**AudioScribe Data Model Design**

**Overview of the SwiftData schema**

The AudioScribe architecture centers around three main SwiftData models:

**RecordingSession**

* Represents a single audio recording session (e.g. a topic).
* Properties:
  + id: Unique UUID.
  + date: Date when the session started.
  + segments: Array of AudioSegment objects recorded during this session.
  + notes: Optional user notes or metadata.

**Purpose:** Group segments by recording events. A session can have thousands of segments.

**AudioSegment**

* Represents a single recorded audio file (user selectable ~10–30 sec chunk).
* Properties:
  + id: Unique UUID.
  + filePath: Stored as a string to support SwiftData (URLs not directly supported).
  + duration: In seconds.
  + transcription: Optional linked Transcription.
  + isUploaded: Marks if this segment has been processed, transcribed & uploaded.

**Purpose:** Decouple audio chunks from the session for efficient streaming and lifecycle management.

**Transcription**

* Represents the text transcription of a segment.
* Properties:
  + text: The actual transcription.
  + created: Timestamp of when transcription was completed.

**Purpose:** Separates language data from raw audio data, allowing us to run additional NLP or reprocessing later without touching the audio.

**Performance & Scalability Optimizations**

**Normalization of data**

* By splitting into **RecordingSession → AudioSegment → Transcription**, each model only contains what it needs.
* For datasets with **10,000+ segments**, loading just RecordingSession metadata keeps memory usage low.

**Lazy loading relationships**

* The UI only iterates session.segments when we drill down in SessionDetailView.
* The top-level list (ContentView) simply displays the dates of RecordingSession objects.
  + This means that even if we have over 1,000 sessions, the memory used by the UI grows only with the number of sessions, not with the total number of segments inside them

**Using primary keys**

* @Attribute(.unique) on id ensures SwiftData indexes each model by UUID.
* This makes FetchDescriptor efficient (as seen in transcribeSegment when fetching by ID).

let request = FetchDescriptor<AudioSegment>(predicate: #Predicate { $0.id == segmentID })

* SwiftData can use its underlying SQLite B-tree index to jump straight to the matching record.

**Avoiding ORM-heavy patterns**

* This design does not maintain large in-memory graphs. Instead:
  + It inserts into context (context.insert(segment)) and save.
  + When transcribing, it fetches only the segment it needs by ID.

**Efficient disk lifecycle**

* Audio files are stored **outside the database**, on disk, referenced by filePath.
* This means SwiftData never bloats with blobs.
* Files are cleaned up automatically after transcription unless user opts to keep them, keeping storage lean.

**Why this design is good for scale**

* It can handle **hundreds of sessions with thousands of segments** without loading them all into memory.
* Fast FetchDescriptor queries by id and indexed relationships ensure read performance.
* UI only loads nested data (session.segments) when needed, so scrolling stays fast.