# Milestone 3.7: Session Metadata Logging and Review (Step-by-Step Guide)

## Introduction and Goals

In this milestone, we will implement **robust session metadata logging** on the PC application. The goal is to **record a comprehensive diary of each session** – capturing all key events, their timestamps, and relevant details. This log will serve both **live feedback** for the operator and **post-session analysis** for the development team. By the end of this task, the PC app will document everything from device connections to recording start/stop times, stimulus events, user markers, and any errors. The result will be a **single log file (JSON format)** per session containing structured metadata, and a UI component that displays these events in real-time. This ensures we address both live monitoring needs and retrospective review of session data.

## Designing the Session Metadata Log Format

To facilitate easy analysis, we choose a **structured JSON format** for the session log (instead of plain text or CSV). JSON allows us to store rich, hierarchical data and is easily parsed by scripts or log analysis tools[[1]](https://www.loggly.com/use-cases/json-logging-best-practices/#:~:text=JSON%20logging%20is%20a%20kind,in%20a%20structured%20JSON%20format). Each session will produce **one JSON file** (e.g. session123\_log.json) containing:

* **Session Info:** Top-level information such as a unique session ID or name, the overall session start and end times (in a standard timestamp format, e.g. ISO 8601), and perhaps a human-readable session label if provided.
* **Device List:** A list of devices involved (with identifiers like “Phone1”, “Phone2”). For each device, we can record metadata like connection time (when it connected to the PC) and any clock offset measured relative to the PC (this helps with synchronization later).
* **Events Array:** A chronological list of event entries, each a JSON object with details:
* event: A short code or name for the event (e.g., "start\_record", "device\_ack", "stimulus\_play", "marker", "stop\_record", "file\_received", "error", etc.).
* time: Timestamp of the event. We should record this with high resolution (including milliseconds). This could be in absolute time-of-day (and date) or as an offset from session start. Absolute timestamps (with date) are unambiguous and recommended for cross-device analysis.
* Additional fields depending on event type, for context:
  + For **start of recording**: which devices were commanded to start (e.g., "devices": ["Phone1","Phone2"]), and the session ID.
  + For **device acknowledgment** of recording start: the specific device ID ("device": "Phone1") acknowledging and maybe an *acknowledgment timestamp*.
  + For **stimulus events**: the media name or ID (e.g., "media": "video1.mp4") and whether it’s a start or stop event.
  + For **marker events**: a custom label or note (e.g., "label": "note1" or "marker\_name": "StartTask"), the time it was inserted, and if applicable the stimulus playback timestamp ("stim\_time": "00:00:10.5" into the video, for example). This allows multiple distinct markers with meaningful names – an improvement over a single generic marker.
  + For **stop of recording**: typically just the time when the stop command was issued.
  + For **file transfer completion**: which device’s file was received, the filename, and file size (for verification).
  + For **calibration events**: any time a calibration image is captured or calibration is performed, including filenames of saved calibration data.
  + For **errors**: an error type or code and a message (e.g., device disconnects or command failures), plus relevant device ID if applicable.
* **Session Summary:** In addition to the events list, the JSON can include summary fields at top level for convenience:
* "start\_time" and "end\_time" of the session (possibly in Unix timestamp or ISO8601 string format).
* "devices" with their info (as described above).
* "calibration\_files" if any (list of filenames produced during calibration).
* "session\_name" if the session had a custom name or number.

By using JSON for structured logging, we make it easier to query and analyze logs programmatically (e.g., filter all "marker" events or calculate time differences)[[1]](https://www.loggly.com/use-cases/json-logging-best-practices/#:~:text=JSON%20logging%20is%20a%20kind,in%20a%20structured%20JSON%20format). The single JSON file per session will contain **all events in one place**, which simplifies data management (no need to merge multiple logs). Below is an **example** of how a portion of the JSON log might look for a session:

{  
 "session": "XYZ\_2025-07-28\_170500",   
 "start\_time": "2025-07-28T17:05:00.123Z",   
 "devices": [  
 {"id": "Phone1", "connect\_time": "2025-07-28T17:04:55.000Z", "clock\_offset\_ms": 50},  
 {"id": "Phone2", "connect\_time": "2025-07-28T17:04:55.100Z", "clock\_offset\_ms": 45}  
 ],   
 "events": [  
 { "event": "start\_record", "time": "17:05:00.123", "devices": ["Phone1","Phone2"] },  
 { "event": "device\_ack", "device": "Phone1", "time": "17:05:00.456" },  
 { "event": "stimulus\_play", "time": "17:05:10.000", "media": "video1.mp4" },  
 { "event": "marker", "label": "note1", "time": "17:05:20.500", "stim\_time": "00:00:10.5" },  
 { "event": "stop\_record", "time": "17:06:00.000" },  
 { "event": "file\_received", "device": "Phone1", "time": "17:06:05.000",   
 "filename": "XYZ\_Phone1.mp4", "size": 12345678 },  
 { "event": "file\_received", "device": "Phone2", "time": "17:06:07.000",   
 "filename": "XYZ\_Phone2.mp4", "size": 23456789 },  
 { "event": "session\_end", "time": "17:06:10.000" }  
 ],   
 "end\_time": "2025-07-28T17:06:10.000Z",   
 "calibration\_files": ["calib\_XYZ\_phone1.jpg", "calib\_XYZ\_phone2.jpg"]   
}

*(The above is an illustrative snippet; actual format can be adjusted as needed.)*

This structure is flexible: we can easily add fields if new event types or data need logging, and it’s human-readable for quick inspection.

## Implementing the SessionLogger Module

To manage this logging, we introduce a dedicated component, **SessionLogger**, responsible for collecting events and writing the log file. Below is a breakdown of this module and how it fits into the application:

* **SessionLogger Class**: This class (or set of functions) will encapsulate all logging functionality. It will be initialized at session start and finalized at session end.
* *Attributes*: It will maintain the data structures for the log: e.g., session\_id (or name), start\_time, an internal list/array of events, and possibly other summary info (like devices list, calibration list, etc.). It will also hold the file handle or file path for the log file being written.
* *Initialization*: A method like start\_session(session\_name, devices) will set up a new log. It records the session start time (now), the session ID/name, and device info. It then prepares the output file, e.g., creates a filename like session\_<name>\_log.json (or uses timestamp in name if no given ID). The file could be placed in a designated folder (for instance, ./Sessions/session\_<ID>/session\_<ID>\_log.json), which helps organize all files per session.
* *Logging Events*: A core method log\_event(event\_type, details) will be used throughout the app to append a new event. This method will:
  + Capture the current timestamp (now) for the event.
  + Create a dictionary/object for the event containing at least the event name and time, plus any fields passed in details (like device, label, etc.).
  + Append this event object to the internal events list.
  + **Flush to disk** (or schedule a flush) so that the event is written out. We don’t want to rely purely on in-memory storage, to avoid losing data on a crash. We can flush each event immediately to be safe – the flush() operation forces buffered data to be written to disk[[2]](https://www.geeksforgeeks.org/python/file-flush-method-in-python/#:~:text=The%20flush,For%20Example). In Python, for example, we can call file.flush() after writing, ensuring the OS writes it to disk right away[[2]](https://www.geeksforgeeks.org/python/file-flush-method-in-python/#:~:text=The%20flush,For%20Example). In C++ Qt, writing to file via QTextStream followed by flush() or simply closing/reopening can achieve the same. We might trade off a tiny bit of performance for reliability by flushing frequently, but the events frequency is low enough that this is fine.
  + If immediate flush on every event is too slow (should not be, but if so), an alternative is to accumulate and flush periodically (e.g., every N events or every few seconds). However, for simplicity and robustness, we'll flush important events immediately so metadata is preserved in real-time.
* *UI Update Hook*: The log\_event method can also interface with the UI – for example, it might emit a Qt signal with a human-readable message, or call a callback function to update the on-screen log. This ensures that as soon as an event is logged, the operator sees it.
* *Finalization*: A method end\_session() will mark the session end. It can add a final "session\_end" event (if we choose to log that explicitly) and record the end\_time in the summary. Then it will write any remaining data and properly **close the file**. Closing the file is important to ensure all data is flushed and the file is not locked. After this, the SessionLogger instance can be discarded or reset.
* **Choosing JSON Library/Method**: Depending on the implementation language:
* In **Python**, we can use the built-in json module. For example, maintain a Python dictionary for the whole log and use json.dump() to write to a file. One approach is to keep the file open and write incremental JSON lines, but since we want a well-formed single JSON, it may be easier to reconstruct and overwrite the file periodically. A simple strategy is to write the full JSON to disk at key points (after each event or after a set of events) by serializing the entire structure (this ensures the file is always a valid JSON). Given JSON is text, rewriting the file for tens or hundreds of events is typically fine.
* In **C++ (Qt)**, we can use Qt’s JSON classes. For example, use QJsonObject/QJsonArray to build the structure and QJsonDocument::toJson() to get a JSON string, then write it to file (via QFile). Qt also allows writing incrementally: one could write an “[[” at start, then for each event write a QJsonDocument of that event followed by a comma, and close ] at the end. But that is complex; it's easier to accumulate and write out when needed. The Qt JSON save game example demonstrates writing a JSON file with QJsonDocument[[3]](https://www.weiy.city/2020/08/how-to-write-and-read-json-file-by-qt/#:~:text=QJsonDocument%20document%3B%20document,8%22%20%29%3B%20iStream%20%3C%3C%20bytes).
* We will use JSON **indented format** if possible for readability (not strictly necessary, but nice for humans). Qt’s toJson(QJsonDocument::Indented) or Python’s json.dump(..., indent=2) can format the file nicely.
* **Log File Structure**: The JSON file will likely have a top-level object with several keys (as shown in the example earlier). We will populate it progressively:
* At session start: create the structure with session info and an empty events list, then write that initial structure to disk (or at least create the file).
* On each event: insert into the events list and rewrite or append as discussed.
* At session end: add end time and any final info, then do one last write (ensuring formatting and closure).
* **Ensure Robustness**: We have to be mindful of edge cases:
* If the app crashes mid-session, we want the log file to at least contain all events up to the crash. By flushing after each event or very frequently, we achieve that. The JSON might technically be missing a closing bracket if we were in the middle of writing, but if we rewrite fully each time, it will always be a complete JSON snapshot up to the last event. This way, even partial sessions are recorded.
* If a session is aborted or an error occurs, still save what was recorded. Perhaps in end\_session we ensure it gets called even on error (maybe via a finally block or in a Qt closeEvent). This might be beyond scope, but worth considering for data integrity.

In summary, **SessionLogger** acts as the central logging utility, exposing simple methods to log events throughout the application, while handling the details of JSON formatting and file output in the background.

## Integrating Logging into Application Events

With the SessionLogger in place, we now integrate logging calls at all the important points in the PC application. The following are the **key events to log and how to handle them**:

* **Session Start**: When the user begins a new session (e.g., by clicking "Start Session" or "Start Recording"):
* Determine a session identifier. This could be user-provided (like an input field for session name) or auto-generated (e.g., a combination of date and an incremental number). Initiate the SessionLogger.start\_session with this ID and the list of currently connected devices.
* Log an event {"event": "session\_start", ...} (or we might use "start\_record" to indicate recording has started, as per our example). This entry can include the session ID and which devices are involved. For example:
* {"event": "start\_record", "time": "17:05:00.123", "session": "XYZ", "devices": ["Phone1","Phone2"]}
* This denotes that at time 17:05:00 the PC instructed Phone1 and Phone2 to start recording as part of session "XYZ".
* Also, log device-specific info at start. We can either include device details in the session start event or log separate events for each device connection (e.g., if the devices were connected earlier, we might have already logged device\_connected events when they connected). For completeness, if not already logged, record something like {"event": "device\_connected", "device": "Phone1", "time": ...} at session start for each device with perhaps their initial timestamps. This ensures the log captures that those devices were online.
* *UI:* The moment session starts, the UI log panel should show something like “Session XYZ started. Devices: Phone1, Phone2”. This feedback confirms to the user that logging has commenced.
* **Device Recording Acknowledgment**: After the PC sends the start command, each device (Phone) should acknowledge that it started recording (assuming the system is designed to do so). When the PC app receives this acknowledgment (likely via a network message or signal):
* Log an event like {"event": "device\_ack", "device": "Phone1", "time": ...} for each device. This captures that Phone1 has begun recording at that moment. We could also include a field for how long it took since the start command, but this can be deduced by comparing timestamps of start\_record vs device\_ack.
* UI feedback: e.g., “Phone1 is now recording (ack received)” appears in the log panel. This lets the operator know that the device responded. If a device fails to ack within some time, that might be logged as an error (see errors below).
* **Stimulus Playback**: If the experiment involves playing a stimulus (video, audio, etc.) from the PC:
* When the operator or system starts the stimulus, log {"event": "stimulus\_play", "time": ..., "media": "<filename\_or\_id>"}. This marks the exact time the stimulus began. If the stimulus is played through the PC, this is straightforward (the PC knows the time it started). If devices were to play stimuli (less likely in this setup), we’d log when we send the play command and when they ack, similarly to recording.
* When the stimulus stops or reaches the end, log an event {"event": "stimulus\_stop", "time": ...} (if relevant for analysis). This is useful for knowing the duration of exposure.
* If multiple stimuli occur in one session, each should be logged (perhaps with identifiers if there are different videos).
* UI: Show messages like “Stimulus video1.mp4 started” and “Stimulus ended” in the live log.
* **User Marker Events**: The operator might insert markers during the session to note important moments (for example, if the subject reacted in a certain way, or to align with external events). We have now the capability for **multiple named markers**:
* In the UI, the operator could have a "Add Marker" button (or several buttons for different marker types) and possibly a text field to enter a label. For instance, they could mark events like "TaskStarted", "UserPressedButton", "ObservationX", etc. Each marker gets a unique label or note.
* When a marker is added, log an event such as:
* {"event": "marker", "label": "TaskStarted", "time": "17:05:30.250", "stim\_time": "00:00:20.5"}
* Here, stim\_time is optional – if a stimulus is playing, we log the timestamp within the stimulus (20.5 seconds into the video) to correlate the marker with the stimulus timeline. If no stimulus, we might omit it or set it to the session relative time.
* The ability to have multiple markers with different labels is now supported by including a label field. Each marker event is distinct in the JSON.
* UI: Each marker should immediately appear in the log panel, e.g., “Marker: TaskStarted at 00:00:20.5 of video (17:05:30)” for clarity. This confirms to the operator the marker was recorded.
* **Errors and Warnings**: We need to capture any abnormal events:
* If a device disconnects unexpectedly during a session, log an event:
* {"event": "error", "error\_type": "disconnect", "device": "Phone2", "time": "...", "message": "Lost connection to Phone2 during recording."}
* This way, when reviewing the log, it’s clear that after a certain point Phone2 might not have data.
* If a command fails (e.g., “Start Recording” sent to a phone but no response in timeout), log something like:
* {"event": "error", "error\_type": "no\_ack", "device": "Phone3", "time": "...", "message": "No start ack from Phone3."}
* If any other exception or user-facing error occurs (file save failed, etc.), it should be recorded similarly.
* UI: Error events should be highlighted if possible (maybe in the text log we can prefix with "ERROR:" or color the text red in the QTextEdit for visibility). The operator will then see messages like “ERROR: Lost connection to Phone2” in real-time. This helps in deciding whether to stop the session or take corrective action.
* **Stop Recording**: When the operator clicks “Stop” to end the recording on all devices:
* Log an event {"event": "stop\_record", "time": ...} at the moment the stop command is issued from the PC. This marks the official stop time of the session recording.
* The devices will presumably stop recording and send back the recorded files. The period between stop command and file receipt is also important to note (to gauge transfer times).
* UI: Show “Stopping recording...” and then “Recording stopped” in the log. The stop event in the JSON helps calculate total recording duration when analyzing.
* **File Retrieval**: After stopping, the PC will collect the recorded files from each device (perhaps via network transfer or USB). For each file received:
* Log an event for file reception, for example:
* {"event": "file\_received", "device": "Phone1", "time": "...", "filename": "XYZ\_Phone1.mp4", "size": 104857600}
* This confirms that Phone1’s video file was received at a certain time, and records its size (e.g., ~100 MB in this example). Logging the size is useful to ensure the file isn’t truncated and matches expectations.
* Do this for each device’s file. If multiple files per device (unlikely in this scenario, but if a device splits files), log each.
* UI: e.g., “Received Phone1 file (100 MB)” message in the log panel for operator feedback.
* **Session End**: Finally, once all the above are done, we consider the session completed:
* We may log a final event {"event": "session\_end", "time": ...} to explicitly mark the end. However, since we also record the overall end\_time in summary, this final event can be optional. Some designs include it for completeness.
* In any case, at this point we call SessionLogger.end\_session(). This will finalize the JSON structure (filling in the "end\_time" field at top, adding any last details like calibration results if needed) and then write out the JSON one last time with proper closure. The file is then closed.
* The log file path can be shown to the user or just known (we might print a message “Session log saved to sessionXYZ\_log.json”).

By logging all the above events, the metadata file becomes a **rich timeline of the session**. Later, one can parse this JSON to, for example, compute the delay between the PC’s "start\_record" command and each phone’s actual start (from ack times), or to align the stimulus timeline with video frames from each phone using the timestamps and offsets.

Throughout this integration, it’s important to ensure that **logging calls do not disrupt the normal operation**: - The logging functions should be lightweight. Inserting an event (a few dictionary operations and a small file write) is very fast relative to video recording tasks, so it won’t be a bottleneck. - We should, however, be careful about **threading**: if device communication happens on background threads (for example, a thread listening for device ack messages), calling UI updates or file writes from that thread can be problematic. The solution is to use Qt’s signal-slot mechanism (or similar if using another framework) to marshal data to the main thread: - For instance, when a background thread receives a “phone ack” message, it emits a signal with device ID; the MainWindow catches this signal in the GUI thread and then calls logger.log\_event("device\_ack", {...}) and updates the UI. This way, the UI update (QTextEdit append) is done on the GUI thread (avoiding thread-safety issues)[[4]](https://stackoverflow.com/questions/55290590/qt-command-log-using-qlistwidget#:~:text=I%20think%20,formatting%20features%20provided%20by%20QTextEdit). The file write can actually be done from any thread if using thread-safe operations, but it’s simplest to do it in one place (main thread) to avoid concurrency on the file.  
- The UI log component (discussed next) will show all events as they are logged, closing the loop for the operator.

## Live UI Feedback: Log Viewer Panel

To give the user/operator immediate insight into what’s happening, we will implement a **log viewer** in the UI that displays the events in human-readable form as they occur. For this, we will add a widget (for example, a text box area) to the main window:

* **Choosing a Widget**: We have two main options in Qt for displaying a scrolling text log: QTextEdit (which supports rich text but is heavier) or QPlainTextEdit (optimized for plain text and large output)[[5]](https://stackoverflow.com/questions/55290590/qt-command-log-using-qlistwidget#:~:text=I%20think%20,formatting%20features%20provided%20by%20QTextEdit). We will use a **QPlainTextEdit** for efficiency, since we don’t need rich text formatting for a log. (Another alternative could be a QListWidget with each event as an item, but that’s not ideal for a continuous log[[4]](https://stackoverflow.com/questions/55290590/qt-command-log-using-qlistwidget#:~:text=I%20think%20,formatting%20features%20provided%20by%20QTextEdit). A plain text box is more suitable for an append-only log view.)
* We will set this widget to **read-only** (the user should not edit log text)[[6]](https://stackoverflow.com/questions/55290590/qt-command-log-using-qlistwidget#:~:text=log).
* Optionally, customize its appearance: for example, a monospace font for clarity, and a fixed height or placed in a dockable panel. In Qt Designer or code, we can place it at the bottom of the main window or as a side panel labeled "Session Log".
* We might name it logTextEdit or logViewer in our UI code.
* **Updating the Log UI**: Whenever SessionLogger.log\_event is called, we will also append a line of text to this QPlainTextEdit. We can do this by calling logTextEdit->appendPlainText(QString) in Qt C++ or log\_text\_edit.appendPlainText(str) in PyQt, passing a formatted string. For example:
* For a start event: append text like "17:05:00.123 - Session XYZ: Recording started on Phone1, Phone2."
* For a device ack: "17:05:00.456 - Phone1 recording started (acknowledged)."
* For stimulus: "17:05:10.000 - Stimulus 'video1.mp4' started."
* For marker: "17:05:20.500 - Marker 'note1' inserted (video time 00:00:10.5)."
* For errors: "17:05:30.000 - ERROR: Lost connection to Phone2."
* etc.  
  These messages are derived from the event data but phrased for quick understanding. It’s helpful to include the time (at least minutes:seconds or timestamp) in front of each line. We might use the session relative time or absolute time; using absolute clock time (17:05:30) is straightforward as shown, or we could show time since session start (e.g., “[+00:00:20.5] Marker note1”). Either is fine, but absolute time gives a sense of real-time progression.
* **Auto-Scrolling**: We will enable the log view to scroll to the latest entry automatically, so the operator always sees the most recent event without manual scrolling. In Qt, after calling appendPlainText, we can do:
* logTextEdit->verticalScrollBar()->setValue(logTextEdit->verticalScrollBar()->maximum());
* to scroll to bottom[[7]](https://stackoverflow.com/questions/55290590/qt-command-log-using-qlistwidget#:~:text=,after%20adding%20the%20text). In PyQt, a similar approach can be used (accessing the vertical scroll bar and setting its value). This is an optional polish, but improves usability.
* **Thread-Safety for UI**: As mentioned, ensure that appendPlainText on the widget is done from the main thread. If our SessionLogger is tightly integrated with the UI (e.g., a method in MainWindow calls it), we’re already on the main thread for UI-triggered events. For background events (like device ack from a network thread), use signals to get back to main thread for the actual UI update. The SessionLogger could emit a Qt signal newLogEntry(QString) that the MainWindow connects to a slot where it does logTextEdit.appendPlainText(). This decouples logging from UI and is clean design.
* **Example Implementation**: Suppose in MainWindow::setupUI() (or in Qt Designer) we add:
* logTextEdit = new QPlainTextEdit(this);  
  logTextEdit->setReadOnly(true);  
  // (maybe set a fixed height or put in a QDockWidget as in example code)  
  addDockWidget(Qt::BottomDockWidgetArea, new QDockWidget("Session Log", this));
* (In practice, we would configure it properly in the UI layout. The Stack Overflow example shows adding a QListWidget to a QDockWidget for a log[[8]](https://stackoverflow.com/questions/55290590/qt-command-log-using-qlistwidget#:~:text=ui,Log); we can do similar but use QPlainTextEdit.)

When an event occurs, e.g., in on\_startButton\_clicked(), after calling logger, we do:

logTextEdit->appendPlainText("Recording started on all devices...");

or use a centralized function to format and append.

* With this live log view, the operator can **monitor the session in real-time**. This provides immediate feedback that things are working (or alerts if something goes wrong). It complements the JSON file (which is more for after-the-fact analysis).

*By combining the live UI log with the saved JSON, we have addressed both real-time visibility and retrospective data needs (fulfilling the “both live and past” requirement).*

## Finalizing the Log and Session Data Management

When the session is finished (either normally or aborted due to an error), we perform some final steps to wrap everything up neatly:

* **Finalize and Close Log File**: As part of stopping the session, ensure SessionLogger.end\_session() is called. This will:
* Record the final timestamp (session end) in the JSON structure.
* Optionally add a "session\_end" event if we want that in the timeline (otherwise the end\_time in summary suffices).
* Write out the JSON one last time with all events and close the file handle. After this, the file on disk is the complete record of the session. We should verify that the JSON is valid (all brackets closed, etc.). If we’ve been writing by repeatedly dumping the whole JSON, it will certainly be valid. If we wrote incrementally, we might need to write a closing bracket or so now.
* Confirm that the file is stored in the correct location (e.g., maybe we created a folder for this session). For example, we could have a folder named Session\_XYZ\_2025-07-28 containing sessionXYZ\_log.json and all the video files and calibration files from that session. This organization makes it easy for the team to find everything related to a session.
* **UI Indication of Completion**: Provide feedback to the user that the session data has been saved. This could be a simple message in the log UI: “Session completed. All files saved.” (We can even log that as an event, or just as a non-logged UI message). Additionally, one could pop up a message box: “Session XYZ finished. Data saved to ...”. This reassurance is user-friendly.
* **Reset for Next Session**: After finalizing, the application might reset some state in preparation for a new session:
* Clear out or archive the log viewer UI (maybe we leave it until a new session starts or allow scrolling through it).
* If devices remain connected, maybe leave them in idle state.
* The SessionLogger object can be discarded or re-initialized when a new session starts. If the user starts a new session, ensure a new JSON file is created (with a new ID or timestamp to avoid overwrite).

By diligently logging and finalizing, we ensure that **no crucial info is missing** in the metadata. We want to capture everything that happened in the session, from start to end, in the correct order.

## Post-Session Review and Verification

Although not a strict requirement, it’s very helpful to allow a **post-session review** to quickly verify data quality on the spot:

* After a session, the operator might want to review what was captured (especially if repeating an experiment is costly or difficult). We can implement a simple **Session Review dialog**. For example, when the session ends, enable a "Review" button. Clicking it could:
* Open a dialog or window that lists all the files produced (e.g., the video files from each device, calibration images, etc., which we know from the log or by scanning the session folder). The user could select a file and click “Open” to play the video (this would use the default video player application, or if we want, an embedded video widget).
* Display some summary stats: e.g., duration of recording (which can be computed from start/stop times), number of frames if that info is available, etc.
* Possibly show the contents of the log file in a friendlier way (maybe reuse the log text we already have).
* If calibration was done, show the calibration images or results (like print out the reprojection error or something, if computed).
* This review feature can be quite simple to start with. Even just opening the folder in the OS file explorer is useful (we can programmatically do that on a button press). The key is that the data is **well-organized** so that manual inspection is easy:
* By saving everything in a dedicated session folder and naming files clearly (e.g., include device name and timestamp in filenames), the user can manually open the folder and know what’s what. For example, Session\_XYZ/.../Phone1\_video.mp4, Phone2\_video.mp4, calib\_left.png, calib\_right.png, sessionXYZ\_log.json all in one place.
* The JSON log itself is an artifact for review; a team member could open it and read through the events, or load it into a tool to check synchronization.
* We will treat the post-session review as a **nice-to-have** addition. The core requirement is to have the data and log saved so the team can do analysis later (which we have achieved). Even without an in-app review UI, they can open the videos in a player and cross-reference with the timestamps in the JSON log to see if sync is okay. However, implementing a minimal review dialog could significantly streamline the workflow, so it’s a good extension if time permits.

## Class/Module Breakdown and Setup

At this stage (end of Phase 3), it’s useful to summarize the application’s structure, highlighting how the new logging feature integrates with existing components:

* **MainWindow (UI Controller)**:  
  This is the primary UI class (likely a QMainWindow). It contains UI elements and slots for user actions (start, stop, marker, etc.). After Milestone 3.7, MainWindow has:
* A new QPlainTextEdit (or similar) widget for the live log. Possibly placed at the bottom or in a dock widget labeled "Session Log".
* Slot methods like on\_startButton\_clicked(), on\_stopButton\_clicked(), on\_markerButton\_clicked(), etc., which now include calls to SessionLogger for logging. For example, in on\_markerButton\_clicked(), after capturing the marker label and time, call logger.log\_event("marker", {...}).
* It maintains an instance of SessionLogger (maybe as a member variable). When a session starts, it initializes this instance; when session ends, it finalizes it.
* If using signals from other threads (like device managers), MainWindow connects those signals to its own slots where it then calls the logger and updates the UI. So MainWindow acts as the orchestrator tying together UI, devices, and logger.
* **DeviceManager / DeviceConnection Module**:  
  This module handles connecting to mobile devices (phones), sending commands (start/stop recording, etc.), and receiving acknowledgments and files. After integration:
* On device connect: it could emit a signal or directly call logger.log\_event("device\_connected", {"device": id, ...}). (If devices connect before session start, these events might be logged outside a session context, possibly we only log them if a session is active. But it could be useful to log device connect events generally too.)
* When the user hits "Start Recording", DeviceManager sends commands to each device. Immediately after sending, we log the start\_record event (with devices list) on the PC. Then DeviceManager awaits responses.
* On receiving a device’s ACK (say in a network callback), DeviceManager can emit a Qt signal like deviceAckReceived(deviceId) which MainWindow catches to log the event and update UI. Or DeviceManager could call logger directly if it has access, but through MainWindow is cleaner for UI update.
* On receiving file from device, DeviceManager likely knows the file path and size. It should log the file\_received event (again via MainWindow or directly). Also it might save the file in the designated folder.
* On device error (disconnect, etc.), DeviceManager emits something like deviceDisconnected(deviceId, reason) which we log as an error event.
* *Note:* If DeviceManager runs in a separate thread, all these communications should use thread-safe signals to hand off to the logger/UI in the main thread.
* **StimulusController**:  
  If there's a module that manages playing stimuli (e.g., using a media player for videos):
* When it starts a stimulus, it triggers a log\_event("stimulus\_play", ...). This could be done in MainWindow if the play button is in UI, or in the controller with a callback to logger.
* If the stimulus has a known duration, maybe the controller can auto-log a stimulus\_stop when finished, or MainWindow logs it when the media player emits an "ended" signal.
* If stimuli are pre-scheduled (scripted experiment), the controller can log events accordingly at scheduled times as well.
* **Calibration Module**:  
  If the system includes a camera calibration step (perhaps earlier milestones handled capturing calibration images from each device):
* After capturing a calibration image from a device, log an event like {"event": "calibration\_capture", "device": "Phone1", "time": ..., "file": "calib\_XYZ\_phone1.jpg"}. This way we know at what time and which image was taken.
* After completing calibration (computing calibration parameters), log something like {"event": "calibration\_done", "time": ..., "result\_file": "calib\_result\_XYZ.json"} if there’s a result. Or simply note in the summary the names of calibration files (as we have a list calibration\_files).
* The UI can show messages like "Calibration images saved for Phone1 and Phone2" in the log view, reassuring the operator that step was done.
* The calibration data will also be saved in the session folder, as indicated in the summary.
* **SessionLogger**:  
  As detailed earlier, this is either integrated into the MainWindow or a standalone module. Its API (start\_session, log\_event, end\_session) is used by all the above components (through the MainWindow). It might also manage the session folder creation (e.g., making a directory for each session if desired).  
  If it’s a separate class, ensure the main application can instantiate and use it easily:
* In C++ Qt, you might add #include "SessionLogger.h" in MainWindow and have a member like SessionLogger logger;.
* In Python, you might import a logger module or class and create logger = SessionLogger() in the appropriate place.
* **IDE/Project Setup**:
* If using Qt Creator (C++), add the new source and header files for SessionLogger to your .pro file or CMakeLists so they compile into the app. Include Qt’s JSON (QT += core5compat if needed for QJson in older Qt, or Qt6 has it in QtCore). Also include <QDateTime> for timestamps.
* If using PyQt in an IDE like PyCharm or VSCode, just ensure the new class is imported and that the UI .ui file is updated to include the log widget (or if adding via code, ensure that code runs).
* Confirm that writing to disk is allowed in the chosen directory. For example, if running on Windows and writing to Program Files, that could be an issue; usually writing to user documents or same directory as exe is fine. For simplicity, we can assume the working directory is writable or choose a known folder (maybe allow configuration of the base output directory).
* Also, consider time zones for timestamps (using UTC vs local time). ISO8601 with Z (UTC) is unambiguous, but local time is more readable. We can log local time with offset or just local if all devices and PC are same zone. These are minor details to be consistent with.

With all modules working together, the PC application at this point is a **full-fledged control hub**: it connects to devices, starts/stops recordings, plays stimuli, performs calibration, **logs every step**, collects the data files, and provides feedback.

## Testing and Validation Plan

To ensure the metadata logging and review features work correctly, we will conduct a series of tests. This includes both simulated tests (without real devices, using dummy responses) and, if possible, a real integration test with actual devices. Below are **step-by-step test checkpoints**:

1. **Basic Logger Functionality Test**:
2. *Goal:* Verify that the SessionLogger can create and write a JSON file properly.
3. *Method:* In a development environment (could be a separate small test script), instantiate a SessionLogger with a sample session name (e.g., "TestSession"). Call start\_session("TestSession", devices=["Dummy1","Dummy2"]). Then call log\_event for a few made-up events: a start\_record, a device\_ack for Dummy1, a marker, a stop\_record, file\_received, etc., and finally end\_session().
4. *Expected:* After running these calls, open the output JSON file (e.g., TestSession\_log.json). Ensure it is well-formed JSON (try opening in a JSON viewer or simply in a text editor to see structure). Check that:
   * The session info (name, times, devices) is present and correct.
   * Each event appears exactly once, in correct order.
   * The timestamps recorded make sense (non-decreasing order).
   * The event fields (device IDs, labels, etc.) match what was logged.
5. *Fixes:* If any data is missing or formatting is wrong (e.g., a missing comma or bracket), adjust the logger code accordingly. This basic test catches file writing issues early.
6. **Live UI Log Test**:
7. *Goal:* Ensure that events are reflected in the UI immediately.
8. *Method:* Run the PC application in a debug mode where actual device interactions are stubbed or not needed. Start a session (this can be with dummy devices or even without devices by simulating that part). Trigger some events manually:
   * Click "Start" (with maybe no real device, but it will still call logger).
   * Check that a line “Recording started…” appears in the log text box.
   * If possible, simulate a device ack by calling the slot that would normally be called (perhaps make a test button that triggers onDeviceAck("Dummy1")).
   * Verify “Dummy1 recording started” text appears.
   * Click a "Marker" button (enter a label if required in UI).
   * See that “Marker <label> inserted…” line appears.
   * Click "Stop".
   * See “Recording stopped” (and possibly file received events if those can be simulated or manually invoked).
9. *Expected:* Every time an event is supposed to log, the UI text area updates instantly with a correct message. The scroll bar should move down to show the latest entry. The UI should remain responsive (no freezing when writing to file).
10. *Note:* Without actual devices, we might not see file\_received events. For test, one could simulate by calling the logger method directly or using a dummy thread to mimic a file arrival.
11. **Full End-to-End Session Simulation**:
12. *Goal:* Test the entire workflow including file outputs and finalization.
13. *Method:* Simulate a scenario programmatically:
    * Connect two dummy device objects (or if possible, two actual test devices configured). The devices could be simulated by threads or QTimers that respond after delays.
    * Start session “Demo1”. Have the dummy devices respond to start commands after a short delay (simulate ack).
    * Mark a stimulus start at some point (we can actually play a short video or just pretend by waiting 5 seconds then logging a stimulus event).
    * Insert a couple of markers during the “stimulus” playback (maybe one 2 seconds in, one 5 seconds in).
    * Stop the session after e.g. 10 seconds of “recording”. Then simulate devices sending files: call the file\_received log for each with dummy sizes.
    * End session.
14. *Expected:* The JSON log should show a **cohesive timeline**:
    * Session start at t=0.
    * Device acks maybe at t=0.2s etc.
    * Stimulus at t=5s.
    * Markers at t~7s (with stim\_time ~2s) and t~10s (stim\_time ~5s).
    * Stop at t=10s.
    * Files received at t=11s, t=12s, etc.
    * Session end at t=12s.  
      The UI log should have shown all these in order. The final JSON file should include everything and be complete (with end\_time). Also check that the session folder contains the dummy “files” (if we created actual dummy files for test) and the log file.
15. This simulation can be done with actual devices if available:
    * Connect real phones, start a short recording (even a few seconds). The phones should record and send back something. Then stop and verify the PC received files.
    * After this real test, open the JSON log and verify it recorded the real events properly (real device IDs, etc.).
16. If any mismatch or missing events are found (for instance, if the stimulus start wasn’t logged because the code didn’t call it properly, or a marker didn’t appear in JSON), fix the code and rerun the test.
17. **Error Handling Test**:
18. *Goal:* Ensure error events are logged and shown.
19. *Method:* Simulate an error scenario:
    * While recording, have one dummy device “disconnect”. You might invoke the device disconnect handler in DeviceManager manually in test. Or in a real test, turn off one device’s app to see if the PC detects a drop (if such detection is implemented).
    * The DeviceManager should emit a signal or call logger for the disconnect.
    * Also simulate a command failure: perhaps tell a dummy device to not respond to the start command and have a timeout trigger. Ensure the timeout handler logs an error (like "no ack from device X").
20. *Expected:* The JSON log should contain events with "event": "error" for these cases, with appropriate messages. The UI log should prepend "ERROR:" or otherwise make it noticeable.
21. Verify that the presence of an error doesn’t break the logging flow (the logger should handle logging an error like any other event). Also check that the session can still be ended gracefully even if a device dropped out. This might depend on broader app logic, but logging should at least capture it.
22. **Performance and Reliability Checks**:
23. Although performance is likely fine, it’s worth checking that the logging doesn’t introduce lag:
    * Try a stress test by generating a lot of log events in a short time (for instance, simulate 50 marker presses quickly). The UI log should update rapidly and the app should not crash. QPlainTextEdit can handle large text efficiently[[5]](https://stackoverflow.com/questions/55290590/qt-command-log-using-qlistwidget#:~:text=I%20think%20,formatting%20features%20provided%20by%20QTextEdit), so it should be okay even if we spam it.
    * Check memory usage for SessionLogger if a session runs long. Storing events in a list is usually fine (hundreds of events are negligible). If a session somehow had tens of thousands of events, memory might grow, but that’s unlikely in our use case.
24. Check the **flush mechanism** thoroughly:
    * After each major step in testing, intentionally crash or kill the app (in a non-production environment) to see what’s in the log file. For example, start a session, let it log a couple of events, then force close the app. Open the JSON file – it should contain those events (maybe missing the very last one if not flushed). If you find that the file is empty or missing many events, that means flush frequency is insufficient. Adjust to flush more often. Ideally, each logged event should be immediately written (and flushed) to disk, so that even a sudden crash loses at most the last log entry (if it was mid-write).
25. **Review Functionality Test** (if implemented):
26. *Goal:* Ensure the post-session review UI works.
27. *Method:* After a test session, click the "Review" button (or whatever triggers the review dialog).
    * See that the dialog lists the files (you might need to have real or dummy files present in the session folder).
    * Try opening a file from the list to see if it launches (if we implemented that).
    * Check that the session summary info (like duration, etc.) is correct as per the log.
28. *Expected:* The review dialog should accurately reflect what was recorded. If a video file is missing or the path is wrong, that indicates an issue in how we saved or how the path is stored. Fix any such path issues (for example, ensure that we log just filenames and know the folder, or log full paths).

By following these test steps, we can verify that the session metadata logging is **comprehensive and reliable**. Each test ensures a different aspect (writing correctness, UI integration, full workflow, error cases, etc.) is working.

If all tests pass, we will have high confidence in our logging system. It will be a powerful tool for debugging and analyzing the synchronization of multi-device recordings after the experiment. For example, if later on we find that one phone’s video is offset by 100 ms, we can look at the log to see if perhaps that phone acknowledged late or had a clock offset, etc., thus identifying the cause.

## Conclusion

Milestone 3.7 adds a crucial layer of transparency and traceability to the PC controller application. We have created a **technical infrastructure for session metadata logging** that records every significant event in a structured JSON log and provides instant feedback via the UI.

In summary, we achieved the following in this milestone:  
- **Single JSON Log File per Session** capturing all events and summary info, enabling structured analysis of the session timeline.  
- Support for **multiple named markers**, which allows the operator to tag various moments with descriptive labels, all of which are recorded with timestamps for later reference.  
- Proper log **finalization on session end**, ensuring that the log is saved and closed cleanly with all data (and making it easy to locate all files from the session together).  
- Both **live logging and post-session review** capabilities: the operator can monitor events as they happen through the real-time log viewer, and the team can review the saved JSON and data files afterwards to verify synchronization and data quality.  
- A full breakdown of how this logging integrates into each part of the application and thorough testing to validate the implementation.

With the completion of Milestone 3.7, the Phase 3 development of the PC application is complete. The application now acts as a robust control hub for multi-device recording sessions – it not only orchestrates the devices and stimulus, but also keeps an accurate journal of the session. This metadata log will be **invaluable for debugging and analysis**, for example, aligning video streams with the stimulus timeline or diagnosing any device issues by reviewing the log events.

The next steps (beyond Phase 3) might involve using this metadata to automatically synchronize videos, or improving user experience based on logged data, but as it stands, we have a solid foundation. By following this guide, developers can implement and verify the session metadata logging and ensure the system is reliable and ready for real-world experimental sessions.

[[1]](https://www.loggly.com/use-cases/json-logging-best-practices/#:~:text=JSON%20logging%20is%20a%20kind,in%20a%20structured%20JSON%20format) JSON Logging Best Practices | Loggly

<https://www.loggly.com/use-cases/json-logging-best-practices/>

[[2]](https://www.geeksforgeeks.org/python/file-flush-method-in-python/#:~:text=The%20flush,For%20Example) File flush() method in Python - GeeksforGeeks

<https://www.geeksforgeeks.org/python/file-flush-method-in-python/>

[[3]](https://www.weiy.city/2020/08/how-to-write-and-read-json-file-by-qt/#:~:text=QJsonDocument%20document%3B%20document,8%22%20%29%3B%20iStream%20%3C%3C%20bytes) How To Write And Read JSON File By Qt – weiy.city

<https://www.weiy.city/2020/08/how-to-write-and-read-json-file-by-qt/>

[[4]](https://stackoverflow.com/questions/55290590/qt-command-log-using-qlistwidget#:~:text=I%20think%20,formatting%20features%20provided%20by%20QTextEdit) [[5]](https://stackoverflow.com/questions/55290590/qt-command-log-using-qlistwidget#:~:text=I%20think%20,formatting%20features%20provided%20by%20QTextEdit) [[6]](https://stackoverflow.com/questions/55290590/qt-command-log-using-qlistwidget#:~:text=log) [[7]](https://stackoverflow.com/questions/55290590/qt-command-log-using-qlistwidget#:~:text=,after%20adding%20the%20text) [[8]](https://stackoverflow.com/questions/55290590/qt-command-log-using-qlistwidget#:~:text=ui,Log) c++ - Qt Command Log using QListWidget - Stack Overflow

<https://stackoverflow.com/questions/55290590/qt-command-log-using-qlistwidget>