The whole thing runs on one small VM



The **Incident**: An audio upload 🥶 froze the entire site for 10s

“I wrote `async def` because `async` means **fast… right**?”

What happened:

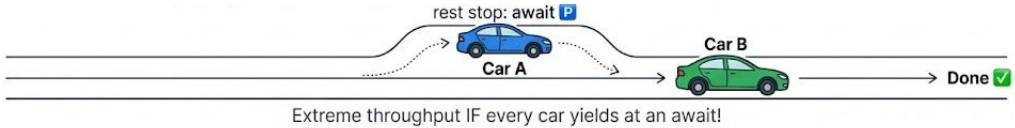
- Unrelated endpoints stopped responding.
- Question:

  “FastAPI is an **async** framework. Why did adding `async def` make my server freeze?”

```python
@router.post("/")
async def create_idea(background_tasks: BackgroundTasks, file: UploadFile = File(...), ...):
    file_bytes = await file.read()          # ① whole upload (up to 20 MB) copied into RAM
    idea_id = service.create_idea(          # ② blocking code, running ON the event loop:
        file_bytes=file_bytes, ...)         #    write file to disk + SQLite commit
    background_tasks.add_task(process_idea, idea_id)   # ③ Whisper: handed off (next slide)
    return {"id": idea_id}
```

The Mental Model: Single-Lane Express vs. Multi-Lane Highway

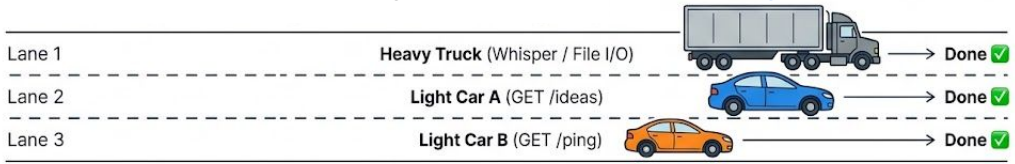
1. The **Event Loop (asyncio)** = A Single-Lane Express Road



2. **Accidental Blocking** inside `async def` (the Single-Lane Express)



3. The **Thread pool** (blocking `def`) = A Multi-Lane Highway



FastAPI's Routing Rule

You write	FastAPI runs it on…
`async def`	the **Event Loop** via `asyncio` It relies on `await` to yield!
plain `def`	a **worker thread pool** via `anyio`

scheduled to: Single-Lane

scheduled to: Multi-Lane

The Fix & The Decision Guide:
When to use `async def` vs. `def`

Core Rule:
Never block the **Event Loop**
with **synchronous code**.

🥶 *Before (the blocking version)*

```python
@router.post("/")
async def create_idea(background_tasks: BackgroundTasks, file: UploadFile = File(...), ...):
    file_bytes = await file.read()        # ① whole upload (up to 20 MB) copied into RAM
    idea_id = service.create_idea(        # ② blocking code, running ON the event loop:
        file_bytes=file_bytes, ...)       #    write file to disk + SQLite commit
    background_tasks.add_task(process_idea, idea_id)   # ③ Whisper: handed off (next slide)
    return {"id": idea_id}
```

✅ *After the fix*

```python
@router.post("/")
def create_idea(background_tasks: BackgroundTasks, file: UploadFile = File(...), ...):
    idea_id = service.create_idea(file_obj=file.file, ...)   # streams from the temp file, no big RAM copy
    background_tasks.add_task(process_idea, idea_id)         # runs AFTER the response is sent
    return {"id": idea_id}                                   # fast: status = PENDING
```

Category	Typical Libraries	Signature
Native Async I/O	Non-blocking OS calls that yields at `await`: `httpx`, `asyncpg`, `aiosqlite`	`async def`
Sync / Blocking I/O	Blocking OS calls: `open()`, `sqlite3`, `requests`	plain `def`
CPU-Bound Compute	heavy CPU math: `whisper`, `torch`, `opencv`	
Not Sure	Calling sync helpers or legacy code	

What to take back to your own project

[Domain]

1. **Complementary and similar are different problems**:
   Check which one your problem needs.
2. **Try rules before complex ML models**:
   You get deterministic behavior and **explainability for free**, especially when a decision affects people.

[Engineering]

3. **Stick to a simple architecture until it breaks**: A single cheap VM with FastAPI and SQLite can take you shockingly far.
4. If you're not sure whether your code blocks, **fall back to plain `def` over `async def`.**

[Mindset]

5. **Define your anti-goals early**:
   Decide what you won't build, and leave the enjoyable parts of the craft to humans.

Thank You. Questions?



Common questions

Couldn't ML find complementarity better than your rules?

Possibly, with a large dataset of parts that belong together, for example multitrack stems, you could train a model to learn complementarity. I don't have that data as a solo developer, and even if I did, I'd lose the explanation. For a creative tool, the musician being able to read why two ideas match and disagree with it is worth more than a slightly better score.

Why not Python 3.13 free-threading?

The project runs on 3.12. Free-threading (PEP 703) could change how threads behave for CPU-bound Python code, but here the heavy work is already in C++ that releases the GIL, so the main win would be for the pure-Python parts. I'd test it before relying on it.

Isn't manual tagging a lot of friction?

It's about ten seconds per idea, and it's done when I revisit an idea, not when I record it. Recording stays instant. Auto-tagging is a reasonable next step, for example detecting key and tempo from audio with signal processing. That would be a model, or DSP, filling in inputs, while the matching decision stays rule-based.

How do you tune the weights?

By listening. When a suggestion feels wrong, I read the explanation, find the rule responsible and adjust it. It's slow, but each change is a decision I understand.

Why not just use an LLM to suggest groupings?

It would give me plausible text and a different answer each time. I wanted the same input to give the same output, and a reason I can trace to a line of code.