# DAY -1

**🔧 What is uv?**

uv is a next-gen Python package manager from Astral. It’s:

* ⚡️ **Ultra fast** (written in Rust)
* ✅ Compatible with requirements.txt or pyproject.toml
* 🧼 Replaces pip, venv, pip-tools, virtualenv, poetry, etc.
* 💻 Used internally at OpenAI

**📦 What is pyproject.toml?**

pyproject.toml is the **modern standard configuration file** for Python projects.  
Think of it like the package.json of Python.

It defines:

* Your **project metadata** (name, version, description)
* Your **dependencies**
* Your **build system**
* Your **tooling config** (e.g., for uv, black, ruff, pytest, etc.)

Here’s a clean summary of everything we’ve done so far — you can paste this directly into your notes:

**✅ Step 2: Production-Grade Python Environment Setup (with uv and pyproject.toml)**

**1. 📦 Installed uv**

Used uv as our modern, fast dependency and virtualenv manager.

Install command:

curl -Ls https://astral.sh/uv/install.sh | sh

**2. 🧪 Created .venv using uv**

Created a virtual environment **inside the project folder** (production-grade approach):

uv venv

source .venv/bin/activate

**3. 🛠️ Created pyproject.toml**

Defined all runtime and dev dependencies in one clean config file.

**Runtime dependencies:**

* fastapi
* uvicorn[standard]
* sounddevice
* numpy
* pydantic

**Dev tools:**

* black → autoformatter
* ruff → linter/static checker
* pytest → testing framework

[project]

name = "scribe-ai"

version = "0.1.0"

description = "Real-time voice-to-note AI scribe"

readme = "README.md"

requires-python = ">=3.9"

dependencies = [

"fastapi",

"uvicorn[standard]",

"sounddevice",

"numpy",

"pydantic"

]

[project.optional-dependencies]

dev = [

"black",

"ruff",

"pytest"

]

[tool.uv]

virtualenvs.in-project = true

[tool.black]

line-length = 88

[tool.ruff]

line-length = 88

target-version = "py39"

select = ["E", "F", "I"]

[tool.pytest.ini\_options]

addopts = "-ra -q"

testpaths = ["tests"]

**4. ⚙️ tool.\* in pyproject.toml**

These are **config sections** for dev tools:

* tool.black → sets line length for autoformatting
* tool.ruff → sets linter rules for unused imports, formatting, etc.
* tool.pytest → controls pytest behavior
* tool.uv → makes sure .venv/ is local to the project

**5. 📥 Installed all dependencies (main + dev tools)**

uv pip install .[dev]

This installs everything:

* Runtime dependencies from [project.dependencies]
* Dev tools from [project.optional-dependencies].dev

A **WebSocket** is a special communication protocol that allows **real-time, two-way interaction** between a **client** (like a web browser or mobile app) and a **server**.

Unlike normal HTTP requests (which are one-way: client asks, server replies), WebSockets keep the connection **open**, so **both** sides can send data anytime — perfect for **live updates** like speech apps, chat apps, games, or notifications.

🧠 Think of it like a **phone call**, not sending letters:

* HTTP = letter-based communication (slow, one-way)
* WebSocket = live phone call (fast, two-way)

In a **speech app**, you need real-time interaction. Here’s why HTTP isn't enough:

* **HTTP**: Every time you want to send data (like a chunk of audio), you must open a new connection, send it, and close it. That creates **delays** and **overhead**.
* **WebSocket**: You open **one connection**, and then you can stream data **continuously** in both directions — **perfect for sending live audio and getting live transcripts** back without waiting.

💡 For example:

* Your app sends audio chunks as the user speaks (WebSocket → Server).
* Server replies with transcribed text instantly (Server → WebSocket).

This **back-and-forth flow** is what makes speech feel live

Your **FastAPI app will act as a WebSocket server**. That means:

* You define a **special route** like /ws/audio-stream
* But unlike REST routes (@app.get, @app.post), you use @app.websocket
* This route does **not** expect a request → response flow — it stays open

📌 Inside this WebSocket route:

* You’ll **wait for audio chunks** from the client (await websocket.receive\_bytes() or .receive\_text()).
* You **process** each chunk (e.g., speech-to-text using Whisper/OpenAI).
* You **send back transcription** using await websocket.send\_text().

🔹 Step 6: What is WebRTC? (And how is it different from WebSocket?)

WebRTC (Web Real-Time Communication) is a protocol for direct peer-to-peer communication between browsers or devices. It's used for:

🔊 Voice calls (e.g., Google Meet, Zoom)

🎥 Video streaming

📁 File sharing (without server)

🧠 WebRTC is optimized for media (audio/video) and includes:

Echo cancellation

Latency control

NAT traversal (to go through firewalls)

Encryption (by default)

**At the simplest level, a mic is just a hardware device that listens to sound and converts it into a digital signal your computer or microcontroller can understand.**

So we need to understand the hardware setup better and also start coding to use to laptop mike as a starting point

The task is nice since we need a device that will capture audio.

**So we need a speaker phone**

<https://www.amazon.in/Raspberry-Model-LAN-mHDMI-Supply-MicroSD/dp/B0C7N5DJC1?ref_=v_sp_product_dpx&th=1>

so I have ordered this I think this would be cool to play around and learn , so that we do and cook amazing stuff …intersting times ahead

Note – A design choice has been made for now we need to learn rasbeery pie first and then we need to play the device bit of it

But we need to cook the v1 for system a doctor can open the app and listen start processing and send notes to self and patient

# DAY -2

Started with a small recap on design I mean static method and class method were used in one project just saw that , back to work now.

So feature 1 will be something like this

**🩺 Feature 1: Patient Follow-Up on Telegram (Post-Consultation)**

**🎯 Goal:**

Allow patients to **ask doubts or follow-up questions** on Telegram after their consultation, and **get instant, personalized replies** — powered by your system's backend intelligence.

**🧩 System Flow:**

1. **Session Completed by Doctor**
   * The doctor records and completes a session using your app.
   * Your backend transcribes the audio and stores it with patient metadata (e.g., name, phone number, session ID, summary).
2. **Patient Gets Summary**
   * A message is sent to the patient on Telegram via your bot with:
     + The summary of the consultation.
     + Advice, prescriptions, red flags.
     + A note: “You can ask your follow-up questions here anytime.”
3. **Patient Sends a Message**
   * The patient types a question on Telegram, like:
     + “Can I take this medicine after food?”
     + “I still have a sore throat, should I be concerned?”
4. **Your Backend Handles It**
   * Identifies patient using Telegram ID or linked phone number.
   * Retrieves the session summary + prescription + red flags.
   * Feeds all of that + patient’s question into an LLM API.
   * Receives a smart, patient-friendly answer.
5. **Bot Replies Back**
   * Sends the response to the patient on Telegram.
   * Ends with a safety disclaimer (e.g., “This is an AI response. Please call your doctor if symptoms worsen.”)

**🏗️ 1. High-Level System Design Overview**

**🔊 Input**

* Mobile app or web client (using WebRTC or WebSocket) for capturing audio from doctor (or both doctor & patient).
* Audio streamed to backend **in real-time**.

**⚙️ Backend Core (FastAPI)**

* Audio ingestion via **WebSocket**.
* Realtime transcription (e.g. **OpenAI Whisper** or **Deepgram**).
* LLM summary + prescription generation.
* Message delivery via **Telegram API** or WhatsApp Business API.

**☁️ Infrastructure**

* Host on **Azure Cloud** with:
  + Azure App Service or AKS for backend
  + Blob Storage for audio
  + CosmosDB / MongoDB for session data
  + Azure Queue for async processing
  + Redis for temporary memory/cache

**📶 2. Streaming Audio Design**

| **Component** | **Tech** |
| --- | --- |
| Audio Source | Browser/Mobile Mic |
| Stream Protocol | WebSocket (or WebRTC) |
| Format | audio/webm or wav chunks |
| Transcription | OpenAI Whisper API or Deepgram Streaming API |
| Buffering | Circular buffer in Python / Redis |
| Resilience | Auto-reconnect, retries, chunk validation |

**🧠 3. AI/LLM Processing**

1. **Transcription**
   * POST /transcribe auto-triggered after stream ends.
   * Or done in real-time for streaming APIs like Deepgram.
2. **LLM Summary**
   * Context-aware summarization using OpenAI (GPT-4-turbo).
   * Use custom system prompts (e.g., SOAP-style outputs).
3. **Prescription Generator**
   * Separate LLM prompt/chain to generate:
     + Medicines
     + Dosage
     + Follow-ups
     + Investigations

**📦 4. Scalable Architecture (Cloud-native)**

* **App Server**: FastAPI behind **Gunicorn** + **Uvicorn workers**.
* **Message Queue**: Azure Queue or RabbitMQ for:
  + LLM jobs
  + Transcription
  + Messaging jobs
* **Worker Pool**: Celery workers or FastAPI background tasks.
* **Storage**:
  + Audio files → Azure Blob Storage
  + Summaries → MongoDB or CosmosDB
* **Cache**: Redis (for user session / intermediate transcripts)

**📲 5. Messaging + Telegram Integration**

* One-time bot setup per doctor.
* Transcripts sent via Telegram using sendMessage API.
* Patients can reply → webhook hits POST /telegram/receive
* Use RAG or memory DB to answer follow-ups.

**🧪 6. Testing & Monitoring**

* **Logging**: Use structlog or loguru for production logs.
* **Monitoring**: Azure Monitor or Grafana.
* **Rate Limiting & Auth**:
  + Use API Keys / JWT for doctor access.
  + Rate limit audio ingestion endpoints.

**✅ 7. Security**

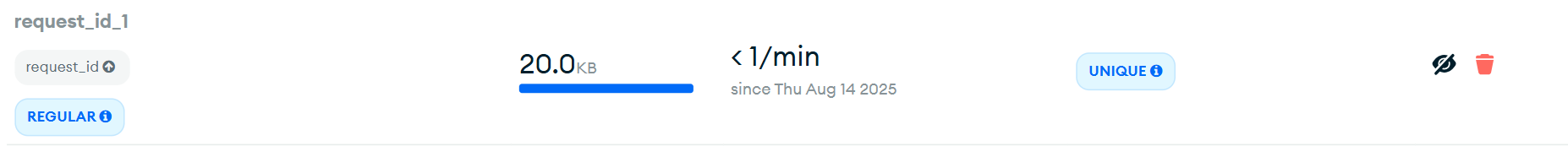
* Use HTTPS/WSS for all communication.
* Audio data encrypted in transit and at rest.
* OTP-based access for doctors via phone or email.
* Patient data stored in compliance with basic HIPAA/GDPR principles.

**⏳ 8. Scalability Notes**

| **Feature** | **Scalable With** |
| --- | --- |
| Audio | Chunked upload, queue buffering |
| LLM Processing | Async via Celery / Queue |
| Messaging | Rate-limited, async |
| Frontend | SSR React/Next.js or Streamlit-lite |
| Logging | Logstash / Azure Monitor |

**Todo: we need this product on whats app I do not know how get a whats app business number register and see how u can cook it**

**So the prime concern is user push message we need to do using what’s app.**



**What we did**

1. Clear schemas (Pydantic v2)

* SessionCreateRequest (input), SessionDB (storage), SessionResponse (output)
* Used model\_dump / model\_validate everywhere (v2 style)

1. Strict fields

* SessionStatus enum: active | closed | archived (no free-form strings)

1. Correct timestamps

* utc\_now() returns tz-aware UTC
* Stored created\_at and updated\_at in BSON datetime

1. Clean inserts & reads

* Insert with exclude\_none=True (no null spam)
* Mongo reads exclude "\_id" so Pydantic doesn’t choke

1. Idempotent create

* Client sends request\_id (UUID) in body
* Code checks for existing request\_id and returns existing session
* request\_id stored in SessionDB

1. DB guarantee

* Unique index on request\_id (sparse, unique) → race-safe idempotency
* Unique index on session\_id for fast lookups

1. HTTP correctness

* POST /v1/sessions/create returns 201 Created
* Sets Location: /v1/sessions/{session\_id}

1. Robust handlers

* Centralized log\_error on failures
* Health check route

Why this is important (quality + longevity)

* Data integrity: Enums, UTC datetimes, and clean dumps stop silent corruption.
* Consistency across services: Typed models + DB constraints mean any producer/consumer sees the same shape.
* Idempotency = real-world reliability: Retries (mobile flaky networks, timeouts) don’t create duplicates.
* Race-condition safety: Unique index turns logic guarantees into DB guarantees.
* **Maintainability: Separate request/DB/response models keep concerns clean and future changes localized.**
* **Observability & auditing: updated\_at, consistent IDs, and structured errors make issues traceable.**
* **Standards compliance: 201 Created + Location plays nicely with clients, SDKs, and tooling.**

**Quick checklist (you’re done ✅)**

* **Pydantic v2 used correctly**
* **Enum status**
* **UTC created\_at/updated\_at**
* **exclude\_none=True on insert**
* **Exclude "\_id" on read**
* **Idempotency via request\_id**
* **Unique sparse index on request\_id**
* **201 Created + Location header**
* **Error logging + health route**

# DAY -3

1.Enforced schema validation at schema layer also

2.so for now skiping transtions check and ttl implemenation at database level,

3.so design will be ->

**Doctor Session: Doctor creates session → Records audio → Real-time transcription → Audio stored in Blob → Transcript chunked → Embeddings created →**

**Chunks stored in MongoDB with vectors → Session completed → Patient notified via WhatsApp Patient Query: Patient asks question via WhatsApp → Query embedding created → Vector search in MongoDB (filter by session\_id) → Relevant transcript chunks retrieved → LLM generates response with context → Reply sent to WhatsApp**

**sessions: {session metadata}**

**transcript\_chunks: {**

**session\_id,**

**patient\_whatsapp\_number, // for filtering**

**chunk\_text,**

**embedding: [vector],**

**timestamp,**

**chunk\_index**

**}**

**conversations: {whatsapp message history}**

**🎯 FEATURE OVERVIEW**

**Core Feature: Real-time audio recording with live transcription during doctor consultations**

**User Journey: Doctor opens existing session → Clicks "Start Recording" → Speaks during consultation → Sees live transcript appear → Clicks "Stop Recording" → Session automatically completed with full transcript stored**

**✨ KEY FEATURES**

**Feature 1: Recording Session Control**

* **Start Recording - Initiate recording for a specific session**
* **Stop Recording - End recording and finalize transcript**
* **Recording Status - Check if session is currently recording**
* **Session Validation - Ensure only one recording per session**
* **Error Recovery - Handle recording failures gracefully**

**Feature 2: Real-Time Audio Streaming**

* **Live Audio Transmission - Stream audio from mobile to server in real-time**
* **Audio Chunking - Process audio in small segments for responsiveness**
* **Connection Management - Maintain stable WebSocket connections**
* **Auto-Reconnection - Recover from network interruptions automatically**
* **Buffer Management - Handle audio data efficiently without memory issues**

**Feature 3: Live Transcription**

* **Real-Time Processing - Convert audio to text as doctor speaks**
* **Partial Updates - Show transcript updates immediately on mobile screen**
* **Final Correction - Replace rough live transcript with accurate final version**
* **Multi-Language Support - Handle different languages if needed**
* **Quality Optimization - Balance speed vs accuracy for medical context**

**Feature 4: Audio Storage & Management**

* **Secure Storage - Save audio files to encrypted cloud storage**
* **File Organization - Organize audio by session with proper naming**
* **Metadata Tracking - Store duration, format, and quality information**
* **Access Control - Ensure only authorized users can access recordings**
* **Cleanup Policies - Automatic deletion of old audio files per compliance**

**Feature 5: Session Integration**

* **Status Updates - Update session status throughout recording lifecycle**
* **Transcript Storage - Save final transcript to session record**
* **Timeline Tracking - Record start/stop times and duration**
* **Data Integrity - Ensure all session data remains consistent**
* **Completion Workflow - Automatically mark session as complete when recording ends**

**🔄 FEATURE WORKFLOW**

**Pre-Recording Phase**

1. **Doctor selects existing session from list**
2. **System validates session is ready for recording**
3. **Mobile app requests microphone permissions**
4. **Server prepares recording infrastructure**

**Recording Initiation**

1. **Doctor taps "Start Recording" button**
2. **System updates session status to "recording"**
3. **WebSocket connection established for audio streaming**
4. **Mobile begins capturing and streaming audio**
5. **Real-time transcription pipeline activated**

**Active Recording Phase**

1. **Audio continuously streams from mobile to server**
2. **Live transcription appears on doctor's screen in real-time**
3. **Audio chunks saved incrementally to cloud storage**
4. **System monitors connection health and quality**
5. **Error recovery mechanisms active for interruptions**

**Recording Completion**

1. **Doctor taps "Stop Recording" button**
2. **Final audio processing and complete file assembly**
3. **High-quality final transcription generated**
4. **Session status updated to "completed"**
5. **All recording data properly stored and indexed**

**Post-Recording**

1. **Complete transcript available in session record**
2. **Audio file securely stored with proper metadata**
3. **Session ready for patient notification workflow**
4. **Recording data prepared for future RAG queries**

# DAY -4

So we have started working on transcription engine, we need to do some planning in order to see what happens and how.

 MongoDB → session + metadata + transcription text.

 Blob Storage → raw audio chunks.

 WebSocket → keeps the pipe open for real-time flow

# DAY -5

Issues Identified in Web socket implementation-

**Biggest gaps / risks (and quick fixes)**

1. **Blocking WS while transcribing.**  
   receive() → transcribe → respond means the client can’t send the next chunk until the previous finishes. Move transcription to a background worker (Celery/Dramatiq) and ACK storage immediately; push transcript\_update later.

**My comments -**

**Yes u r right.**

We will use asyncio.create\_task() so that it runs in background and we can keep receiving chunks of audio

* Concurrent processing = no guaranteed order

So we need to create a buffer and make sure we send eveything in an particular order.

Even this way u won’t be able to move ahead and do stuff since u need to understand 100 users ,what will happen u will have a bottle neck both cpu and rate limits as well.

So the solution is using async with background tasks , ensuring limits are being setup with Semaphore

Enusring we have a response buffer

* non-blocking WebSocket

1. **Sequence handling & resumes.**  
   You track expected\_sequence in memory and reset to 0 on reconnect; DB already knows what you’ve stored. On WS connect, **compute next expected** from audio\_chunks and reject duplicates/out-of-order with an explicit error. Also enforce that the **binary** frame’s sequence matches the most recent metadata.
2. **Race conditions on transcript append.**  
   update\_session\_transcript reads → concatenates → writes; parallel chunks can interleave. Prefer **per-chunk transcripts** (array by sequence) and build the full transcript on read/finalize, or use a single atomic update (e.g., aggregation-pipeline update) with sequence guards.
3. **Status transitions not fully used.**  
   You define ending but never set it; /end jumps straight to completed. Consider ending when the doctor clicks end, finalize transcripts, then completed. Also make /end **idempotent** (200 if already completed).
4. **Auth & abuse controls.**  
   WS accepts any caller with a session id. Add **JWT / API-key** auth, origin checks, size/timeouts, and per-session rate limits. Enforce max\_chunk\_size\_bytes you advertise in ConnectionConfirmed.
5. **Model mismatch in comments.**  
   Comment says “gpt-4o-transcribe” but code calls whisper-1. Align comments, or parameterize the model name.

**Fixed**

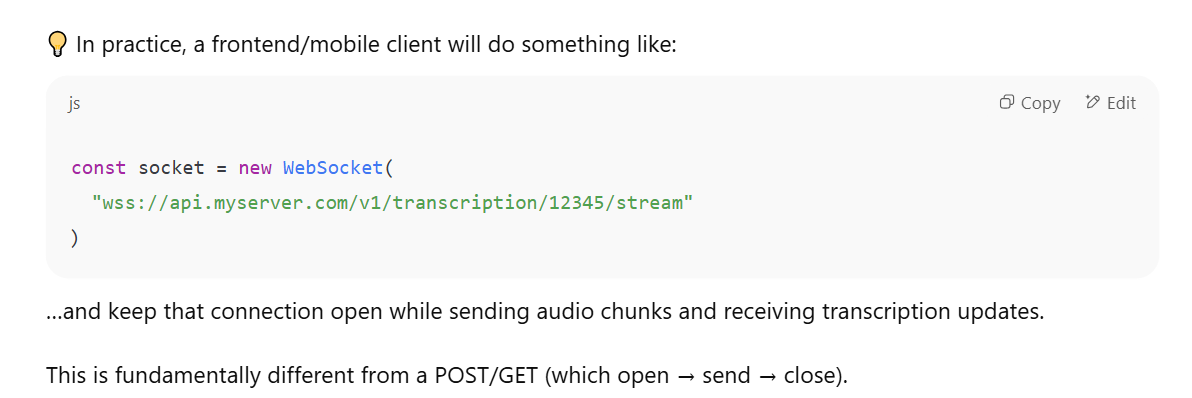
1. **Health check heaviness.**  
   models.list() and list\_containers() can be slow/privileged; cache or gate with an env flag, and avoid doing this on hot paths or tight probes.
2. **modified\_count checks can be misleading.**  
   If fields already had those values, modified\_count is 0 even though the write matched. Consider checking matched\_count (and errors) instead.
3. **Content type and container setup.**  
   You upload .webm with audio/webm; if you rely on automatic codec inference (Opus), document it and ensure the mobile side always sends Opus. Also ensure the container audio-chunks exists at boot.

# DAY -6

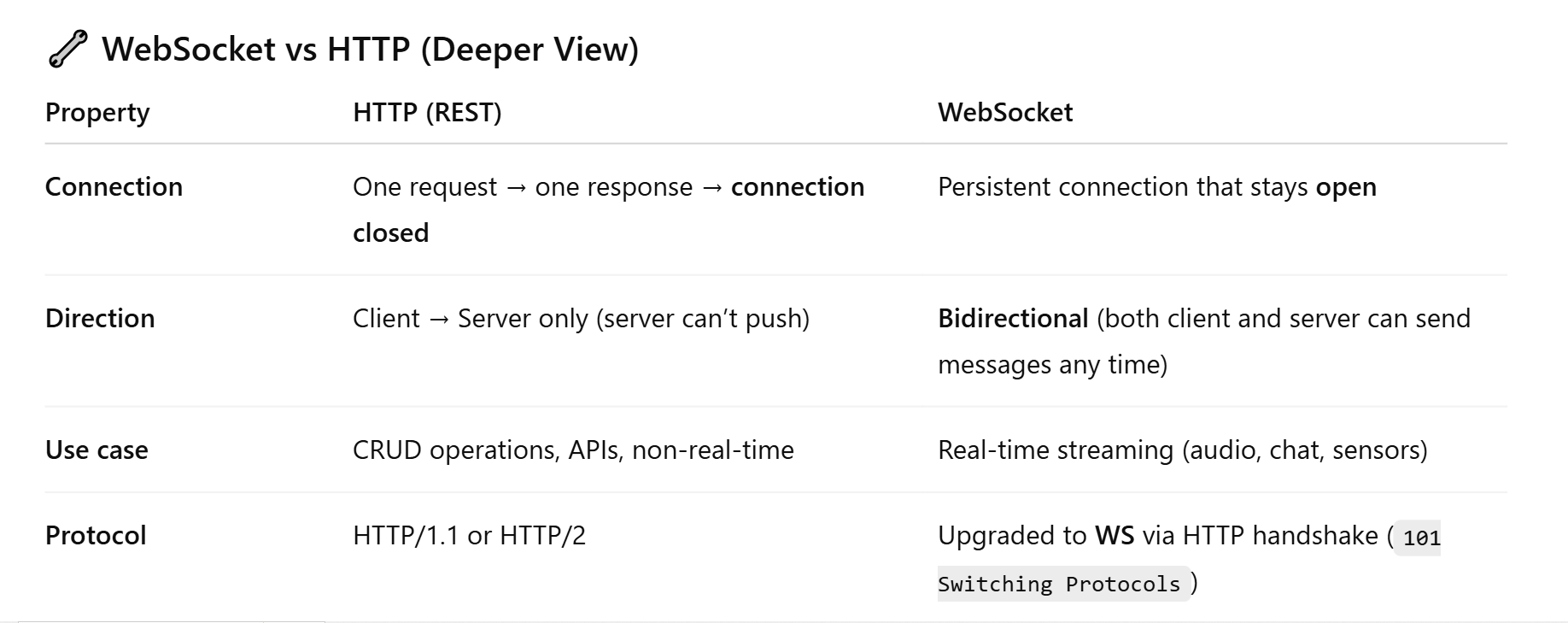
# DAY -7

Mostly need to understand the transcription system and improve it.

1. In FastAPI, WebSockets are used for **real-time bidirectional communication** (persistent connection), unlike HTTP which is **request/response and stateless**.

2. 

3.



1. **await validate\_websocket\_connection(transcription\_session\_id)**

so first we will validate websocket connection mainly that is it a valid session\_id on which we can do transcription

|  |
| --- |
| await websocket.accept() |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| **Important** → this performs the WebSocket **handshake** and confirms the connection. If you don’t call .accept(), the client’s connection will be immediately closed. |

 await mark\_websocket\_connected(transcription\_session\_id)

This will mark status to streaming

 # 4. Send connection confirmation to mobile app

        confirmation = ConnectionConfirmed(

            transcription\_session\_id=transcription\_session\_id

        )

        await websocket.send\_text(confirmation.model\_dump\_json())

Now we r confirm and sending a confirmation to the client , that hey websocket is conncted.

 # 5. Keep connection alive and process messages

        expected\_sequence = 0  # Track expected chunk sequence

        active\_tasks = {}  # Track background processing tasks

        response\_buffer = {}  # Buffer for out-of-order responses

        next\_sequence\_to\_send = [0]  # Mutable reference shared across tasks

        while True:

            # Wait for messages from mobile app

            message = await websocket.receive()

            if "text" in message:

                # Text message (JSON metadata)

                import json

                try:

                    json\_data = json.loads(message["text"])

                    response = await process\_websocket\_message(transcription\_session\_id, json\_data)

                    await websocket.send\_text(json.dumps(response))

                except json.JSONDecodeError:

                    error\_response = {

                        "type": "error",

                        "error\_code": "INVALID\_JSON",

                        "error\_message": "Invalid JSON format"

                    }

                    await websocket.send\_text(json.dumps(error\_response))

            elif "bytes" in message:

                # Binary message (audio data) - START BACKGROUND TASK INSTEAD OF BLOCKING

                audio\_data = message["bytes"]

                # Create background task with semaphore control (rate-limited)

                task = asyncio.create\_task(

                    process\_audio\_chunk\_with\_semaphore(

                        response\_buffer,

                        next\_sequence\_to\_send,

                        websocket,

                        transcription\_session\_id,

                        expected\_sequence,

                        audio\_data

                    )

                )

                # Store task reference (cleanup handled in background function)

                active\_tasks[expected\_sequence] = task

                expected\_sequence += 1  # Increment for next chunk

            else:

                # Unknown message type

                error\_response = {

                    "type": "error",

                    "error\_code": "UNKNOWN\_MESSAGE\_TYPE",

                    "error\_message": "Message must be text or binary"

                }

                await websocket.send\_text(json.dumps(error\_response))

Very important code piece->

*read incoming messages* → *identify their type* → *dispatch to the appropriate function*  
while **not blocking** the loop (so that more chunks can be received while previous ones are still being processed).

**1️⃣ expected\_sequence = 0**

Every audio chunk that arrives from the client needs a **sequence number** (0,1,2,…).  
We set expected\_sequence to 0 because the **very first** chunk we receive will be assigned sequence number **0**.

So this variable is just a **counter** that increments after each incoming audio message.

**2️⃣ active\_tasks = {}**

When audio bytes arrive, we **don’t** process them in the loop (that would block).  
Instead, we launch a **background task** using asyncio.create\_task(...).

Those tasks run *in parallel*, and we store them like this:

active\_tasks = {

0: <Task object for chunk 0>,

1: <Task object for chunk 1>,

...

}

This helps if later we want to:

* cancel tasks on disconnect,
* check which ones are still running,
* etc.

So this is basically a **task registr**

**Step 1: What is a Coroutine in Python (async/await)?**

A **coroutine** is like a special kind of function in Python that can **pause itself** (await) and let other code run in the meantime.

A coroutine **does not run immediately** when called.  
Instead, it gives back a “coroutine object” (like a promise of work), and you need to either:

* await it (run it and wait for result), or
* wrap it in a **task** to let it run in the background.

**Step 2: The Event Loop (the engine behind async)**

Think of the **event loop** as the **manager** in charge of scheduling coroutines.

* Python keeps one **event loop** running (per thread).
* The event loop decides:
  1. Which coroutine is ready to run
  2. Which coroutine is waiting (e.g., on await asyncio.sleep() or I/O)
  3. It keeps switching between them super fast → this gives the feel of *concurrent execution*.
*  A coroutine starts
*  When it hits an await (like waiting for sleep, network, DB call), it **yields control back**
*  The event loop says: “Cool, you’re waiting, let me run someone else meanwhile”
*  Another coroutine can now run instead of sitting idle
*  Later, when the first one’s waiting is done, the event loop resumes it

#### 3️⃣ response\_buffer = {}

Important:  
Background tasks finish **out of order** (chunk 3 might finish before chunk 2 depending on network/processing time).

We **can’t** send the responses in the wrong sequence — the client expects 0,1,2,3 in order.

So we use response\_buffer as a temporary staging area where each task puts its result when it finishes:

response\_buffer = {

0: { …response for chunk 0… },

2: { …response for chunk 2… }

}

Then another function (send\_buffered\_responses) looks at this buffer and sends **only** if the **next expected** sequence is available.  
(If sequence 1 is missing, it will wait even if 2 and 3 are ready.)

So: **response\_buffer == “waiting room for out-of-order responses”**.

**4️⃣ next\_sequence\_to\_send = [0]**

This one is subtle.

We need all background tasks to **share a single counter** that tracks:

“what is the **next** sequence number that should be sent to the client?”

We *can’t* use an int, because integers in Python are **immutable**.  
If we passed an int, each task would get its own copy and updating it wouldn’t affect the others.

By using a **list**, all tasks share the *same object* and can update next\_sequence\_to\_send[0].

For example:

* Initially => [0]
* After sending chunk 0 => becomes [1]
* After sending chunk 1 => becomes [2]

So think of it as a **mutable shared counter**.

So technically:

* response\_buffer holds **ready responses**
* next\_sequence\_to\_send[0] acts like the **pointer to the next one that should be sent**

🙌 You’ve correctly captured the interaction between them.

So here in order to effectively send stuff up , we need to keep a buffer and a pointer

Absolutely — that's a **classic concurrency/control flow principle**.

✅ Since processing is **asynchronous and out-of-order**,  
✅ but the **consumer (mobile app)** expects responses **in-order**,  
➡️ you need **two things**:

| **Concept** | **Purpose** |
| --- | --- |
| **Buffer (queue/dict)** | Temporarily holds completed results |
| **Pointer (next\_sequence\_to\_send)** | Keeps track of the *exact* order in which they must be sent |

This pattern appears in many systems (stream processing, TCP reassembly, ordered message queues, etc).  
It’s basically **"reordering with a sliding window"**.

So yes — this is a **good CS principle**:

***When concurrency leads to out-of-order completion, introduce a buffer + ordering pointer to restore correctness.***

We’re now here:

while True:

# Wait for messages from mobile app

message = await websocket.receive()

**🔹 await websocket.receive()**

* Suspends until **either** a text frame **or** a binary frame arrives.
* Returns a **dict** with either "text" or "bytes" as the key.

Yes — that’s **exactly** right.

Let’s isolate the **text branch only** and walk through it clearly:

**🟦 Text Messages = Metadata Messages**

When the client sends **text** over the WebSocket, it’s **not** the actual audio —  
it’s usually a small JSON payload that describes **what audio is about to come**.

In code:

if "text" in message:

json\_data = json.loads(message["text"])

response = await process\_websocket\_message(transcription\_session\_id, json\_data)

await websocket.send\_text(json.dumps(response))

So the flow for **text messages** is:

| **Step** | **What it means** |
| --- | --- |
| Client → sends a JSON string | { "type": "audio\_chunk\_metadata", "sequence\_number": 0, "chunk\_size\_bytes": 81234 } |
| Server → parses it | json\_data = {...} |
| Server → runs process\_websocket\_message(...) | Validates session state and metadata |
| Server → immediately replies | e.g. { "type": "metadata\_received", "message": "Ready to receive audio chunk 0" } |

✅ This **lets the client know** that the server is ready to accept the actual **binary chunk** next.

**💡 Why is metadata sent first?**

Because sending a raw binary chunk *without context* is risky.  
The server needs to know:

* Which **sequence number** the chunk belongs to
* How **big** the chunk is (for checks / debugging)

So the client does:

1. send text → "hey, I’m about to send chunk #5 (size=31000)"

2. wait for server to reply → "okay, send it"

3. send the binary audio bytes

This is basically a **handshake per chunk**, and lets the server reject/accept each chunk individually.

Would you like to:

**A)** go deeper into process\_websocket\_message(...) to see exactly how that metadata is validated  
**B)** move on to the binary branch (actual audio data handling)

*So here we need a process in order to sort of like send a metadata chunk before sending the audio this will help us track sequence number and audio size\*

**💡 Why is metadata sent first?**

Because sending a raw binary chunk *without context* is risky.  
The server needs to know:

* Which **sequence number** the chunk belongs to
* How **big** the chunk is (for checks / debugging)

So the client does:

1. send text → "hey, I’m about to send chunk #5 (size=31000)"

2. wait for server to reply → "okay, send it"

3. send the binary audio bytes

**This is basically a handshake per chunk, and lets the server reject/accept each chunk individually.**

**✅ So yes, the function validates the metadata and then sends back an immediate acknowledgement to the client that says:**

**“✅ Metadata received — you may now send the actual bytes for this chunk.”**

Function Purpose

* This is a wrapper around the background pipeline.
* It ensures the system doesn’t process too many audio chunks concurrently.

async with TRANSCRIPTION\_WORKER\_POOL:

* TRANSCRIPTION\_WORKER\_POOL = asyncio.Semaphore(5) (defined at top of utils.py).
* Semaphore = a concurrency limit.
* Here, max 5 audio chunks can be processed at the same time.
* If a 6th chunk arrives, it waits until one of the 5 finishes.

**Delegates to process\_audio\_chunk\_background(...)**

* Once the semaphore is acquired, it runs the **real work**: store → transcribe → update transcript → buffer response.
* When finished, the semaphore is automatically released.

✅ That’s the entire purpose of this function: **rate-limited parallelism**.

**🔄 Full Flow of the WebSocket Endpoint**

1. **WebSocket Connection Established**
   * Client connects → server validates session → calls await websocket.accept() → connection is now open.
2. **Client Sends Metadata (text frame)**
   * Client sends JSON like:
   * {
   * "type": "audio\_chunk\_metadata",
   * "sequence\_number": 0,
   * "chunk\_size\_bytes": 81324
   * }
   * Server runs process\_websocket\_message(...) → validates session state and metadata → replies:
   * {
   * "type": "metadata\_received",
   * "message": "Ready to receive audio chunk 0",
   * "sequence\_number": 0,
   * "expected\_size\_bytes": 81324
   * }
3. **Client Sends Audio Chunk (binary frame)**
   * Raw audio bytes arrive in "bytes".
   * Server does **not** block. Instead, it spawns a background task for that chunk.
4. **Background Task Processing (max 5 at a time)**
   * Controlled by semaphore (TRANSCRIPTION\_WORKER\_POOL).
   * Each task does:
     + store audio in blob storage,
     + transcribe via Whisper,
     + update full transcript in DB,
     + prepare response dict.
5. **Buffer + Pointer for Ordered Sending**
   * Each task puts its result into response\_buffer[sequence\_number].
   * Then send\_buffered\_responses() checks:
     + if pointer (next\_sequence\_to\_send[0]) matches, send it, increment pointer,
     + keep flushing until it hits a gap.
   * This guarantees client sees responses in strict order: 0,1,2,… even if tasks finish out-of-order.

**Buffer exists because tasks finish out of order, but responses must go out in order.**

**Step 3 — Where the Race Happens**

Here’s your buffer-flushing code again:

while next\_sequence\_to\_send[0] in response\_buffer:

response = response\_buffer[next\_sequence\_to\_send[0]] # READ

await websocket.send\_text(json.dumps(response)) # SEND

del response\_buffer[next\_sequence\_to\_send[0]] # DELETE

next\_sequence\_to\_send[0] += 1 # UPDATE

At first glance this looks fine:

* Read from buffer,
* Send it to client,
* Delete from buffer,
* Increment pointer.

⚠️ But the problem is:  
**multiple background tasks can run this function at the same time.**

Example:

1. Task A and Task B both call send\_buffered\_responses() at nearly the same moment.
2. Both check the condition: next\_sequence\_to\_send[0] in response\_buffer.
3. Both see True (say sequence 0 is ready).
4. Both grab response\_buffer[0].
5. Both try to send it.
6. One deletes it, the other tries deleting → crash or duplicate.

A **race condition** in computer science happens when **two or more concurrent processes (threads, tasks, or programs) try to access or modify the same shared resource at the same time**, and the final outcome depends on the **timing** or the **order** of their execution.

Because the execution order is not guaranteed, the program may behave unpredictably—sometimes it works correctly, sometimes it produces wrong results, or sometimes it crashes.

**we need an asyncio.Lock() to ensure that only one background task at a time can access the critical section where we:**

1. **Check the response buffer**
2. **Send responses to WebSocket**
3. **Update the sequence counter**

**So wow->buffer pointer race condition async lock and background task**

**Step 1 — What actually happened (the symptom)**

During your stress test, you saw:

RuntimeError: websocket.close after sending websocket.close

This means:

* Your code **tried to close the same WebSocket connection twice**.
* The first .close() worked.
* The second .close() blew up because the connection was already closing/closed.

**must verify the WebSocket is still open before sending messages, otherwise we get the "websocket.close after websocket.close" error.**