**ψ-Trajectory Memory Subsystem Audit and Blueprint**

**Real-Time Expressiveness and Fidelity**

The proposed ψ-Trajectory Memory captures oscillator **phases**, **amplitudes**, and **emotional state vectors** across three temporal bands (micro, meso, macro). This multi-scale approach is intended to record both rapid facial/motor nuances and slower emotional shifts. **Cadence:** A high sampling rate at the micro level (e.g. 60 FPS or higher) appears appropriate – human micro-expressions can flash in ~40–200 ms[pmc.ncbi.nlm.nih.gov](https://pmc.ncbi.nlm.nih.gov/articles/PMC3296074/#:~:text=A%20microexpression%20is%20a%20brief,one%20of%20a%20hostile%20nature), so a ~60 Hz capture can reliably encode these subtle cues[forums.developer.apple.com](https://forums.developer.apple.com/forums/thread/651180#:~:text=It%20depends%20a%20lot%20on,action%20classification%20to%20work%20well). The meso and macro bands (with lower frequencies) ensure that broader gestures and mood arcs are not lost in compression by overly fine sampling. Overall, the three-tier cadence should be **sufficient for expressiveness**, provided that:

* **Micro-band** covers lip-sync, eye blinks, and quick expressions with frame-level granularity (≈16 ms/frame). This aligns with industry practices (e.g. ARKit face tracking runs at 60 FPS for smooth capture[forums.developer.apple.com](https://forums.developer.apple.com/forums/thread/651180#:~:text=It%20depends%20a%20lot%20on,action%20classification%20to%20work%20well)).
* **Meso-band** updates cover intermediate behaviors (intonation patterns, gestures over 0.5–5 seconds) without needing every frame. A plausible rate is 5–10 Hz for sentence-level or gesture-level changes.
* **Macro-band** captures emotional context or scene-level state (spanning tens of seconds to minutes). This could be as sparse as 1 Hz or event-triggered snapshots for significant shifts.

**Fidelity:** Storing raw oscillator states at these rates (even quantized) allows re-rendering the avatar with *frame-accurate audio-visual sync*. As long as the playback uses the same timebase as recording (e.g. driven by the original audio timestamps or a fixed frame clock), the avatar’s movements will align with the speech precisely. One potential improvement is to include **explicit timestamp fields** or frame indices in the schema – this guards against any drift if the playback loop timing differs. In summary, the current schema and cadence are conceptually sound for real-time fidelity: fine-grained enough to capture human-like expressiveness, but stratified to avoid wasting space on slow-changing signals.

**Data Integrity and Compression Trade-offs**

The subsystem uses delta-encoding, 16-bit quantization, and Zstd compression (level 22) to store the time-series efficiently. This design strikes a balance between **size and fidelity**:

* **Int16 Quantization:** Using 16-bit integers for oscillator phases, amplitudes, and emotion vectors greatly reduces storage versus 32-bit floats, while maintaining high precision. A 16-bit range gives ~1e-5 resolution (for example, an angle quantized to 16 bits has ~0.006° precision), which is **well below human perceptual thresholds**. Quantization noise is essentially negligible in facial animation (16-bit is akin to “CD-quality” resolution in signal terms). The only integrity concern is that rounding can introduce tiny errors – but these manifest as minute avatar differences (e.g. a facial feature off by a fraction of a millimeter) that are not noticeable. To be safe, critical signals (like lip-sync phoneme timings) could use alignment markers or higher precision if ever needed, but int16 appears sufficient.
* **Delta Encoding:** Storing frame-to-frame differences (Δ) exploits temporal coherence. Adjacent frames in an avatar’s motion tend to be similar, so delta values cluster near zero. This leads to extremely compressible data (many repeated small numbers). Facebook’s Gorilla time-series codec uses a similar idea – XORing consecutive float values so that only the changing bits are stored, yielding up to 12× reduction in size[medium.com](https://medium.com/@agustin.ignacio.rossi/facebooks-secret-weapon-for-time-series-data-meet-gorilla-bb844137276e#:~:text=For%20floating,size%20while%20keeping%20full%20precision). The ψ-memory’s Δ-encoding should achieve comparable efficiency: long runs of small deltas compress down to a few bytes[medium.com](https://medium.com/@agustin.ignacio.rossi/facebooks-secret-weapon-for-time-series-data-meet-gorilla-bb844137276e#:~:text=For%20floating,size%20while%20keeping%20full%20precision). This also enhances **data integrity**: since changes are stored, any single-bit error tends to only affect that moment’s value (not the absolute state of all future frames). We should, however, store an occasional full snapshot (key frame) to guard against error propagation – e.g. record an absolute state every few seconds so that if one delta is lost/corrupt, the sequence can resync at the next key frame.

**Data integrity considerations:** Below is a flat-file, append-only design that preserves all the advantages of the ψ-trajectory scheme while eliminating every database dependency.

1 · Container Format: .psiarc (Ψ-Archive)  
<session-dir>/

2025-05-14T21-17-03.psiarc ← main binary container

2025-05-14T21-17-03.meta.json← UTF-8 manifest  
**1.1 Binary container layout (.psiarc)**

| **Offset** | **Bytes** | **Field** | **Notes** |
| --- | --- | --- | --- |
| 0 | 4 | magic “ΨARC” |  |
| 4 | 2 | version (uint16) |  |
| 6 | 8 | unixEpochMs (start) |  |
| 14 | 2 | headerCRC16 |  |
| 16 | … | **Chunk stream** (see below) |  |

**Chunks**

┌────────────┬────────┬───────────┬───────────┐

│ uint8 tag │ uint24 length │ zstd-blob │

* └────────────┴────────┴───────────┴───────────┘  
  **Tags**  
  0x01 micro-band 0x02 meso-band 0x03 macro-band  
  0xFE audio-PCM 0xFF END
* Length is big-endian 24-bit (max 16 MiB per chunk).
* Each chunk contains **Δ-encoded int16 arrays** Zstd-compressed (-22).
* After writing the final 0xFF tag the file is **fsync()**’d then atomically rename()d from .tmp → .psiarc.  
  → crash-safe with no partial files.

**1.2 Manifest (.meta.json)**

{

"version": 1,

"persona": "mentor\_jane",

"duration\_ms": 8420,

"fps\_micro": 60,

"frames\_micro": 505,

"codec": "int16+delta+zstd",

"emotion\_vec0": [0.76,-0.12,0.44],

"sha256": "87c1…"

}  
Human-readable, helps tooling discover sessions without opening big binaries.

* **2 · Write API (C++ / Rust pseudocode)**
* rust
* CopyEdit
* let tmp = format!("{dir}/{ts}.tmp");
* let mut f = File::create(&tmp)?;
* f.write\_all(&header)?;
* zstd::stream::write::Encoder::with\_quality(f, 22)?
* .write\_all(&delta\_buffer)?; // micro
* // repeat for meso, macro, audio…
* f.write\_all(&[0xFF,0,0,0])?; // END tag
* f.sync\_all()?; // atomic guarantee
* std::fs::rename(tmp, final\_path)?;
* **Micro frames** buffered in RAM; flush every N seconds or at utterance end.  
  (You already handle this in your ring buffer.)
* No SQL transactions—atomic rename is the integrity primitive.
* Optional: write periodic *key-chunks* (absolute snapshots) to cap Δ-run length.

**3 · Read / Replay API**

rust

CopyEdit

let mm = memmap2::Mmap::map(&file)?;

scan\_chunks(mm, |tag, blob| {

let data = zstd::decode\_all(blob)?;

match tag {

0x01 => replay\_micro(data),

0x02 => replay\_meso(data),

0x03 => replay\_macro(data),

0xFE => play\_pcm(data),

\_ => {}

}

});

* **Memory-map** + streaming decode ⇒ replay starts in < 1 ms even on mobile.
* Seek support: store byte offsets for key-chunks in the .meta.json under "key\_index": [ { "time\_ms": 5000,"offset":123456 }, … ]. Jump-to-middle becomes an mmap.seek() + partial decode instead of linear scan.

**4 · Storage Cost**

| **Band** | **Raw/min (bytes)** | **Δ+Zstd 8×** | **Note** |
| --- | --- | --- | --- |
| micro (60 fps, 15 × int16) | ~108 kB | **≈ 13 kB** |  |
| meso (10 Hz, 10 × int16) | ~19 kB | ≈ 2 kB |  |
| macro (1 Hz, 8 × int16) | ~1 kB | ≈ 0.1 kB |  |
| **Total ψ** | **128 kB** | **≈ 16 kB/min** |  |
| Audio PCM at 24 kHz mono int16 ≈ 2.9 MB/min (can also be zstd-compressed or encoded as Opus to drop below 0.5 MB/min). |  |  |  |
| Bottom line: **full synchronized audio-visual replay ≈ sub-MB per minute**. |  |  |  |

**5 · Integrity & Drift Guarantees**

* **Per-chunk CRC32** (after decompression) ensures corruption is caught.  
  Store CRC in chunk footer or manifest.
* **Phase wrap** stored with modular Δ formula Δθ = ((θ₂-θ₁+π) mod 2π)−π => never overflows int16.
* **Absolute key-chunks every 5 s** guarantee replay divergence ≤ 1 LSB even after bit-flip in earlier Δ stream.
* **Replay fidelity test** (landmark RMSE, phase error, sync drift) unchanged—works identically with flat file.

**6 · Export Pipeline (no DB paths)**

1. **Load** .psiarc → mmap.
2. **Replay** into off-screen FBO (Metal/Vulkan).
3. **Mux** with audio buffer via:
   * **Desktop:** ffmpeg -i pcm.wav -f rawvideo -pix\_fmt rgba …
   * **iOS:** AVAssetWriter (H.264 via VideoToolbox).
   * **Android:** MediaCodec + MediaMuxer (MP4/H.264).
4. Output session-id.mp4.

No database calls are involved—only direct file I/O.

**Drop-in Next Step**

*Copy psi\_archive.rs (or .cpp) stub into tori/kha/psi\_mem/ and wire*

rust

CopyEdit

PsiRecorder::start("mentor\_jane")?;

…

PsiRecorder::push(frame\_time, phases, amps, emos)?;

…

PsiRecorder::finish()?; // writes .psiarc + .meta.json

You’ll be recording deterministic, replayable ψ-trajectories without ever touching SQLite, while keeping all compression and integrity benefits previously outlined.

**Replay Determinism and Potential Drift**

A key requirement is that replaying from ψ-memory yields **deterministic, perfectly synced avatar behavior**. Potential issues to evaluate include quantization loss, phase wrap handling, and long-term drift:

* **Quantization Loss:** As noted, int16 quantization introduces tiny differences from the original analog values. However, because the system replays *the exact recorded samples*, those differences do not accumulate unpredictably – they are baked into the recorded trajectory. In other words, we are not re-simulating dynamics from scratch (which could diverge if we rounded differently); we are doing a **frame-by-frame playback** of the stored sequence. This ensures determinism: given the same ψ snapshot data, the avatar output will be identical on each run. The only slight variability might come from how the rendering engine or audio engine handles time – for example, if frame timing is off by a microsecond, but that’s negligible. To maintain determinism, it’s crucial that the playback reads the same quantized values and does not apply any additional filtering that could amplify the quantization noise. The quantization error per frame is effectively like injecting a tiny random jitter to some control values – but at 16-bit scale, this is akin to micro-degree or 0.001-level amplitude changes, far below perceptibility. Thus, quantization alone should not cause visible *jitter* or drift. In testing, we can verify that an original live animation and the replayed animation look indistinguishable to the eye (the “Mean Opinion Score” from viewers should remain essentially unchanged, indicating no quality drop).
* **Phase Wrap and Modular Arithmetic:** Oscillator phases are cyclic (e.g. 0–2π range). The system must handle phase wrap-around carefully when delta-encoding. We should confirm that the implementation computes phase differences in a *circular* sense. For example, if an oscillator’s phase goes from 359° to 1° (near 2π wrap), the naive delta is +2°, not –358°. Using modular arithmetic to pick the smaller equivalent delta is essential to avoid encoding a huge jump that would decode incorrectly. In practice, we can ensure: delta\_phase = ((phase\_curr - phase\_prev + π) mod 2π) - π (in radians) or the int16 equivalent. This yields deltas in the range –π to +π (or –32768 to 32767 in int16 ticks) representing the minimal phase change. During replay, we accumulate these deltas and apply a modulo 2π at each step to reconstruct the true phase. This method keeps phase continuity intact. **Determinism check:** Using modular arithmetic avoids any ambiguity – the sequence of int16 deltas combined with mod addition will recover the exact original quantized phase at each frame. We must be cautious in the implementation: in C/C++, 16-bit integer overflow is undefined for signed types, so we should use unsigned arithmetic or explicit wrap functions (e.g. in Rust use .wrapping\_add). By doing so, the replay will exactly match the recorded trajectory cycle for cycle.
* **Long-Clip Drift:** Over very long clips (say >5–10 minutes of continuous animation), two kinds of drift could occur: **(a)** audio-video desynchronization, or **(b)** phase drift relative to original continuous values. (a) *Sync drift* can happen if the replay timing is not locked. For example, if the system plays audio of length L and tries to render N frames, if the effective frame rate is slightly off, by the end there could be offset. The solution is to tie frame playback to the audio clock or a common timestamp. Given we store the data along a timeline, the replay should use those timestamps – e.g. if 60 Hz, 600 frames should correspond exactly to 10 seconds. Ensuring that and dropping/duplicating a frame if needed will keep A/V sync perfect (in practice, with correct timing logic, drift will be zero). (b) *Phase drift relative to original*: This refers to the scenario where, due to quantization, an oscillator’s phase might slowly shift from where it would have been without quantization. Since we are not recomputing the phase analytically but simply copying the recorded values, this is not a concern for *playback vs original playback*. It would be a concern if we tried to use the stored parameters to *re-simulate* beyond the recorded duration (the error would accumulate). But our use-case is faithfully **re-rendering the same performance**, not extrapolating it. Therefore, within the recorded window, any phase error is exactly what was introduced at recording – there is no additional drift during replay. For sanity, we can define a measure of phase consistency: e.g. after replaying a long clip, check that oscillators that should have completed X cycles did so within a small fraction of a cycle error (phase error). In practice, an int16 phase gives 1 part in 65536 resolution; even if an oscillator ran for 1000 cycles, worst-case it might accumulate an error of a few quantization steps (<<1% of a cycle). This could be measured as a **ψ-phase stability metric** (see next section). If needed, one could mitigate even that tiny drift by occasionally storing an absolute phase value (like an anchor every N frames) – but it’s likely unnecessary given the resolution.

In summary, the replay should be **bitwise deterministic** and free of perceptible drift if implemented correctly. The keys are: use the recorded timeline exactly (to preserve sync), apply proper modular arithmetic for phases, and avoid re-calculating or filtering the data during replay. The subsystem’s design (snapshotting all needed state) means we are effectively doing a *video playback* of avatar parameters, so it should exactly mimic the original performance without divergence. Rigorous testing can be done by overlaying original vs replayed animations and checking for any divergence (see Fidelity Checks below for formal metrics).

**Storage Cost Estimates**

Despite logging multi-dimensional data at high frequency, the storage costs are quite manageable after compression. Let’s approximate the footprint per minute of dialogue:

* **Raw data rate:** Suppose the avatar has on the order of ~10 oscillators or parameters (phases for various facial/gestural oscillators, amplitude signals, plus an emotional state vector of a few components). At the micro-band (e.g. 60 fps), that might be ~10–15 values per frame. Each value stored as int16 (2 bytes). This yields ~20–30 bytes per frame. At 60 frames/sec, that’s ~1200–1800 bytes/sec. Over a minute (60 s), uncompressed data is ~72–108 KB. The meso and macro bands contribute marginally by comparison – even if meso adds, say, 2–5 values at 5 Hz, that’s under 0.5 KB/sec, and macro maybe a few values per second at most. So uncompressed, all bands together might be on the order of **100–120 KB per minute** of animation.
* **Compressed size:** Delta encoding and Zstd compression will shrink this dramatically. Because adjacent frames are highly correlated (many small deltas or zeros), we can expect compression ratios easily in the 5×–10× range (possibly more). For example, Facebook’s Gorilla achieved ~12× on some telemetry data using similar delta/XOR compression[medium.com](https://medium.com/@agustin.ignacio.rossi/facebooks-secret-weapon-for-time-series-data-meet-gorilla-bb844137276e#:~:text=For%20floating,size%20while%20keeping%20full%20precision). Even conservatively, if 100 KB/min compresses by 8×, that’s about **12–15 KB per minute** of dialogue. In other words, a 10-minute conversation might produce a ~150 KB ψ-trajectory file – extremely lightweight. This is negligible compared to the raw audio for 10 minutes (which would be several MB for uncompressed PCM). It’s also tiny relative to video: a minute of even 720p video is tens of MB.
* **Database overhead:** Storing in SQLite adds a small overhead per row or blob. If we store one big blob per dialogue or per utterance, the overhead is maybe on the order of a few hundred bytes for metadata. Even if we stored frame-by-frame (which we likely won’t, due to using compressed blobs), the overhead per row (~40 bytes) on 3600 rows (60 fps × 60 s) is ~144 KB, which is actually comparable to the data itself. This is why grouping the data and compressing as a blob is preferable. The storage is then essentially the compressed size plus minimal overhead. **Projection:** We can expect on the order of *single-digit kilobytes per minute*, making this subsystem highly storage-efficient. Logging hours of interaction would only be a few megabytes. This means we can comfortably keep extensive ψ-memory archives or include them in exported project files without bloat.
* **Memory and performance:** Decompression of a 100 KB (compressed) blob is near-instant on modern devices (Zstd can easily decompress at hundreds of MB/s on a desktop[percona.com](https://www.percona.com/blog/evaluating-database-compression-methods/#:~:text=Now%20let%E2%80%99s%20look%20at%20the,compression%20ratios%20achieved), and even mobile can do tens of MB/s). Thus, loading a minute of data (say 15 KB compressed) is essentially instantaneous (<1 ms). We could even stream directly from the database without fully loading if needed, but given how small it is, it’s simplest to load into memory. Writing the data (compression at level 22) is the heaviest part – compressing ~100 KB might take a few milliseconds at level 22 (which is fine if done asynchronously at end of recording or in a background thread). In real-time recording, we likely will buffer and compress after the fact, or compress in chunks. There’s no issue storing raw deltas temporarily in memory for a few seconds and compressing periodically.

In conclusion, storage cost is **very low**. We should of course validate these estimates with real data – e.g. record a known duration and measure the blob size – but the design is clearly geared to make the ψ-trajectory logs a tiny fraction of overall application data. This opens the door to keeping a **library of performances** or embedding the data in saved sessions, since a full hour of avatar animation might only be ~0.5–1 MB compressed.

**Cross-Platform Replay and Export Options**

The subsystem plans to enable real-time replay in-app and exporting the synchronized audio + avatar visuals via FFmpeg (with PCM audio and RGBA frame sequence). We need to ensure this is handled portably across platforms:

* **FFmpeg CLI vs Libav Libraries:** On desktop platforms (Windows, macOS, Linux), calling the FFmpeg CLI is a convenient solution to encode the final video. The pipeline would involve rendering the avatar frames to images (or feeding raw RGBA frames) and piping those along with audio to FFmpeg to produce a video file (e.g. MP4 with H.264 or similar). This approach offloads all encoding muxing complexity to an external process. However, on **mobile platforms, shipping the FFmpeg binary** is less practical – it adds a large footprint and may violate app store policies (due to dynamic linking or GPL components if not careful). Instead, one can use FFmpeg’s libraries (libavcodec, libavformat, etc.) compiled into the app (under LGPL compliance) or use native platform encoders. Using libav\* directly gives more control (we can feed frames to the encoder in memory) and avoids spawning processes. It also allows progress callbacks for UI.
* **Hardware-Accelerated Encoders:** For real-time export on devices, leveraging hardware video encoders is crucial. Desktop FFmpeg can utilize GPU encoding (e.g. NVENC on Nvidia, VideoToolbox on macOS) if configured, but even the CPU x264 at ultrafast preset should handle our moderate resolutions in real-time. On **Android**, the MediaCodec API provides H.264/H.265 encoding in hardware. FFmpeg’s support for MediaCodec is limited (historically only decoding was supported, and full encoder integration arrived only in recent FFmpeg versions)[stackoverflow.com](https://stackoverflow.com/questions/66127317/why-is-ffmpeg-not-using-hardware-decoding-on-android-when-using-the-h264-mediaco#:~:text=Why%20is%20FFMPEG%20not%20using,%E2%80%93). It may be simpler to use Android’s MediaMuxer and MediaCodec directly: render each frame to a Surface or retrieve pixel buffer, feed to MediaCodec, while feeding audio samples to the audio track of MediaMuxer. On **iOS**, we can use AVFoundation/AVAssetWriter or VideoToolbox to encode video frames with hardware acceleration. These platform APIs ensure efficient encoding (minimal CPU usage and faster completion) – important for battery and performance on mobile.
* **Export Format:** We should choose a widely compatible container and codec. MP4 with H.264 video and AAC (or even uncompressed PCM) audio is a safe default. H.264 is universally playable and hardware-encodable on virtually all devices. If we target modern platforms only, H.265/HEVC could be an option for smaller files, but H.264 is probably sufficient given our content (mostly talking-head animation – not extremely high-motion video). The resolution of avatar exports might be, say, 720p or 1080p, which H.264 can encode in real-time easily on desktop and mobile hardware encoders.
* **Pipeline Integration:** In a desktop scenario using FFmpeg CLI, the app would likely dump the audio PCM to a file (or pipe), and each frame to an image (or pipe raw frames via STDIN to FFmpeg using a y4m/raw format). Automating FFmpeg with the right arguments (framerate, resolution, etc.) will produce the video file. If using libraries, the integration is a bit more involved but offers finer control (we can avoid intermediate files by feeding data in memory). The choice may come down to development resources – FFmpeg CLI is quicker to implement but requires bundling FFmpeg and is less flexible on mobile. A hybrid approach is possible: use FFmpeg CLI on desktop, and custom/native encoding on mobile.
* **Mobile Optimizations:** On mobile devices, one must be mindful of memory and performance. It may be beneficial to implement frame rendering in a GPU-friendly way – e.g. render the avatar frames offscreen at the desired resolution using OpenGL/Metal, and then use that GPU texture directly with the hardware encoder (some encoders support input surfaces to avoid readbacks). This avoids large RGBA buffers shuffling between CPU/GPU. Since the ψ-memory gives us deterministic frame data, we could even replay the animation offscreen faster-than-real-time to generate the video quicker than the actual duration (if the encoder and rendering can keep up). However, encoders often run near real-time for complexity reasons, so expect roughly 1× speed on mobile when encoding with high quality.
* **Platform Compatibility:** FFmpeg libraries (libav) are cross-platform but require per-architecture builds. We should also consider **licensing**: using H.264 encoding technically requires patent licenses; most hardware encoders cover this for us (device OS manufacturers handle licensing). Using x264 (software encoder via FFmpeg) is LGPL and fine legally, but we should prefer hardware where available. If distributing an app, dynamic linking of FFmpeg is needed to comply with LGPL. These are engineering details, but important for production readiness.

In short, **for desktop** we can leverage FFmpeg (either CLI or libav) for a straightforward implementation. For **mobile**, plan on using native frameworks (MediaCodec, AVAssetWriter) to ensure smooth and efficient export. The output will be a synced audio-video file that can be played anywhere. The ψ-subsystem provides perfectly synced streams (we know exactly which frame corresponds to which audio time), so assembling them is straightforward once encoding is set up. We just need to carefully implement the frame loop to match audio timestamps. For instance, if the audio is 44.1 kHz PCM, we know how many audio samples per frame (if 60fps, 44100/60 ≈ 735 samples/frame); we should start at t=0, feed frames and audio in lockstep to the muxer. The result should have no A/V offset.

**Schema and Table Structure Refinements**

The current proposal uses a SQLite table to store the compressed ψ trajectories. We can refine the schema for clarity, efficiency, and future extensibility:

**Option 1: Single Table, One Row Per Recording** – A table where each entry corresponds to a contiguous dialogue or session segment, storing all bands’ data:

sql

CopyEdit

CREATE TABLE PsiTrajectory (

id INTEGER PRIMARY KEY,

session\_id TEXT, -- or foreign key to a Sessions table if needed

timestamp\_start REAL, -- start time of this segment (or NULL if relative)

duration REAL, -- duration of the clip in seconds

fps\_micro REAL, -- sampling rate of micro band (e.g. 60.0)

fps\_meso REAL, -- sampling rate of meso band (if fixed, or 0 if event-driven)

fps\_macro REAL, -- sampling rate of macro band (or 0 if event-driven)

data\_micro BLOB, -- Zstd-compressed blob of micro-band deltas

data\_meso BLOB, -- compressed blob of meso-band data (deltas or keyframes)

data\_macro BLOB, -- compressed blob of macro-band data

keyframe\_interval INT -- (optional) interval of periodic keyframes in frames, if used

);

In this design, when a recording is finished, we compress the collected data for each band and store it in one row. This makes retrieval simple (one SELECT gives all data). It also keeps the database small (few rows). The session\_id could link to a higher-level table if we have multiple conversations or characters. We include metadata like the FPS for each band so the replay system knows how to interpret timing (even if in practice micro FPS might be constant 60, it’s good to store it in case we adjust it or to document the rate used). If meso and macro aren’t regular sampling (perhaps macro only on significant changes), we might instead store those as event lists with their own timestamps inside the blob – in that case fps\_meso/macro might not apply. Alternatively, we could omit fps for macro and simply timestamp its entries relative to start.

**Option 2: Separate Tables per Band** – Another approach is to have three tables (psi\_micro, psi\_meso, psi\_macro), each storing time-series for that band. For example:

sql

CopyEdit

CREATE TABLE PsiMicro (

session\_id TEXT,

frame\_index INTEGER, -- or timestamp

phase\_blob BLOB, -- compressed delta or values for all oscillator phases at this frame

amplitude\_blob BLOB, -- compressed delta or values for amplitudes (if separate)

emotion\_blob BLOB -- compressed delta or values for emotional vector (if part of micro)

-- Primary key could be (session\_id, frame\_index)

);

And similarly for PsiMeso, PsiMacro with appropriate sampling intervals (or event times). However, this granularity seems unwieldy – it would result in many rows. It’s more efficient to treat each band’s entire sequence as a blob as in option 1. We could still use separate tables per band but probably unnecessary unless we anticipate querying, say, only macro trends across sessions (which is unlikely – usually we replay the whole thing together).

**Recommended Schema:** Use a **single table, one row per segment**, with separate BLOB fields for each band. This keeps things simple. If needed, we can even combine all bands into one blob (since they’re captured together). But keeping them separate might aid debugging and allow using different compression or strategies per band (e.g. macro might compress extremely well and could be combined with micro, but separate fields let us decide per band compression level if desired).

We should store some form of **initial conditions** in the data as well. For delta encoding, the first frame needs a baseline. This can be handled by including an initial full snapshot inside the compressed blob (for micro, the first frame’s absolute values). Our format could be: [header][compressed\_deltas] where header contains the first frame values (uncompressed or lightly compressed since it’s just one frame). The schema doesn’t need an extra column for that; it’s internal to the blob format. But it’s worth noting in documentation.

**Example:** A micro blob might start with a 16-bit value count and the initial values for each channel (phase0, phase1..., amp0, ... emot0,...), then followed by delta bytes for each subsequent frame. Similar for meso/macro, though macro might be stored as absolute values at each timestamp since it’s sparse.

**Indexes:** If we want to support partial replay or random access (e.g. jump to middle of a clip), we might add an index table mapping time → blob offset. A lightweight approach: include keyframes every N frames as mentioned. For instance, every 300 frames (~5s) we store an absolute snapshot. Then in an index table, record byte offset in the blob for those keyframes. This way, to seek to time T, we find the last keyframe <= T, jump in the blob and decompress forward from there. This is an advanced feature; initially, we can skip it if we always replay from start. But designing the format with optional keyframes now would be smart.

**Data Integrity:** The schema can include a checksum column if we want to double-verify data on loading. SQLite will store our blob faithfully, and Zstd has checksums internally, so this may be redundant. But for a production system, we might add:

sql

CopyEdit

crc32\_micro INTEGER, -- checksum of uncompressed micro data (for verification)

...

so that after decompression we can verify integrity easily. This could catch edge cases like a partially written blob or file corruption.

Finally, ensure to use **transactions** when writing a new row. Possibly use a separate thread to compress and then insert, so the UI isn’t blocked (especially at level 22 compression).

**Replay Fidelity Checks Definition**

To guarantee the replay is **indistinguishable** from the original live performance, we define formal fidelity metrics and checks:

* **Audio-Visual Sync Drift (MOS Drift):** We define a metric for **Mean Offset of Sync** over the clip. One simple check is comparing the total durations: if the avatar frames playback duration deviates from the original audio duration. Formally, if original audio starts at $t\_0$ and ends at $t\_1$ and the replayed video frames span from $t\_0'$ to $t\_1'$, then the drift $\Delta\_{sync} = (t\_1' - t\_0') - (t\_1 - t\_0)$. This should be 0 for perfect sync. We will enforce that $|\Delta\_{sync}| < 1\ \text{frame interval}$ (e.g. <16 ms) for any exported clip. Additionally, we can measure **offset over time**: at various points, check the alignment of phoneme-to-viseme timing. For example, the peak of an utterance’s waveform should coincide with the peak mouth opening in both original and replay. A practical test: overlay the original audio waveform with the replay’s lip-jaw angle curve and verify maximal cross-correlation at zero lag. The “MOS drift” term could also imply a subjective Mean Opinion Score – i.e. have human raters judge lip-sync quality. But objectively, we ensure no drift accumulates: the first word and last word are just as synced. If needed, a high-speed camera test (record the avatar and measure A/V offset) could be done, but likely overkill. Our software check using timestamps suffices.
* **Landmark RMS Error:** We will use a geometric measure to quantify visual fidelity of the face. During testing, we capture the avatar’s key facial landmark positions (e.g. eyes corners, mouth shape, etc.) from the original run and the replay. Let $L\_{i}(t)$ be the 2D (or 3D) position of landmark $i$ at time $t$ in the original, and $L'\_{i}(t)$ in the replay. We define the **root-mean-square error** for landmark $i$ over the clip as:  
  RMSEi=1T∫t0t1∥Li(t)−Li′(t)∥2dt.\text{RMSE}\_{i} = \sqrt{\frac{1}{T} \int\_{t\_0}^{t\_1} \|L\_{i}(t) - L'\_{i}(t)\|^2 dt }.RMSEi​=T1​∫t0​t1​​∥Li​(t)−Li′​(t)∥2dt​.  
  Discretely (summing over frames),  
  RMSEi=1N∑f=1N∥Li,f−Li,f′∥2.\text{RMSE}\_{i} = \sqrt{\frac{1}{N} \sum\_{f=1}^{N} \|L\_{i,f} - L'\_{i,f}\|^2 }.RMSEi​=N1​∑f=1N​∥Li,f​−Li,f′​∥2​.  
  We then consider the maximum or mean over all key landmarks. For perfect fidelity, RMSE would be 0. In practice, with quantization, we expect extremely small errors (e.g. <0.1 pixel). We set a tolerance threshold (e.g. **Landmark RMSE must be < 1% of the facial motion range**). For instance, if the mouth can open 50 mm, an RMSE of 0.5 mm is a 1% error, likely imperceptible. We aim for well below that. This check directly compares the original animation data to the replayed data (in our test environment, we can bypass rendering and compare the internal values of the model’s rig or blendshapes to avoid any rendering differences). Passing this means the replay is visually on-point.
* **ψ-Phase Stability:** This metric focuses on the internal oscillator state consistency. We want to ensure oscillators that were in sync remain in sync, and frequencies did not drift. A robust measure is to check the **phase error per oscillator** over time. Let $\theta\_k(t)$ be the phase of oscillator $k$ in original, and $\theta'*k(t)$ in replay. We compute the circular difference $\Delta\theta\_k(t) = \mathrm{mod}*{2\pi}(\theta'\_k(t) - \theta\_k(t))$, mapped to $[-\pi,\pi]$ range. We then look at statistics of $\Delta\theta\_k$ over the entire clip. Ideally this stays near 0. We can define:
  + **Max phase error**: $\max\_{t \in [t\_0,t\_1]} |\Delta\theta\_k(t)|$ for each oscillator $k$. This tells us the worst-case deviation in radians.
  + **Phase drift rate**: $(\theta'\_k(t\_1) - \theta'\_k(t\_0)) - (\theta\_k(t\_1) - \theta\_k(t\_0))$, basically difference in total cycles completed in replay vs original. This should be 0 (meaning the oscillator completed the same number of rotations). Any non-zero indicates drift. We expect this to be zero or at most ±1 quantization step over the whole clip (which is negligible).
  + **Phase coherence**: Another check is if multiple oscillators were meant to maintain a phase relationship (say two coupled oscillators), we ensure the relative phase between them in replay matches original. That can be checked similarly by differences of differences.

We will declare the ψ-phase stability as **pass** if for all oscillators, $\max |\Delta\theta| < 0.5^\circ$ (in radians ~0.0087) and net drift is zero cycles. These tight bounds basically assert that quantization did not let any oscillator wander off noticeably. Given our storage approach, we expect this to hold easily (phase errors on the order of 0.005° per step as estimated, which over thousands of frames remains under 0.5° total drift).

In addition to these, we can monitor **audio waveform error** if the system ever regenerates speech, but here we assume audio is original PCM, so no error there. The above metrics – sync alignment, landmark RMS, and phase error – form our replay fidelity test suite. We can automate these in development by recording a test animation, replaying it, and computing these values. If all are below thresholds, the subsystem passes. In production, these checks ensure that the ψ-memory is truly a lossless (perceptually) recording of the performance.

**Rust/C++ Porting Considerations (Precision and Pitfalls)**

Porting the ψ-Trajectory system to a low-level language (Rust or C++) requires careful handling of numerical precision and memory:

* **Fixed-Point and Floating-Point Types:** In higher-level prototypes (Python, etc.), floats are typically 64-bit by default and integers are unbounded. In C++/Rust, we must choose appropriate types:
  + Use **exact-width types** for stored data: e.g. uint16\_t or int16\_t for the quantized values and deltas to match the design. This avoids any ambiguity about size across platforms.
  + When accumulating or converting types, beware of overflow. For example, summing int16 deltas in an int16 variable will overflow quickly. Instead, accumulate in a larger type (e.g. 32-bit int) then cast to 16-bit with modulo as needed.
  + For phase arithmetic, using unsigned 16-bit might be convenient (as it naturally wraps modulo 65536). If using signed 16-bit, ensure to apply modulo operations manually. In C/C++, signed overflow is undefined, so do not rely on wraparound of short. Instead, use unsigned math or standard library functions (or in Rust use Wrapping arithmetic).
  + If any intermediate calculations require floating point (e.g. if converting an amplitude to a physical value), decide on float vs double. Given 16-bit precision of data, a 32-bit float is usually enough (it has 23 bits of mantissa, plenty to exactly represent 16-bit values). But if combining many operations, double might be safer to avoid cumulative rounding. For example, if we compute an emotional vector norm or something for analysis, double is fine. However, for simply moving data, float is fine. **Consistency** is key: use the same formulas and precision as originally used when encoding. If the original quantization scaled a float to int16 with a certain formula, replicate it exactly – differences in rounding between languages can cause off-by-one errors. Write unit tests comparing a known sequence encoded/decoded in old vs new implementation to verify bit-perfect agreement.
* **Endianness:** SQLite stores integers in the host endianness by default (for binary blobs, it just stores bytes as given). If we write binary blobs of int16 from a little-endian system, and then read on a big-endian system, the bytes would flip. To ensure portability, it’s best to define the blob format endianness explicitly. Typically we choose little-endian for multi-byte values in the blob. In practice, most systems we target (x86, ARM) are little-endian, but if there’s any chance of cross-endian use (or just for clarity), we can either store everything in a byte-stream form (which we already do) or include a flag in the blob header. When implementing in C++/Rust, if writing out binary data structures, use functions that control endianness or manually byte-swap to little endian when writing to the blob.
* **Zstd Integration:** Both Rust and C++ have Zstd libraries available. We need to ensure the compression settings match the original (level 22, and any dictionary if used). Precision issues here are not numerical but about using the library correctly: allocate compress buffers of proper size (the max compressed size can be obtained via Zstd API), check return codes, etc. In Rust, the zstd crate can simplify this. In C++, linking to zstd and calling ZSTD\_compress2() is straightforward. Just ensure to pass the exact level. A trap: Zstd’s highest level is 22 as of now, but if we ever upgraded to a new version that extends levels, we should keep it consistent on encode/decode. Also, confirm that the decoder on other platforms can handle the data (should be fine as Zstd is stable and cross-language).
* **SQLite Access:** Using SQLite from C++ might involve the C API or a wrapper (like SQLiteCpp). In Rust, rusqlite crate is common. Be mindful of data types when binding/retrieving:
  + For blobs, ensure we use the appropriate calls to get raw bytes and not try to interpret as text. Also, check that the blob size matches expectations. One pitfall is forgetting to step the statement or finalize properly – but that’s more general.
  + Use transactions for writing as mentioned. In Rust, rusqlite has transaction support; in C++, call BEGIN/COMMIT or use the API.
  + Also, be careful with Unicode if session\_id or other fields are text – ensure consistent encoding (UTF-8).
* **Precision in Math Ops:** If any computations are done in replay (for instance, blending between micro and meso signals, or applying the oscillator values to a rig), be aware of floating-point determinism. C++ and Rust (on the same hardware) will give deterministic results for deterministic code, but differences can arise if using optimized BLAS or SIMD instructions that accumulate differently. Since most of our replay is just reading values and setting them, we’re fine. But say we do a cubic interpolation between frames – slight differences in how FMA vs separate multiply-add might yield tiny variations. If we need bit-exact reproducibility across implementations, we’d have to be strict about using the same math ops order. Generally, this is not a big issue unless the replay does heavy computation. Our design is more about *record and playback*, so it should be inherently consistent.
* **Memory Alignment and Packing:** If we create structs to mirror the data (like a struct for a frame with several int16 fields), watch out for struct padding and alignment. It’s usually simpler to treat the data as an array of int16 or a byte buffer. If using struct, mark with #pragma pack or equivalent in C++ to avoid padding bytes sneaking in (which would break the expected size). In Rust, one can use #[repr(packed)] if needed. But since we likely will build the blob manually, it’s fine.
* **Multi-threading and Concurrency:** In a systems language, one might utilize threads to e.g. compress in background or stream frames. Ensure to guard the SQLite connection if it’s shared (SQLite is not fully thread-safe unless using serialized mode). Alternatively, use a separate DB connection for background operations. When porting, these concerns come in – whereas in Python GIL might have avoided some race conditions implicitly. Just design clearly who owns the data: e.g. once recording is done, hand off the buffer to a worker thread to compress and write, while the main thread can continue.
* **Testing in the new language:** Create test vectors: a short known sequence of oscillator values, compress them in the old implementation, then decode in the new one and compare, and vice versa. This will catch any little-endian, struct packing, or off-by-one mistakes early. Also test extreme values: phases wrapping around 0, int16 min/max (e.g. an amplitude of -1.0 that maps to -32768 perhaps). Make sure those edge cases don’t cause undefined behavior (e.g. if -32768 is stored in a signed 16-bit, reading it back is fine, but just be careful with negation or absolute value of that as it can overflow a signed 16-bit).

By following these guidelines, the Rust/C++ implementation will maintain the same fidelity and stability as the prototype. The main “precision traps” are avoiding unintended overflow and ensuring identical data transformations. With the above in mind, the port should yield a high-performance, robust ψ-memory module ready for integration.

**Integration into TORI’s Replay/Export Pipeline (Checklist)**

Finally, to integrate the ψ-Trajectory Memory subsystem into the overall TORI/ALAN avatar engine, we outline a checklist of tasks and best practices. This will ensure a smooth deployment and modular control via the UI:

**1. Recording Pipeline Integration:**

* **Hook into Avatar Update Loop:** Modify the avatar animation loop to capture state each frame (or at the designated micro timestep). Likely, after the avatar computes a frame’s pose (driven by live input or AI), we sample the oscillator phases, amplitudes, etc. Insert a call like PsiRecorder.capture(frame\_time, phases, amplitudes, emotions) in the update cycle.
* **Buffering Strategy:** For performance, do not write to SQLite each frame. Instead, accumulate data in memory (e.g. in arrays or a custom struct). Possibly use a ring buffer if this is continuous, but since each dialogue has an end, a dynamic array that we reserve sufficient size for is fine.
* **Meso/Macro capture:** If meso and macro bands are derived (e.g. by downsampling or by events), the recorder should handle them. For example, macro might capture at scene breaks or when the emotion vector changes significantly. Define how to capture macro – perhaps check at each second if there’s a new dominant emotion or just record the start and end emotion. The implementation should populate those buffers accordingly (could be as simple as copy the current emotion every second).
* **Δ-Encoding on the fly:** We have a choice – we can store raw values for all frames then post-process into deltas, or compute deltas as we go (after the first frame). Computing deltas on the fly is fine (less data to store transiently). But be careful: if using deltas, you must remember the last value; that’s easy. Alternatively, store raw and delta-encode at compression time – both approaches are okay. On-the-fly delta saves memory if we have huge data, but our data isn’t huge. Simplicity might argue for storing raw then delta encode during compression.
* **Threading:** Decouple recording from compression. When a session ends, stop capturing and signal the compression/storage routine to begin. This can be done on a background thread so the UI can remain responsive. If the user immediately starts another session, you can either delay it until compression done or run them concurrently (just ensure separate buffers).
* **Transaction Safety:** When writing the compressed blob to SQLite, start a transaction and commit. Ensure error handling – if it fails (out of disk, etc.), log it and perhaps alert the user. Also, consider what to do if the app crashes mid-session: perhaps write interim checkpoints? This might be overkill unless sessions are very long. A simpler approach: ensure the memory buffer persists until flush, so a crash means losing the current unsaved session only.

**2. Replay Functionality Integration:**

* **Loading Data:** Provide a function like PsiReplay.load(session\_id) that queries the SQLite, retrieves the compressed blobs, and decompresses them into memory structures. This should set up the replay state (fill arrays of values or an iterator that can yield frame by frame).
* **Timing and Sync:** The replay system should drive the avatar animation at the recorded tempo. There are two main ways:
  + *Frame-stepping:* If the engine has a mechanism to render at a specified frame rate, you can step through each recorded frame on a timer (e.g. schedule update at 1/60s intervals). At each step, set the avatar’s oscillator states to the stored values for that frame, and render. This basically “puppets” the avatar with recorded data. The audio should be played back simultaneously – ideally, start audio and then start stepping frames. Use the audio clock or a high-precision timer to ensure if a frame is ever late, you catch up or drop as needed to keep sync. Since our data is dense (every frame recorded), dropping one frame out of 60 won't even be noticed, but desyncing audio will. It might be wise to tie frame display to audio timestamps (e.g. every 160 samples at 48kHz corresponds to one frame at 60Hz, so after N audio samples played, ensure N/160 frames have been displayed).
  + *Time-based interpolation:* Alternatively, one could use the system clock to drive and interpolate the values if the frame rate differs. But since we recorded presumably at engine frame rate, simplest is 1:1 playback.
* **Determinism:** Use the stored values exactly. Do not run the normal animation logic (which might involve AI or random elements). Instead, we likely need a mode in the avatar engine like Avatar.setOscillatorStates(phases, amps) to directly override the internal state. Essentially, bypass normal driving logic during replay. This could be a flag in the code (e.g. if(replayMode) use replay data; else use AI/live data). Implement this cleanly so it’s easy to switch on/off.
* **UI Controls:** Hook up play/pause/stop controls in the UI to the replay system. For example, when user clicks “Replay last session,” the system should load data, start audio playback (from file or cached audio buffer) and begin the frame loop. Provide a way to pause – this likely means pausing audio and halting the frame timer. Provide stop – reset to start. If possible, allow seeking (this is where having keyframe index helps if we want a scrub bar). Seeking can be a stretch goal – not necessary for a basic replay but nice to have. Even without explicit seek, being able to start from beginning is required; starting from arbitrary points is optional.
* **Testing Replay:** Do dry-runs where we log values, reload, and ensure they match as discussed. Also test that if the user interacts or the system receives other events during replay, it doesn’t interfere. Possibly disable user input to the avatar during replay (to not mix live and replay data).

**3. Export (Video Creation) Pipeline:**

* **Offscreen Rendering:** For exporting video, we might want to render at a higher resolution than the real-time view, and without UI overlays. Set up an offscreen buffer or a special export mode that draws the avatar to a pixel buffer each frame. For instance, if using OpenGL, create an FBO of the desired size. This may involve increasing detail (if the avatar has level-of-detail settings). Make sure lighting etc. remains consistent.
* **Frame Capture Loop:** Similar to real-time replay, but instead of drawing to screen at realtime, iterate through every frame of the session as fast as possible, rendering to the buffer and retrieving the pixel data. This can be done in a loop that is not limited by 60fps (since for export we can go faster or slower as needed). However, if encoding on the fly, the encoding may dictate speed. A common approach is:
  1. Start audio playback or have audio data ready (if we use an encoder that needs audio input in real-time, we might have to feed it in sync; if using file-based muxing, we can compress audio ahead of time or use the original file).
  2. Initialize video encoder (via FFmpeg or platform API) with width, height, frame rate.
  3. For frame\_index from 0 to N-1: set avatar to state at that frame\_index, render to buffer, get pixels. Send pixels to encoder (or write to pipe). If using FFmpeg CLI, you might write raw frames to a pipe it’s reading. If using libav or platform, call encode function with the frame.
  4. Also feed audio samples to encoder/muxer. If using FFmpeg CLI and giving it audio file, you can just let it handle audio. If manual, you need to feed audio frames aligned with video PTS (presentation timestamps).
  5. Finalize: finalize encoder, close file, etc.
* **Progress and Feedback:** Encoding could take some seconds or minutes depending on length and resolution. Provide a progress indicator. We know total frames = N; we can update progress = frame/N \* 100%. Also handle cancellation: if user cancels, break out and properly close encoder (partial file likely not usable, but at least free resources).
* **Mobile-specific:** On iOS, the integration might use AVAssetWriter with an AVAssetWriterInputPixelBufferAdaptor for video. On Android, use MediaCodec and MediaMuxer as outlined. These require carefully handling the threading (often you have to feed frames on a separate thread and wait for encoder callbacks). Ensure to use correct color format (likely RGBA -> NV12 or similar conversion may be needed for hardware encoders). We might need to include a converter or let the GPU do it. FFmpeg software can accept RGBA easily but hardware usually likes YUV. This is an implementation detail, but worth planning.
* **File Handling:** Decide where to save the video. On desktop, prompt user or use a default path. On mobile, use an accessible location (app Documents or temp and then perhaps share via share-sheet). Make sure to handle permissions if needed (Android file write permissions, etc.). After export, provide user feedback (“Export completed: video saved to ...”).
* **Cleanup:** After export, destroy any offscreen surfaces, free large buffers, and reset the avatar to normal mode.

**4. Modular UI Integration:**

* **UI Triggers:** In the UI layer of TORI, add options/buttons for:
  + Start/stop recording (if recording isn’t automatic). Perhaps the system records every conversation automatically, or maybe user toggles. Clarify this. It might be better to always record by default (since it’s lightweight) and then offer replay/export if needed.
  + A “Replay” button to review the last conversation or a list to pick past recordings (hence session\_id in DB can help enumerate).
  + An “Export video” action, possibly on the replay screen or as a separate menu (“Save Performance as Video”).
* **Modularity:** The UI should ideally interact through a **Controller/Service API** rather than directly manipulating internals. For example, have a PsiMemoryManager class that provides methods: startRecording(sessionId), stopRecording(), replay(sessionId), exportVideo(sessionId, filename). The UI calls these, and the manager handles the heavy work (possibly spawning background tasks as needed). This keeps UI code simple and the subsystem encapsulated.
* **User Feedback:** When replaying, the UI might indicate it’s a playback (maybe a bar or icon “Playing back recording…”). During export, show a progress bar or spinner. If the export is long, maybe allow it to run in background and notify when done.
* **Multiple Sessions:** If the product allows saving multiple sessions, create a UI list from the database (session\_id could be timestamp or user-given name). The user could pick one to replay or export. Ensure the system can handle loading old data – that’s another reason to store fps and such in the DB in case it changes version to version.
* **Testing in UI:** Do a full run: record a snippet, stop, then replay it – the avatar should mimic exactly. Then export it and watch the video file with sound – verify sync (e.g. does a clap happen with mouth clap, etc.). Also test edge cases: very short recordings (a few seconds – does replay handle it gracefully?), very long (does memory hold up, does progress stay accurate?). Test cancelling export.

By following this checklist, integration into TORI/ALAN will be robust and user-friendly. The ψ-Trajectory Memory subsystem will operate behind the scenes during normal use (quietly logging), and then the UI can tap into it for advanced features (instant replays, or exporting a “video” of the AI avatar’s performance to share). We maintain modular design: the UI doesn’t need to know the compression details – it just triggers recording or playback, and the subsystem handles the rest.

With these considerations, the ψ-Trajectory Memory subsystem is defined as a **production-grade spec**. We have covered schema design, real-time fidelity assurances, compression trade-offs, deterministic replay concerns, cross-language implementation details, and integration steps. This blueprint can guide the development team in implementing persistent, high-fidelity avatar memory and a seamless replay/export feature that enhances TORI’s capabilities. The end result will allow an avatar’s performance to be captured, losslessly replayed at any time, and exported to common video formats – all while ensuring precision and synchronization at the level of human perceptual limits (or better).