ChatGPT

Unified GSR & Dual-Video Recording System – Comprehensive Plan and Documentation

Revised System Requirements

System Architecture and Multi-Device Control

- **Central PC Controller**: A desktop application serves as the central control hub for the system. The PC can connect to and manage multiple Android smartphones concurrently, coordinating recording across several phones in sync [1] [2]. All phones are orchestrated from the PC for unified start/stop and configuration. The PC application also records its **own** data (e.g. a USB webcam feed) as another synchronized stream [3], acting as an additional sensor node.
- **Remote Recording Control**: Users can initiate or stop recordings and adjust settings on any connected Android device **remotely** from the PC interface. For example, the PC can remotely toggle which streams to capture on each phone, set frame rates, etc. Each phone should acknowledge and execute commands from the PC (e.g. *Start Recording*, *Stop Recording*) with minimal latency [4]. This bi-directional communication ensures the operator can centrally control all devices.
- **Standalone Phone Operation**: Each Android phone is also capable of independent operation when needed. In the absence of a PC connection, the phone app can start/stop recordings and manage sensors on its own via the local UI [5]. This guarantees data collection is still possible if the PC is unavailable. When a PC later connects, it can sync up with the phone, retrieve status, or download recorded data.
- **Multiple Device Support**: The system supports multiple smartphones recording in parallel, all synchronized under one session. The PC interface distinguishes each device (e.g. by a unique device ID or name) and handles multiple incoming video streams and sensor streams simultaneously [2]. This enables **synchronized multi-angle recordings** – for example, two phones capturing different viewpoints with thermal, RGB, and GSR data in unison.
- **PC Camera Integration**: The PC itself can act as a sensor node. The desktop app can record from a **USB webcam** (e.g. a high-quality Logitech camera or any UVC-compliant camera) connected to the PC, treating the PC's camera feed as another video stream in the system [3]. This feed is time-synchronized with the phone data, so that footage from the PC's perspective aligns with the mobile data streams. The PC app should be able to record this video locally and/or stream its frames into the synchronization framework so it aligns with the phones' timelines.

Camera and Sensor Data Acquisition on Phones

- **Dual Camera Streaming on Phone**: Each Android phone will capture **two video streams simultaneously**: (1) an RGB video stream from the phone's built-in camera (using the Camera2 API, with full control over focus, exposure, etc.), and (2) a thermal infrared video stream from a USB thermal camera (e.g. Topdon TC001 or InfiRay P2 Pro) attached via USB-OTG [6]. Both streams run in parallel and are precisely timestamped **on the phone**, ensuring each thermal frame can be correlated to the corresponding RGB frame [7]. The phone app must efficiently manage this dual-camera pipeline so that thermal and visible videos remain in lockstep.
- **Shimmer GSR Sensor Integration**: The system integrates the **Shimmer3 GSR+** sensor (or a similar wireless GSR device) as a core component. Each phone can connect to a Shimmer GSR

unit via Bluetooth Low Energy to collect galvanic skin response data in real time [8]. The GSR sampling rate should be high (configurable from 128 Hz up to 1024 Hz) with low latency (~20 ms) to capture rapid changes in skin conductance [9]. The app should support Shimmer's multi-range settings (covering 10 kΩ to 4.7 MΩ resistance with auto-ranging) to accommodate different skin conductance levels [9]. This ensures accuracy across both low and high perspiration conditions.

- **Flexible Sensor Attachment**: Each phone will typically pair with its **own** GSR sensor via BLE. However, the architecture should allow flexibility in sensor assignment (for instance, if a phone needed to read from a central hub or a different device's sensor). By default, each Android device handles its own GSR sensor over Bluetooth for simplicity. This modular approach means adding or reassigning sensors (e.g. one Shimmer broadcasting to multiple phones) could be supported in the future with minimal changes.

Data Recording and Storage

- **On-Device Data Storage**: All high-resolution raw data is recorded locally on each phone during a session. The Android app saves the RGB video, thermal video, and GSR readings to the phone's internal storage or SD card while recording [10]. Storing data locally avoids network bottlenecks and ensures no data is lost if connectivity to the PC is interrupted [11]. Each modality is saved in an appropriate format: e.g. video streams to MP4 files (or sequential image frames), GSR to a CSV log or binary file, etc. [12]. By recording on-device, the system guarantees that each phone retains a complete copy of its data that can later be merged with others.
- **Timestamped Data Files**: All recorded files (videos and sensor logs) include timing information so they can be aligned post-hoc. Video files should contain frame timestamp metadata (or accompany a separate timeline log) and sensor logs should timestamp each sample [13]. The timestamps used in files correspond to the same synchronized clock as the live system (see **Time Synchronization** below), so that data from different devices can be aligned by timestamp during analysis [14]. In practice, this means if two phones record independently, their data can later be merged on the PC by matching the timestamp fields.
- **Offline Data Transfer**: After a recording session, the system will support transferring the recorded data from each phone to the PC for centralized storage and analysis. This could be done via USB cable, Wi-Fi file transfer, or any secure method as convenient [15]. The process should be as seamless as possible, e.g. the PC app might assist in pulling files from each connected phone. The transfer should preserve directory structures and file naming so that data from multiple devices remains organized (for example, each phone's data in a separate folder labeled by device name or participant ID) [16]. Easy aggregation of data is important for researchers to perform combined analysis.
- **Data Export and Format**: The system should facilitate exporting the synchronized dataset in standard formats for analysis. One recommended approach is to integrate with the **Lab Streaming Layer (LSL) LabRecorder** on the PC to save all incoming streams into a single **XDF** file (Extensible Data Format) [17]. An XDF can contain multiple data streams (thermal video, RGB video, GSR, etc.) with aligned timestamps, which is ideal for replay and analysis in tools like MATLAB, Python, or EEGLAB. If using XDF, the PC app would record the streams live as they arrive. Alternatively, the phone can save data in common formats (MP4, CSV, etc.) and the user can later import or synchronize them manually. In any case, consistency in format and timestamp usage across devices is crucial – e.g. ensuring all videos use the same encoding and all logs use a common timestamp base – so that merging data from multiple phones is straightforward [18].
- **Storage Management**: Given the potentially large volume of data (especially dual video streams), the system should include basic storage management options. For example, allow the user to choose the save location (internal phone memory vs. SD card) and provide warnings if

storage is running low [19]. The app might also enforce or suggest a maximum recording duration or split recordings into multiple files of a manageable size to avoid extremely large files [19]. These measures ensure that long recording sessions do not accidentally exhaust device storage or produce unwieldy files.

Time Synchronization and Clock Alignment

- **On-Phone Timestamping**: Each Android phone timestamps its own data at the moment of capture to avoid any network-induced uncertainty. The app uses Android's monotonic clock (`SystemClock.elapsedRealtimeNanos()`) as a unified time base for all sensor modalities [20]. Every video frame and every GSR sample is tagged with this local timestamp as it is generated. For example, if the phone streams video frames to the PC, it can periodically send timing metadata (such as RTCP packets in the case of RTP video streaming) to inform the PC of the mapping between the frame timestamps and the phone's clock [21]. By doing so, even if network transmission has variable delay, each piece of data can later be realigned according to the original capture time. This ensures millisecond-level alignment of thermal, RGB, GSR, and audio streams *on the same device*.
- **Continuous Clock Alignment**: In a multi-device setup, the system must keep the clocks of all devices in sync over time. The Lab Streaming Layer (LSL) framework, if used, inherently handles clock synchronization by sending periodic time sync packets and computing offsets between each phone and the PC [22]. If a custom networking approach is used instead of LSL, the system should implement a similar mechanism: for example, when a phone connects to the PC, perform a handshake where the phone sends a timestamp from its clock and the PC compares it to its own clock to calculate an offset [23] [24]. This offset can then be applied to all incoming timestamps from that phone. The system should periodically re-check and correct for any drift (clock differences) during long recordings [22], so that synchronization stays tight (e.g. within a few milliseconds) even over extended sessions. All devices could also synchronize to a common time reference (such as an NTP server or having the PC act as a time server) to minimize initial offsets.
- **Low-Latency Monitoring**: To make real-time monitoring practical, the streaming and synchronization approach should introduce minimal latency. Using high-bandwidth transports like Wi-Fi Direct for video streams helps keep latency low (a few hundred milliseconds or less) [25]. The PC should be receiving data nearly in real-time to display previews from each camera. If using LSL, it supports real-time streaming efficiently. In cases where video preview is needed with ultra-low latency, specialized protocols (RTP/RTSP or even WebRTC) could be employed for the video stream, but this is an optional enhancement. The key requirement is that the time from capture on the phone to display on the PC is short enough (sub-second) to allow the operator to confidently observe and direct the experiment in real time [25]. Even if network latency varies, the **recorded** data remains accurately timestamped; low latency is mainly a usability factor for live viewing.
- **Coordinated Multi-Device Start/Stop**: The system must ensure that when recording is initiated across multiple devices, they all begin capturing at the same synchronized moment (similarly for stopping). To achieve this, the PC controller can send a **"start recording" command** to all connected phones (or to a designated master phone) with a timestamp or short countdown (e.g. "start in 3…2…1…") [26]. The Android app's networking layer already supports receiving a networked start signal and uses a `CountDownLatch` to coordinate its internal modules [27], so extending this to an external command is feasible. All devices thus start (and later stop) within a few milliseconds of each other. This coordination, combined with each device's local timestamping, means the resulting data streams are aligned from the moment of onset.

Extensibility and Future-Proofing

- **Modular Sensor Architecture**: The software design is modular to allow future extensions. New sensor modalities or data streams should be addable with minimal changes to the existing system. For example, if a new physiological sensor (like ECG or respiration) needs to be included, a new capture module and data event type can be defined and plugged into the framework similar to the existing ones [28] [29]. The synchronization and networking subsystems are generic, so they can handle additional streams as long as they are timestamped and serialized in the standard way. This modularity is reflected in both the Android and PC code organization (each modality implementing common interfaces), ensuring that the platform can evolve into a general multimodal sensing system.
- **Flexible Configuration Protocol**: The communication between PC and phones should be designed to negotiate available sensors and settings. As the system may not always have the exact same set of sensors on each phone, the PC should query or be informed of what streams a connected device supports (e.g., one phone might have an extra sensor attached) [30] [31]. The protocol for configuration commands can include fields for device capabilities, so that the PC can dynamically adjust (for instance, only request data from sensors that exist on that device). This makes the system more robust to different hardware setups and allows incremental upgrades.
- **Reliability and Fail-Safes**: The design emphasizes robust operation even in real-world conditions. If a phone temporarily drops connection or a sensor fails, the system should handle it gracefully (e.g., attempt reconnection, buffer data, and alert the user). Battery usage and performance are monitored so that long recording sessions remain feasible (the app might log CPU usage or frame rate via a performance monitor) [32]. The aim is to produce **"field-ready"** data collection software that can run outside the lab without constant supervision.
- **Research-Grade Data Quality**: By addressing the above requirements, the application will provide a **comprehensive, synchronized multi-device recording system** that combines the strengths of mobile sensors with a powerful PC interface. It will capture thermal imagery, RGB video, audio, and GSR signals with precise time alignment across devices. Using frameworks like LSL for synchronization ensures that even with multiple phones, the data can be merged seamlessly for analysis [33]. This design philosophy – essentially creating a *"pocket lab"* – allows researchers to collect multimodal data in the field with near laboratory-grade synchronization accuracy [34] [35]. The result is a platform that meets the project's needs and can be extended in the future, opening doors to new experiments in psychophysiology and beyond.

Feature List

Below is a summary of the key features and capabilities of the multi-device GSR + dual-video recording system:

- **Synchronized Multi-Modal Capture**: Concurrent recording of multiple data streams – thermal infrared video (e.g. 256×192 @ 25 Hz), RGB video (1080p @ 30 fps), high-frequency GSR signals (128 Hz or higher), and audio (44.1 kHz) – all timestamped to a common clock for frame-level synchronization [36] [37]. The system ensures that every thermal frame, RGB frame, GSR sample, and audio sample can be aligned in time (within <50 ms skew) across all modalities for accurate multimodal analysis [38].
- **Real-Time Previews and Feedback**: Live preview of the video streams on the Android device (with the ability to toggle between RGB and thermal views) and real-time display of GSR values and status indicators during recording [39] [40]. The PC application provides a dashboard to monitor incoming data from all connected devices virtually in real time, enabling the operator to see multiple camera angles and GSR readings as they happen. This includes indicators for connection status, battery levels, and any errors (e.g. lost connection or sensor malfunction).

- **Remote Control & Multi-Device Coordination**: Centralized control of recording from the PC. The user can start/stop recordings on all connected phones simultaneously with one click [4] [41]. Settings (such as video resolution, frame rate, which sensors to enable) can be configured from the PC for each phone. A master/slave architecture is used for multi-device sync: one device (or the PC) acts as master coordinator, and other devices sync to it for start/stop timing and clock alignment [42] [2]. The PC can also control its own camera recording in sync with the phones.
- **Modular & Extensible Design**: The software is built with a modular architecture – each sensor modality (thermal, RGB, GSR, audio) is handled by a dedicated module or service, implementing a common interface [43] [20]. This makes it easy to extend or modify the system. For example, adding a new sensor (say, a heart-rate camera or an ECG device) would involve adding a new module without major changes to existing code. The Android app's package structure and the PC app's planned structure both reflect clear separation of concerns (capture, sync, networking, UI, etc.), simplifying maintenance and future development.
- **Hardware Integration via SDKs**: Seamless integration with specialized hardware: the **Topdon TC001 thermal camera** is supported via its SDK for live thermal imaging [44], and the **Shimmer3 GSR+ sensor** via the Shimmer BLE API for high-resolution GSR and PPG data [45] [46]. The app automatically detects and connects to these devices (prompting the user for required permissions, e.g. USB access for the camera or Bluetooth for the sensor). If the hardware is not present, the system can fall back to simulation modes so development and testing can continue (see below).
- **High-Fidelity Data Acquisition**: The system maximizes data quality from each modality. Thermal images are captured at the camera's native resolution (e.g. 256×192) and can be interpolated to 320×240 for display [47]. RGB video is captured at 1920×1080 with controlled focus and exposure to ensure clarity [48]. GSR data is sampled at 128 Hz by default (configurable higher) and converted to meaningful units (microSiemens) with support for the Shimmer's auto-ranging, providing accurate skin conductance readings [48] [9]. A PPG signal from the Shimmer is processed to extract heart rate in real time. Audio is recorded uncompressed (e.g. 44.1 kHz stereo WAV) for full fidelity. All data streams are buffered and timestamped such that even under high load, the timing remains consistent (frames or samples may be dropped in extreme cases to preserve overall sync rather than queue up and lag).
- **Real-Time Analysis and Bio-feedback**: Basic real-time analytics are built in. For instance, the system computes and displays **heart rate** from the Shimmer's PPG signal on the Android UI (and sends it to the PC) [49]. This is done via a simple peak detection algorithm on the IR or green PPG waveform, yielding beats-per-minute that update every few seconds. The GSR signal is displayed in real time on the phone (and critical events could be flagged, like surges in GSR). These real-time bio-feedback features allow an operator or participant to observe physiological changes live (e.g., see one's GSR rise and heart rate spike when a stressor is introduced). More complex analyses (like heart rate variability or stress index computations) can be added later thanks to the modular design.
- **Robust Error Handling & Fallback Modes**: The software is designed to handle common error conditions gracefully. If a hardware device is not found or fails during operation, the system falls back to a **simulation mode** for that modality rather than crashing [39] [50]. For example, if the Topdon camera is not attached, the Thermal module can generate a fake thermal feed or simply disable that stream while alerting the user. If the Shimmer sensor disconnects, the app will notify ("GSR sensor disconnected, attempting to reconnect...") and can temporarily revert to simulated GSR data so that the recording continues [51] [52]. All permission checks (camera, microphone, Bluetooth, storage) are done at startup and if any permission is denied, the user is prompted with a clear message – the app won't just fail silently. Networking is resilient: the Bluetooth/WiFi connections have retry logic and timeout handling to recover from dropouts [53] [54]. These measures ensure the system is **demo-ready** and can handle real-world usage where things don't always go perfectly.

- **Multi-Platform Data Collection**: The integrated platform supports both **mobile and desktop data sources**. In practice, this means you can capture data from a mobile device's sensors and a PC's peripherals in one synchronized session. For example, a study might use three phones around a subject (capturing thermal/RGB/GSR from different angles) *and* a PC with a high-end webcam and perhaps additional sensors—all recording together. The PC app aggregates streams from phones via either direct connections or LSL, and it can also incorporate local sensors (like a USB EEG device or webcam video). This flexibility makes the system suitable for a wide range of research scenarios, from an out-in-the-field experiment with just a phone and wearable sensor, to a full multi-camera lab setup.

Hardware and SDK Integration

This project involves integrating several hardware devices through their SDKs or APIs. Key integrations include:

- **Topdon TC001 Thermal Camera (USB)**: The Android app interfaces with the Topdon TC001 thermal imaging camera via Topdon's provided SDK. The integration is achieved by including the Topdon SDK libraries (`topdon-thermal-sdk.aar` along with its dependency `libusbirsdk.aar`) in the Android project [55]. These libraries provide a native interface to the camera's sensor. The app initializes the Topdon camera using this SDK (e.g. calling an `initializeTopdonSDK()` method which loads the native library) [56], and receives raw thermal frames (typically 16-bit temperature matrices). **USB permission** for the camera is handled at runtime: when the camera is plugged in, the app checks for permission and, if not granted yet, requests it via Android's `UsbManager` [57] [58]. After initialization, thermal frames are delivered via SDK callbacks. The app then post-processes these frames – for example, converting the raw temperature data to a grayscale or colorized bitmap for preview. Developers must ensure the Topdon AAR files are included in the APK (the build should bundle them in `app/libs/` as mentioned) so that the camera works in production [59]. If the Topdon SDK is missing, the Thermal module will default to simulation without crashing, but obviously real thermal capture will not occur [60]. (The Topdon TC001's specs: 256×192 resolution at ~25 Hz frame rate, via USB-C connection.) In summary, by using Topdon's SDK, our app can capture live thermal video and temperature readings from the TC001 and synchronize them with other sensor data.

- **Shimmer3 GSR+ Sensor (Bluetooth)**: The Android app integrates the Shimmer3 GSR+ device to collect galvanic skin response and photoplethysmogram (PPG) data. We leverage Shimmer's official **Android API** for Shimmer3 devices (the *Shimmer Android Instrument Driver* and related libraries) [61] [46]. This API (provided by Shimmer, see *ShimmerEngineering/ShimmerAndroidAPI* on GitHub) simplifies connecting to the sensor over BLE and subscribing to its data streams. In our project, the Shimmer integration involves scanning for the GSR+ device (via its Bluetooth name or MAC address), establishing a BLE connection, and then using the Shimmer API calls to start streaming GSR and PPG data. The Shimmer SDK delivers packets containing sensor readings at the configured rate (e.g. 128 Hz). Our app's **GSRCaptureModule** parses these packets – extracting the GSR value (which might come as raw ADC counts that we convert to microSiemens) and the PPG signal (infrared LED readings) [62] [46]. We then create timestamped `GSRDataEvent` objects and feed them into the sync manager just like other modalities. The Shimmer API also supports configuration, such as enabling the GSR auto-range feature (10 kΩ–4.7 MΩ) and setting the sampling rate. If for some reason we chose not to use the official SDK, we could implement the BLE service and characteristic handling manually (since Shimmer's GSR is essentially transmitted via BLE notifications), but using Shimmer's tested library is recommended for reliability. To include the Shimmer SDK in the project, we add Shimmer's AARs

or Gradle dependencies (which Shimmer provides via JFrog/Maven) – e.g., `com.shimmersensing:ShimmerAndroidInstrumentDriver` and related packages [63]. In summary, with the Shimmer Android API integrated, our app can establish a BLE link to the Shimmer3 GSR+ and continuously record high-resolution GSR and PPG data in sync with video. The Shimmer's data is timestamped on arrival to the phone (optionally using the Shimmer's internal timestamp as reference) and merged into the unified data stream.

- **Lab Streaming Layer (LSL)**: To synchronize data across the PC and phones, we utilize the Lab Streaming Layer framework. On the **Android side**, an `LSLStreamingService` (already present in the code structure) can create an LSL outlet for each data stream (thermal, RGB, GSR, etc.) [64] [65]. We will incorporate the LSL Android library (either as an AAR or by using the native C++ LSL library via JNI) so that the phone can publish streams on the network. On the **PC side**, we use *pylsl* (the Python LSL library) to resolve and subscribe to those streams. LSL automatically handles timestamp synchronization: each phone stream gets a time offset so that data arriving at the PC is timestamped in a global reference frame. Our PC app will likely run a background thread to pull samples from each LSL inlet as they arrive. The use of LSL means that adding or removing a device stream is as simple as starting or stopping an outlet – no hard-coded socket protocols for each sensor. It also means we can record data using existing LSL tools (like LabRecorder, which can save all streams to an XDF file in sync [17]). The developer needs to ensure the LSL dependency is included and that all devices are on the same network. In testing, if LSL is not available, we still have fallback networking (Bluetooth or Wi-Fi direct) to transmit data, but LSL is the preferred approach for its robust sync. Essentially, LSL in this system acts as the **time-sync glue** that binds together multiple devices' data streams with sub-millisecond accuracy, greatly simplifying multi-device coordination.

- **Other Device SDKs**: The system is designed such that additional devices can be integrated in the future. For example, if integrating a heart-rate camera or an EEG headband, their respective SDKs or APIs would be included in a similar fashion. The monorepo structure (see below) will allow placing third-party SDK libraries in a dedicated location (e.g., an `android-app/libs` folder for .aar files or as Gradle/Maven dependencies, and Python `requirements.txt` for any PC-side SDK packages). The documentation will be updated to list any such SDKs and their sources. For the current scope, the primary external libraries are: **Topdon Thermal SDK**, **Shimmer SDK**, and **LSL** (plus standard ones like OpenCV on PC for video processing, if used, and PySide6 for the UI). Each of these has been accounted for in the development plan, with instructions on how to obtain and include them so that a new developer can set up the project and have all hardware functionality working out-of-the-box.

Time Synchronization Mechanism

Accurate timestamp synchronization is central to this project. The goal is that data from different sensors and different devices can be correlated in time with high precision. Below we detail the multi-layer synchronization approach:

- **Unified Timestamps on Each Device**: Each Android phone uses a single clock (the device's monotonic system clock) to timestamp all sensor data locally [20]. In practice, whenever a thermal frame, RGB frame, GSR sample, or audio chunk is captured, the app tags it with `elapsedRealtimeNanos()` at that moment. Because we use a monotonic clock, these timestamps are immune to wall-clock changes or NTP adjustments – they only increase steadily. This guarantees **intra-device synchronization**: all data streams on the phone share a common timeline. For example, if a thermal frame has timestamp 1000000000 ns and an RGB frame

1000020000 ns, we know the RGB frame was captured 20 ms after the thermal frame on that phone. These timestamps are used both for logging (e.g., writing to file with timestamps) and for streaming (if sending to PC, the timestamp goes along with the data).

- **Global Clock Sync Across Devices**: When multiple devices are involved, we synchronize their clocks to a common reference so that, say, *10.000 s* on Phone A corresponds to *10.000 s* on Phone B and on the PC. The Lab Streaming Layer (LSL) greatly facilitates this – each LSL outlet on a phone can provide time correction info to any inlet that connects. LSL computes the offset between the phone's clock and the PC's clock by exchanging time packets (essentially an NTP-like sync) [22]. This means data arriving via LSL is timestamped in the PC's time frame automatically. If we were to use a custom networking solution (e.g., sending JSON packets over TCP), we would implement a similar mechanism: for instance, upon connection, the PC might request a timestamp from the phone, compare it to its own time, and calculate an offset. In Phase 3 of the plan, we indeed include exchanging timestamps at connect time to align clocks [23] [24]. We would also periodically re-sync during long sessions (LSL does this continuously under the hood). The result is that **inter-device synchronization** is maintained, with clock offsets typically on the order of a few milliseconds or less (LSL often achieves <1 ms after correction). All devices thereby share a virtual global clock.

- **Start-of-Recording Synchronization**: Even with clocks aligned, we also ensure that the *start* of a recording or experiment is synchronized. The PC, acting as master controller, sends a **start command** simultaneously to all devices (or in a cascade to a master phone then to others) with either an implicit "start now" or a scheduled start time [26]. The Android app's `RecordingController` uses a latch to coordinate modules, and it can also wait for an external trigger. In our implementation, when the PC user clicks "Start", the PC immediately sends out start signals; each phone receives it and starts its RecordingController (with all its capture modules) at the same moment (network latency on a local network/Bluetooth is low, but if needed we could include a tiny delay and a countdown to account for latency). This ensures that if two phones have a ~5 ms communication delay difference, they still start nearly together. At stop time, synchronization is slightly less critical but we follow a similar approach: the PC either tells all devices to stop at once or tells the master to propagate the stop. The outcome is a set of recordings (on each device) that all share the same start timestamp. Combined with clock sync, this means if Phone A started at PC-time 12:00:00.000 and Phone B at 12:00:00.005, we can adjust B's data by -5 ms to line up perfectly. In testing, we will verify multi-device start sync by looking at an event (like a clap sound or a flashlight flash) captured by all devices and confirming their timestamps match up.

- **Maintaining Sync During Recording**: Over the course of recording, especially long ones, device clocks could drift if not corrected. With LSL, continuous correction is applied – each data sample pulled in has a timestamp that has been corrected for the latest measured offset. If we use direct Wi-Fi or Bluetooth streaming for some reason, we will implement periodic sync checks (e.g., every minute, exchange a timestamp ping to recalibrate offset). Another strategy (less precise) is to sync all devices to internet time (NTP) beforehand, but since we want sub-10 ms accuracy, application-layer sync is preferred. If using Wi-Fi, having all devices on the same Wi-Fi access point and maybe using LAN time sync protocols can help, but again, LSL already covers this when used. We have also considered using the PC as a time server that phones query periodically. In summary, the system doesn't just sync once and forget – it **actively maintains clock alignment** throughout the session.

- **Latency vs. Accuracy Trade-offs**: It's worth noting that for previewing data (e.g. viewing video streams live on the PC), we accept a small latency in favor of accurate timestamps. The PC might

display a thermal frame 200 ms after it was captured on the phone, but that frame's timestamp is exactly preserved. This is fine because analysis is done on recorded data with timestamps. However, we do aim to keep preview latency reasonably low for user experience. The use of Wi-Fi Direct networking provides ample bandwidth so we don't have to buffer many frames. We target a preview latency of a few hundred milliseconds or less [25], which is achievable with direct peer-to-peer connections and no significant processing delays. If needed, we can reduce camera resolution or frame rate for preview only, while still recording full quality in the background, to keep the live view smooth.

- **Verification of Sync**: As part of development, we will include tools or logs to verify synchronization. For example, the Android `SynchronizationManager` may log timestamps and send periodic **sync markers** (a known event) to each data stream [66]. By comparing when those markers arrive on the PC from different streams, we can confirm alignment. We will also do empirical tests like moving an object in front of both an RGB and thermal camera and checking frame alignment, or tapping the GSR sensor at the same time as a visual cue and checking that the event shows up in all data at the same timestamp. These will give confidence that our synchronization mechanism is working as designed.

System Architecture

Overall Architecture: The system follows a client–server model with a **modular architecture** on each side. The **Android mobile app** functions as a data capture client, responsible for collecting multiple sensor streams and timestamping them. The **PC application** acts as the server/controller, coordinating multiple clients and aggregating their data. Both applications are built to be extensible and maintainable by separating functionality into distinct modules. Below we describe the architecture of each component and how they interact.

Android App Structure (Mobile Client)

The Android application (project **fyp-gsr-android**) is written in Kotlin and organized into packages by feature/responsibility. It employs a pattern akin to Model-View-Controller (MVC) or Model-View-ViewModel, separating the data capture, control logic, and UI. Key components of the Android app include [67] [20]:

```
app/src/main/java/com/gsrrgbt/android/
├── capture/                  # Sensor capture modules (one per modality)
│   ├── thermal/              # ThermalCaptureModule – Topdon TC001
integration [43]
│   ├── rgb/                  # RGBCaptureModule – phone Camera2 integration
 [68]
│   ├── gsr/                  # GSRCaptureModule – Shimmer GSR+ BLE
integration [69]
│   └── audio/                # AudioCaptureModule – microphone recording
├── sync/                     # SynchronizationManager – unified timestamping
& sync [20]
├── data/                     # Data model classes (DataEvent types for each
sensor) [70]
├── controller/               # RecordingController – orchestrates recording
sessions [27]
```

```
├── networking/              # Networking services (for multi-device sync)
│   ├── NetworkingManager – selects BT, Wi-Fi Direct, or LSL modes [71]
│   ├── BluetoothNetworkingService – phone-to-phone BT communication [72]
│   ├── WiFiNetworkingService – Wi-Fi Direct data exchange [73]
│   └── LSLStreamingService – stream data to PC via Lab Streaming Layer [64]
└── ui/
    └── MainActivity.kt      # UI layer (start/stop controls, live previews,
status) [74]
```

(Directory structure summarized from the Android app repository)

As shown above, each **capture module** implements a common interface (defined in `CaptureModule.kt`) for starting, stopping, and providing status of that sensor [43]. The **SynchronizationManager** (`sync/`) maintains the master clock timestamp and synchronizes events between modules (it effectively timestamps each piece of data and can insert sync markers) [20]. The **Data** classes (`data/`) define unified structures (like `ThermalFrameEvent`, `GSRDataEvent`, etc.) which carry both sensor readings and a timestamp [70]. These are serializable (we use Kotlinx serialization to JSON) so they can be logged or sent over the network. The **RecordingController** (`controller/`) manages the lifecycle of all modules – when a recording starts, it initializes all enabled modules and uses a `CountDownLatch` to make sure they all begin together [27]. It also coordinates stopping and resource cleanup, and interfaces with the networking layer to notify other devices or respond to remote start/stop commands. The **Networking** layer (`networking/`) contains services for different networking methods: Bluetooth, Wi-Fi Direct, and LSL. All implement a common `NetworkingService` interface so the app can switch between them (or even run multiple). For example, the `BluetoothNetworkingService` might implement a simple master-slave protocol over RFCOMM or BLE to sync two phones, whereas `LSLStreamingService` publishes streams to a PC [75] [64]. The **UI layer** (`ui/`) is kept minimal – primarily `MainActivity` which provides a start/stop button, options to select devices or modes, and a preview Surface for video. The UI observes status updates from the controller and modules (e.g., "Thermal camera connected" or live GSR value display) and presents them to the user.

This modular design on Android means, for example, the GSR module can be developed and tested in isolation (using a simulator if needed) and it communicates with the rest of the app through well-defined interfaces (posting `GSRDataEvent` to the sync manager, etc.). The Android app is thus both flexible and robust: if a module fails or is not present, it doesn't crash the entire app – the controller can disable that module and continue with others (with appropriate user warnings). The architecture also aligns with common best practices for sensor apps where a background service handles data collection while the UI thread remains free for interaction.

PC App Structure (Desktop Controller)

The PC application (project **fyp-gsr-windows**, to be eventually merged into a monorepo) is written in Python (3.x) and uses the **PySide6 (Qt6)** library for its GUI. It also interfaces with a C++ backend for efficient video capture and possibly high-performance processing (the Windows app originally used a C++ DLL for camera capture and OpenCV). The refactoring plan reorganizes the PC app to mirror the modular structure of the Android app for consistency and maintainability. The proposed structure for the PC app is as follows:

```
pc-app/
├── fypgsr/                      # Main Python package for the app
│   ├── ui/                      # GUI components (Qt windows, dialogs,
widgets)
│   ├── core/                    # Core logic (session management, data
logging, timing)
│   ├── capture/                 # Data acquisition modules (PC webcam, other
sensors, C++ integration)
│   ├── network/                 # Networking & device integration (e.g.
DualPhoneManager for phone comm)
│   ├── config/                  # Configuration management (loading/saving
settings)
│   └── utils/                   # Utility functions and helpers
└── ... (other files like main.py entry point, resources, etc.)
```

(Proposed directory layout based on the Windows refactoring plan) [76] [77]

In this structure, the **UI** layer (PySide6) is separated from the core logic. For example, there might be a `MainWindow` class in `ui/` that presents buttons and video canvases, and it communicates with a controller class in `core/` that manages the recording session. The **core** module would handle things like coordinating multiple device streams, starting/stopping recordings on command, collating data from different sources, and writing combined logs. The **capture** module on PC side handles inputs like the PC's own camera (this could use OpenCV or DirectShow via C++ for high performance) and possibly other local sensors (if, say, the PC had its own Shimmer or other devices connected). The PC's capture components might interface with the same C++ engine that was used previously, but now organized into a clear module. The **network** module is crucial: it includes classes to discover and manage connected Android devices. For instance, a `DualPhoneManager` (as referenced in the plan) could handle pairing with phones via sockets or LSL, receiving their streams, and injecting commands [78]. If using LSL, the network module would wrap pylsl calls to subscribe to streams and could spawn threads to continuously pull data. If using a custom socket protocol, it would handle message parsing (like distinguishing GSR vs video packets). Regardless, by isolating network code here, the core logic can be agnostic to how data arrives. The **config** module will load user preferences (like default save paths, device names, etc.) and provide them to UI or core as needed. The **utils** may include helpers for things like timestamp conversion, file IO, or a small wrapper for converting raw frames to Qt images for display.

A major aim in refactoring the PC app is to **decouple the GUI from the data processing**. In the original Windows app, some UI classes directly accessed data and device logic, which made the code hard to maintain. In the new architecture, communication will likely use signals/slots or callback interfaces. For example, when the PC receives a new GSR sample from a phone, the network module could emit a signal carrying that data; the UI layer's chart or display widget, which is connected to that signal, will update itself. This way, the data flow is event-driven and modular. The UI will not block waiting for data – it will just respond when data comes. This prevents UI freezes and makes better use of threads (Qt allows non-GUI work in background threads while the main thread handles GUI refresh).

Additionally, by aligning the PC app's module structure with the Android's (e.g., having analogous capture and network modules), we make the whole system easier to reason about. A solo developer can look at the Android `ThermalCaptureModule` and find a similar `WebcamCaptureModule` on PC, or see the Android `NetworkingManager` and find a similar network coordinator on PC. This symmetry is

intentional and is a direct outcome of the Phase 2 refactoring on the PC side (modular reorganization) [79] [80].

Interaction Between Android and PC

In operation, the **Android app(s)** and the **PC app** work together as follows:

- During setup, the user starts the PC application, which begins listening or searching for phones (via Bluetooth, Wi-Fi, or LSL service advertisements). The user also launches the Android app on each phone. The phones advertise their presence (for example, via Bluetooth device name or an LSL stream with a known name).
- The PC app discovers the devices (either the user selects them from a list or it auto-connects if they are known). Upon connection, the PC might send an initial configuration (e.g., "use 1080p RGB, 30 fps, GSR enabled") to each phone. Alternatively, the phone could send its capabilities and the PC then responds with commands to start particular streams.
- When the user hits "Start" on the PC, the PC dispatches the start command to all connected phones. The phones concurrently start their recordings (thermal, RGB, GSR, etc.), timestamping from their local clocks. They also begin streaming data to the PC (if in a live streaming mode).
- As data arrives at the PC (from one or multiple phones), the PC app's network module associates each data packet or stream with the correct device and passes it to the core logic. The core logic can then, for example, log each stream to disk (perhaps tagging with the device ID) and update the UI. The PC's own video feed (if used) is similarly treated as just another data source with timestamps.
- The Sync mechanism ensures that if we later review the data, say in an XDF file or by aligning timestamps, a GSR peak on phone A and a thermal event on phone B that happened at the same moment will have the same timestamp. If using LSL, this alignment is inherently handled; if using custom, our timestamps and offsets ensure alignment.
- When "Stop" is pressed on the PC, a stop signal goes out to all phones. They finalize their recordings (closing files, etc.) and cease streaming. The PC stops its local recording as well. At this point, the user can retrieve the recorded files from each phone (the app could assist by initiating a file transfer, or the user does it manually).
- The PC can then optionally merge the data. If we recorded via LSL and LabRecorder, we might already have one file with everything. If not, we have separate files per device (which include timestamps) that we can merge in analysis software by timestamp.

Throughout this process, the architecture's design (modular, event-driven, synchronized) is what enables smooth operation. The **synchronization manager** on each phone and the equivalent time-sync logic on PC ensure consistency. The **modularity** allows each part to be developed and debugged independently (for instance, one can test just the networking between a phone and PC by simulating data, or test just the phone's ability to record to local storage without involving networking).

Finally, from a **scalability** perspective, the architecture is built to handle at least 2–3 phones initially (Phase 5 of the blueprint focuses on multi-device scaling) [81], but in principle it could handle more with some adjustments (main bottleneck being wireless bandwidth and PC processing power). Because each device's data is processed largely independently (and synchronized by timestamps), adding a new device mostly linearly increases load, which a multi-threaded PC app can handle by dedicating threads or cores to each stream.

In conclusion, the system architecture provides a clear separation of concerns, synchronized data handling, and a flexible foundation for this complex multi-sensor, multi-device application. It is designed

to be maintainable for a solo developer (with well-defined components) and robust in performance to meet the project's requirements.

Technical Audit of Android Application (FYP-GSR-Android)

Summary of the Android app's current implementation, code health, and areas for improvement as identified in the technical audit.

Code Structure and Modularity – The Android application is generally well-structured. It follows a modular design with separate packages for capture modules, sync, networking, data models, etc., as described above. The use of interfaces (e.g. `CaptureModule`) and separation of UI from core logic is a positive aspect [67] [20]. This structure aligns with the project goals of extensibility and synchronized capture. The audit confirmed that each sensor modality is encapsulated in its own class, which improves clarity. For example, there are distinct classes for `ThermalCaptureModule`, `RGBCaptureModule`, `GSRCaptureModule`, and `AudioCaptureModule`, each handling start/stop and data callbacks for that sensor [43]. The **SynchronizationManager** provides a unified timing mechanism and queues events to the logger or network, which is critical for maintaining sync [20]. The **Networking services** are abstracted behind a `NetworkingService` interface and include placeholders for Bluetooth, Wi-Fi Direct, and LSL implementations [82] [65]. Overall, the app's **architecture** is sound and reflects good software engineering practices for a research prototype.

Hardware Integration Gaps – While the code structure is solid, the audit found that some hardware-specific features are incomplete or in a "stub" state. Specifically: - The **Shimmer GSR+ integration is not finished**: The `GSRCaptureModule` currently doesn't implement actual BLE communication; it runs in simulation mode generating dummy GSR data [69]. There is a `DevicePairingDialog` in the UI that scans for BLE devices and lists the Shimmer, but after selecting the device, nothing happens – the code to initiate a connection and stream data is marked TODO and was never implemented [83]. This means **no real GSR data** can be recorded at present – a critical gap, since GSR is central to the project. Essentially, the app can find the Shimmer sensor but not actually use it. This is the top priority to fix. - The **device pairing workflow is incomplete**: As mentioned, selecting the Shimmer in the app's pairing dialog should trigger the BLE connection and preparation of the GSR module. Currently, that flow does not proceed (the app UI doesn't show an error, it just doesn't start GSR) [83]. This needs to be finished so that the user, after choosing the sensor, gets either a successful "connected" state or a clear failure message. - The **Topdon thermal camera integration is partially implemented**: The code to initialize the Topdon camera via its SDK is present and when the camera is attached, it loads the library. However, the **thermal preview is not implemented** – i.e., the app does not display the thermal image on screen even though it's capturing frames in the background [84]. The `ThermalCaptureModule` has a method stub (e.g., `renderThermalFrameToSurface`) that isn't fleshed out, so users toggling to "Thermal view" in the app would just see a blank or no change. This is a major user experience issue because the thermal camera's output cannot be visualized yet. Additionally, some aspects of thermal image processing (like calibrating temperature values to a color palette) are not done – currently it likely just obtains raw data. There's also an edge case noted: if the camera is plugged in *before* the app starts, or unplugged mid-run, those scenarios need graceful handling (requesting permission on startup, detecting disconnection) [85] [86]. - **External libraries and dependencies**: The project uses external .aar libraries for Topdon and (potentially) for LSL. The audit noted that if those AARs are not present in the project, the app will still compile because stub classes are in place (the code declares some dummy classes so it can compile without the actual SDK) [60]. However, if you run the app without the AAR, the thermal module will silently fall back to simulation (because calls to the Topdon SDK do nothing and just return default values) [59]. This is convenient for development but poses a deployment risk – if a developer forgets to include the Topdon SDK when building the release APK, the app will not actually

use the camera even if it's plugged in, and there's no obvious warning to the user besides maybe lack of thermal view. The audit recommends enforcing presence of required AARs (e.g. failing to start thermal capture if the library isn't loaded, with a user-facing error). Similarly, for LSL: if the LSL lib isn't included, the LSL networking option should be disabled or should notify the user.

Permissions and Stability – The app requests a set of permissions on startup: Camera, Microphone, Storage, Location (for BLE scanning) – as required. The audit found that these permission checks are mostly in place. If a user denies a permission, the app currently shows a simple toast like "Permission denied" and might not proceed properly [87] [88]. There is a check that prevents starting a recording if not all required permissions are granted (which is good – it won't try and then crash) [89]. However, this could be made more user-friendly by disabling the Start button until permissions are OK or by showing a dialog guiding the user to grant permissions. In testing, we should verify that if location permission (needed for BLE on newer Android) is denied, the app doesn't hang on scanning. The audit also noted that there's no special handling if, say, Bluetooth or USB is turned off – those should ideally prompt the user. Minor stability issues identified include: potential null-pointer exceptions if a module is stopped without being started (some sequences not fully guarded), but these are edge cases. Overall, once permissions are granted, the app was found to run its simulation mode reliably, and the thread model (each capture on its own thread) means the UI stays responsive.

User Interface and UX – The Android UI is minimal, aimed at an engineering audience. It's basically one screen with a preview window, a start/stop button, and some text indicators for status. The **audit conclusion** was that this is fine for now given the primary users are the developers/researchers themselves. There is no in-app tutorial or extensive instructions, which is acceptable at this stage [90]. However, for a more polished product or for usage by non-developers, one would want to add labels and guidance (for example, explaining what to do if the thermal view is blank, or how to pair the GSR sensor). One immediate UX improvement identified is to **give feedback on error conditions**. Currently, if the app fails to connect to the Shimmer or if the Topdon camera isn't detected, it may only update a status text or log a message. The audit suggests showing a quick **Toast or dialog** like "Thermal camera not found, using simulation." [91] or "GSR sensor connection lost" so that the user is aware of what the system is doing. Little touches like this can prevent confusion during a demo.

Another UI issue is that certain features that are not fully implemented (like LSL streaming, multi-device sync toggle) still appear in the UI and could confuse users. For a demo scenario, it might be wise to hide or gray-out those toggles so that the demonstrator doesn't accidentally activate a half-baked feature. The audit specifically mentioned this for the LSL option in settings – if it's not going to be used in the demo, it should be disabled to avoid mis-clicks [92].

Performance Considerations – The app includes a `PerformanceMonitor` (as referenced in code) that can log CPU usage and frame rates. Since actual hardware wasn't fully in use yet (no real GSR, no real thermal frames being processed in the UI), performance issues are theoretical. But some potential concerns identified: - The thermal image processing (once implemented) could be CPU intensive (converting 25 FPS of 256×192 data to bitmaps). The audit suggests doing this in a background thread (the Thermal module's coroutine) and that seems straightforward [93] [94]. We will follow that advice. - The networking could become a bottleneck if trying to send full raw frames over Bluetooth (likely impossible) or Wi-Fi without compression. However, the code as written doesn't send full images – it was sending just metadata or perhaps downsampled data. The audit recommends to continue not sending large data over slower channels and perhaps disable network streaming for the demo if not needed [95] [96]. - Memory usage: with recording, large video buffers and arrays are used. The audit recommends profiling memory to catch any leaks – e.g., ensure that after stopping recording, the app frees up the byte arrays or camera buffers so that long runs don't accumulate memory [97]. We will need to do this when implementing stop; possibly forcing a garbage collection after a session (not normally good

practice, but acceptable in a controlled app for demo) [98] . - Battery and thermal (device heat): running two cameras and BLE will be taxing on a phone. The audit suggests to monitor device battery drain and CPU usage during test runs and consider optimizations if needed (like reducing frame rate or resolution if the phone struggles) [32] . Since our target devices are fairly modern, we expect them to handle it, but this is something to keep an eye on.

Conclusion of Audit – In summary, the FYP-GSR-Android app is an ambitious multi-sensor capture tool that is well architected but **incomplete in key areas**. The foundational code (module separation, sync mechanism, data structures) is strong and aligns with project requirements [89] [99] . The main issues are that some crucial components were left as prototypes or stubs – notably the actual sensor integrations (Shimmer GSR, and to a lesser extent, full Topdon functionality) and parts of the UI/UX (thermal preview, user feedback). These issues could significantly affect a live demonstration (for example, having to explain that "this graph is just simulated" would undermine credibility, or not showing the thermal view on screen would fail to showcase a key feature). Therefore, the next step after the audit is to **prioritize fixes and upgrades** that address these gaps. The audit's findings directly informed the Phase-by-Phase upgrade plan that follows, ensuring that the most critical problems are solved first (enabling real sensor data and making the system robust for a demo), and that subsequent enhancements (performance tuning, multi-device features) build on a stable foundation. Android hardware and OS compatibility were also considered: the app targets Android 8.0+ which is fine, and uses Camera2 API which is widely supported. We will need to test on the actual target devices (with the thermal camera plugged in and a Shimmer connected) as soon as possible to validate all assumptions. The audit gives us confidence that no major architectural rewrite is needed – rather, it's about **completing and refining** what's there. In the next section, we outline a phased plan to implement those improvements.

Phase-by-Phase Upgrade Plan (Android App)

To improve the FYP-GSR-Android application, we have a phased upgrade plan addressing the audit findings in order of priority. Each phase has specific objectives and key tasks, ensuring the app becomes demo-ready and future-proof step by step. The most critical fixes and features are tackled first (Phase 1), followed by performance and UX enhancements (Phase 2), and finally extended capabilities like multi-device sync (Phase 3). This sequencing is designed for a solo developer working within a limited timeframe, focusing on essential functionality before nice-to-have features.

Phase 1: Core Stabilization and Hardware Integration

Objective: Complete the essential hardware integration and fix critical issues so that the app can perform its primary function – recording real multimodal data – reliably during a live demo. Phase 1 addresses the "blockers" that currently force the app to use simulated data or could cause a demo to fail. By the end of Phase 1, the app should be able to truly showcase all four data modalities (thermal video, RGB video, GSR, audio) working together in sync with actual sensors.

Key Tasks:

- **Implement Shimmer GSR+ Support (replace simulation)** – This is the top priority because real GSR data is central to the project. Using the Shimmer Android SDK or a custom BLE implementation, enable the app to connect to the Shimmer3 GSR+ device and stream data [45] [62] . Concretely:
- Initialize the BLE connection in `GSRCaptureModule`. After the user selects the Shimmer from the DevicePairingDialog, the app should call a method to connect to that BLE device (using its

MAC address or name). Upon connection, subscribe to the Shimmer's GSR and PPG data streams (the Shimmer API provides callbacks or a data handler for this) [100] [46].
- Parse incoming BLE packets to extract the GSR values (in microSiemens or raw units) and PPG samples. The Shimmer documentation defines the packet format and frequency – e.g., at 128 Hz, we expect a new sample roughly every 7.8 ms, possibly in batches. Ensure each received sample is timestamped with `SystemClock.elapsedRealtimeNanos()` at the moment of reception (this is simpler than trying to use the Shimmer's internal clock, which we can potentially ignore or use to double-check timing) [101].
- Create `GSRDataEvent` objects carrying the sensor readings and push them to the `syncManager` (e.g., by calling `syncManager.onGSRData(sample)` as the simulation mode did) [102]. This will integrate the real GSR data into the existing data pipeline (so it gets logged, synced, and potentially streamed just like simulated data was).
- Handle disconnection gracefully: if the BLE link drops, the app should notify the user (e.g., toast "GSR sensor disconnected, attempting to reconnect") and attempt an auto-reconnect a few times [103]. Possibly implement an exponential backoff for reconnection attempts. This prevents having to manually re-pair mid-session and makes the system more robust to wireless hiccups.
- **Test with the actual Shimmer hardware** as soon as possible. Verify that when you touch the GSR electrodes or change conductance, the app's displayed GSR value changes accordingly (and reasonably – e.g., moistening fingers should decrease resistance / increase conductance). Also verify PPG: though we might not display PPG waveform directly in Phase 1, ensure we receive PPG samples (maybe log them or compute a simple BPM to verify). Adjust any unit conversions if needed – Shimmer might provide either raw ADC counts or already converted µS; if raw, apply the formula from Shimmer's guide to convert to Siemens. The audit suggests that initially, just getting the raw values in and seeing them change is fine; we can refine conversion later [104] [105].

- *Trade-off note*: Using the official Shimmer API might simplify a lot of this, but if it proves cumbersome or time-consuming to integrate, implementing the BLE communication with Android's BluetoothGatt directly could be an alternative (since we really only need GSR and PPG chars). Given time constraints, choose the path that gets real data flowing reliably – the key is not to drop GSR samples and to maintain sync [106].

- **Fix BLE Device Pairing Workflow** – Complete the logic that occurs after a user selects a device in the `DevicePairingDialog`. Currently it's a stub, so implement it to (a) save the chosen device (so the GSRCaptureModule knows which device to connect to), and (b) initiate the connection procedure. This likely means calling a method in GSRCaptureModule like `connectToDevice(BluetoothDevice device)` or similar. Ensure the UI reflects the outcome (e.g., could change a status label to "GSR: Connected [DeviceName]" upon success, or show an error if connection fails). This will tie into the Shimmer support above.

- **Add Missing Microphone Permission** – Ensure the app requests and obtains the `RECORD_AUDIO` permission on startup (it appears this was not in the original manifest or code, meaning audio was always falling back to simulation) [107]. Add `android.permission.RECORD_AUDIO` to the manifest and to the runtime permission request list. After doing so, test that the app actually records real audio via the phone microphone: perform a short recording with audio enabled, then check the saved audio file or stream for presence of sound (e.g., speak or clap during recording and confirm it's captured). This fix is straightforward but important, since without it, audio data was fake.

- **Ensure Thermal Camera Initialization & Permissions** – Improve the startup sequence for the Topdon camera: on app launch or when starting a recording, check if the Topdon device is present and if the app has permission to access it. If not, call

`UsbManager.requestPermission()` for the Topdon USB device [57] [108]. This triggers the Android system permission dialog for USB (the user should check "use by default" to avoid repeated prompts). By doing this in Phase 1, we avoid a scenario where in a demo you plug in the camera and nothing happens until you manually grant permission via system UI. Instead, our app will proactively request it. Once permission is granted (or if it was already), proceed to open the Topdon camera normally.

- Also handle if the camera is plugged in while the app is running versus before launch. The app should be able to detect and initialize in either case. Using a USB broadcast receiver (if not already) to detect device attach events could be part of this.

- Essentially, this task is about making the **thermal camera "just work"** when connected, without manual fiddling in system settings during a demo [109].

- **Disable/Hide Unready Features in UI** – For the demo, features that are not fully implemented by Phase 1 should be hidden or disabled to avoid confusion. Specifically, if Lab Streaming Layer (LSL) streaming or multi-device mode switches exist in the UI (e.g., a toggle or menu), and we are not confident to show them in Phase 1, gray them out or remove them temporarily [110]. Also possibly hide the "heart rate" display if it's shown but we haven't implemented it yet (or make sure it shows something like "--" instead of a fake value). This is a "safety" step to ensure the demo operator doesn't accidentally invoke something that isn't working. It's a temporary measure – these features will be enabled in later phases when ready.

- **Validate All Required Permissions** – Double-check that after Phase 1 changes, the app now requests **Camera, Audio, Location, Bluetooth, and Storage** permissions properly. Run the app on a fresh install and ensure the permission dialog covers everything. If the user denies something critical (e.g., Location, which is needed for BLE scan on Android 12+), make sure the app stops and alerts appropriately. Possibly implement a dialog that explains "This app needs X permission to function" if any are denied, and then exits or keeps requesting. The audit noted the app currently just toasts and won't start, which is acceptable, but we can refine if needed [111].

- **Thermal Camera Preview Display** – Implement at least a basic version of the thermal video preview on the phone. This greatly enhances the demo value (people can **see** the thermal feed on the device). In `ThermalCaptureModule`, complete the `renderThermalFrameToSurface()` (or equivalent) method to draw the thermal imagery to the UI's TextureView [112] [113]. A simple approach (as suggested in the audit) is: take the ByteArray from the Topdon SDK for each frame (which likely contains 16-bit per pixel temperature values). Convert this to an 8-bit grayscale image: find the min and max values in the frame (which might correspond to coldest and warmest points) and scale the 16-bit values into 0–255 range [114]. Then create a grayscale Bitmap of size 256×192 (or 320×240 after interpolation) and fill it with those values. Draw this Bitmap to the TextureView's Canvas. This can be done using standard Android Canvas drawing on a Surface, or by using an `ImageReader` and `Surface` if needed. Given time, even a rudimentary grayscale is fine (blue-black for cold, white for hot). We don't need fancy color maps in Phase 1 (those can wait for Phase 3 if desired) [94] [115]. The main thing is to show *something* for thermal output.

- Once implemented, test by pointing the camera at something warm (like a hand or cup of hot water) and ensure the preview updates and shows differences in intensity. It might be grayscale; that's okay. As long as the audience can see the thermal variation, it's a win. If the Topdon SDK

provides a built-in way to get an RGB image (some SDKs offer a function to get a pre-converted image), we could use that to save time [116]. Otherwise, manual scaling as above.
- This preview should run on a background thread to avoid blocking the UI. The Thermal module likely already captures on a background thread (via coroutine). Just ensure that updating the TextureView is done on the UI thread (you may need to use a handler or post the Canvas draw to the main looper).

- **Result**: After this, toggling the app's view to "Thermal" mode should show the live thermal video. This is critical for the demo, as demonstrating the thermal camera's functioning is likely a highlight (e.g., showing the heat pattern of a person's face or hand) [117] [118].

- **Full Integration Test**: With all the above implemented, conduct an end-to-end test of a recording session with real hardware:

- **Setup**: Plug in the Topdon camera to the Android phone, pair the Shimmer GSR+ to the phone via Bluetooth settings (or have it in pairing mode to connect in-app), grant any permissions as prompted.
- **Start Recording**: Use the app to start a recording for a short period (1–2 minutes). Enable all modalities (thermal, RGB, GSR, audio). Observe the app during recording: check that the thermal preview is visible, the RGB preview works (if toggling), the GSR value on UI is updating (it should no longer be a static simulation pattern, but reflecting real changes), and audio indicator (if any) is working.
- **Stop Recording**: Stop the recording and ensure the app doesn't crash or freeze. After stopping, verify that each data file is saved and contains data: e.g., the **thermal video file** has frames (if we save it, maybe we only stream? But likely we save), the **RGB video file** is non-empty, the **GSR CSV** has real values (not the placeholder pattern), the **audio file** has a non-zero file size and if opened you can hear the recorded sound [119] [120].
- **Check for Exceptions**: Watch Logcat or the app's log for any exceptions during start/stop. For example, a common bug might be a null-pointer when stopping if something wasn't initialized, or if we forgot to call the Topdon SDK's release function. The audit pointed out to ensure we call `releaseTopdonSDK()` or equivalent on stop to free the camera [121] [122]. Fix any issues encountered (like if the GSR thread didn't stop properly, etc.).
- **Performance**: Also monitor performance during this test. If you notice one module lagging behind (e.g., thermal frames coming in slowly or audio stuttering), note it. It might be something to address in Phase 2 optimization. For now, just ensure nothing is critically broken (like app runs at 5 FPS or so).

By completing Phase 1, we expect the Android app to transition from a mostly theoretical implementation (with simulated data) to a **practically usable tool with real sensor inputs**. In other words, the app will genuinely record synchronized thermal video, RGB video, GSR, and audio – fulfilling the core promise of the project. This will enormously enhance credibility during any demonstration: for example, one will be able to **show their hand** in front of the cameras and see both the thermal imprint and a spike in GSR on the UI in real time (a compelling synchronized effect) [123] [124]. Phase 1 fixes also prevent "embarrassing" issues like crashes or silent failures due to missing permissions – we harden the app for basic use. Essentially, after Phase 1, we have a stable baseline: the app can collect real multimodal data on a single device and stream or store it, which is the foundation for everything else.

Phase 2: Demo-Readiness and Performance Optimization

Objective: Now that the core functionality is in place (all sensors working), Phase 2 focuses on refinements to ensure the app runs **smoothly and reliably** in a demo or real-use scenario. This includes optimizing performance (preventing any lag or data loss), improving the user interface

feedback and polish, and adding any features that, while not strictly core, are important for demonstrating the system's value (for example, basic real-time analysis like heart rate calculation, or nicer visualization of data). Phase 2 is about taking the working prototype from Phase 1 and **polishing** it so that it impresses observers and feels robust.

Key Tasks:

- **Optimize Frame Handling and Throughput**: Ensure the app can handle the data rates without frame drops or UI freezes, especially under full load (thermal + RGB + BLE + audio simultaneously). Several sub-tasks:
- *Network/Bandwidth Adjustments*: If the Phase 1 test indicates any network strain (for instance, if we attempted to stream video frames over Wi-Fi or BT and it struggled), make short-term optimizations. One likely approach is to **not send full image frames over the network in real time** if not needed [95] [96]. The code already was designed to send mostly metadata and save full frames locally (according to audit). Double-check that – e.g., if using Wi-Fi Direct, maybe we only send frame timestamps and not the images themselves (especially if we're also recording on device). If the demo doesn't require live video streaming to PC, we might even disable network streaming of video entirely for now to save CPU. Basically, avoid saturating the network or device I/O with redundant tasks.
- *Thread Priorities*: Confirm that the sensor capture threads are not being interfered with by lower priority processes. Android's `Dispatchers.Default` is used which should be okay [125]. We could experiment with raising the thread priority for the capture threads (`android.os.Process.setThreadPriority()` to a slightly higher priority) to ensure video grabbing isn't delayed by UI work [125]. However, be cautious not to starve the UI or other tasks. This is optional; only do if we see frame delays.
- *Frame Drop Policy*: The system should prefer dropping frames to lagging if overwhelmed. Check any buffering mechanism in place. The audit mentioned a `DataStreamer` dropping events if a channel is full [126]. We might tweak the buffer sizes: for example, if currently it buffers up to 1000 events, that might be too many (causing latency); reducing to, say, 100 could ensure we drop older frames rather than accumulate latency if the PC/consumer can't keep up [127]. This trade-off (completeness vs real-time performance) should be biased toward real-time for demo. We can document this and perhaps allow the setting to change later.

- *Memory Profiling*: Use Android Studio's profiler or logs to watch memory usage during a long recording [97]. Ensure that after stopping, memory usage drops back down (which indicates no major leaks). Specifically check that big objects like camera buffers or byte arrays are being freed. If we find any persistent large memory usage, explicitly null out references at stop and consider calling `System.gc()` at a safe point (like after recording stops) to hint at garbage collection [98]. While forcing GC is generally not recommended, in a controlled recording app it might be acceptable right after a recording to clean up.

- **UI/UX Improvements and Feedback**: Make the app's interface more informative and user-friendly, now that real data is flowing.

- *Status Indicators*: Refine the status text or indicators for each module. For example, have a small label or icon that turns green when each sensor is active: "Thermal: ✔", "GSR: ✔ (5.2 µS)", "Camera: ✔", etc. [128] [129]. Label the values with units (°C or arbitrary for thermal, µS for GSR, BPM for heart rate) so an observer knows what they're seeing [130] [131]. This might involve updating the UI layout to include these labels. The audit explicitly said to **add units to displayed values** to avoid confusion ("GSR: 5.2 µS" instead of just "5.2") [129].

- *Heart Rate Display*: If the Shimmer is providing a PPG, implement a basic heart rate calculation and display. The README and initial goals mentioned real-time heart rate, and audit notes indicated it was supposed to be a feature [49] [132]. We can implement a simple peak detection on the PPG signal to compute beats per minute. If the Shimmer firmware or API gives a ready-made HR, we could use that (some Shimmer devices calculate HR from PPG on-board). Otherwise, implement in our GSRCaptureModule: keep a short sliding window of the last few seconds of PPG samples, detect peaks (perhaps by threshold or derivative) and compute the time between peaks. This doesn't have to be super accurate or fancy (since it's a demo, a little smoothing is fine) [133]. Update the UI to show "Heart Rate: XX BPM". Ensure to smooth it over e.g. 5-10 seconds to avoid jumpiness [133] [134]. This will be a great visual addition – seeing the heart rate number change live can impress viewers.
- *Visualize GSR Trend*: If possible, add a tiny real-time plot of the GSR values. Even a small line graph that scrolls as GSR changes would illustrate the signal better than just a numeric value. Qt on Android or a simple Canvas graph could be used, but given time, this might be optional. At minimum, show the numeric value and perhaps a bar or something indicating magnitude.
- *Error/Info Notifications*: Implement toasts or dialogs for important events. For example, if the app falls back to simulation for any reason (like sensor disconnect), show a Toast ("Using simulated data for X"). If the user tries to start recording but something isn't ready (e.g., Shimmer not connected), show a clear message. Essentially, no silent failures. Also consider a "Recording saved to /path/file" toast after stopping, for confirmation. These little UX touches make the system feel more complete.

- *General UI Polish*: Clean up the layout so it looks okay on different screen sizes (maybe use relative layouts). Label anything unlabeled. Perhaps add a simple settings menu if needed for toggling simulation or switching network mode (though we might keep advanced stuff hidden for now). Ensure the preview scaling is correct (the aspect ratio of the camera previews should be maintained). If time permits, add an app icon and splash screen to make it look professional.

- **Enhanced Stability and Bug Fixes**: Address any smaller bugs that were observed in Phase 1 testing or new ones from Phase 2 changes. For example: make sure the RecordingController's synchronization latch always releases even if a module fails to start (maybe add a timeout to avoid hanging) [135]. Or ensure that pressing "Stop" twice doesn't cause issues, etc. These are the kinds of edge cases to smooth out so the app feels robust. By the end of Phase 2, the developer should have high confidence that the app can run through a full demo consistently without hiccups.

- **Performance Tuning for Continuous Use**: If Phase 1 revealed any heavy CPU usage, address it. For instance, the thermal image conversion – check the frame rate of the thermal preview. If it's dropping significantly below 25 Hz, consider optimizing (maybe using native code for conversion or lowering frame rate). The goal is to have **smooth previews** and data handling. On the flip side, ensure we're not doing unnecessary work: e.g., if we log data to CSV frequently, maybe buffer writes to reduce I/O overhead (though Phase 3 PC sync might make local logging secondary).

- Also consider battery optimization: perhaps add an option to disable phone screen dimming while recording (so the screen stays on during a session), but allow it to dim if idle to save battery after. Little things like that might matter in longer usage.

With Phase 2 done, the Android app should be **polished**. The optimizations reduce the chance of frame drops or slowdowns that could disrupt synchronization or cause stutters [136]. The UI improvements make it clear to any observer that all modalities are functioning in sync – e.g., seeing numbers update and icons glowing gives confidence that the system is working [137]. We also address any minor issues

left from Phase 1 testing, increasing reliability. We deliberately deferred any “heavy or risky” new features (like complex analysis or true multi-device syncing) to later, to avoid introducing instability at this stage [138]. Phase 2 is about **polishing what we have**: making the user experience smooth, the data quality solid, and the presentation impressive for a demo in terms of both visual feedback and system stability.

Phase 3: Comprehensive Multi-Device Synchronization and Networking

Objective: With the single-device use case solid, Phase 3 expands the system’s capabilities to fully support **multi-device scenarios and PC integration**. This phase fulfills the project’s goal of enabling synchronized recording from multiple Android phones and streaming the data to a central PC application. It involves implementing and testing the networking features (Bluetooth and Wi-Fi Direct communication between devices, as well as PC connectivity) and ensuring that time synchronization across devices is accurate. Essentially, Phase 3 adds the distributed aspect: whereas Phases 1–2 focused on one phone, Phase 3 connects multiple phones and the PC into one unified system.

Key Tasks:

- **Finalize Phone-to-Phone Networking (Master/Slave)**: Ensure that two or more Android devices can discover each other and synchronize recordings via direct wireless communication (for cases where a PC is not controlling them, or as a subsystem of PC control). This includes both Bluetooth and Wi-Fi Direct modes:
- *Connection UI*: In the Android app, provide a way for the user to initiate multi-device mode. For instance, a switch “Multi-Device Mode” with options to act as **Master** or **Slave**, and whether to use **Bluetooth** or **Wi-Fi Direct** for inter-device sync [139] [140]. In Master mode, the phone will host the session; in Slave mode, it will connect to a master. The UI can be simple (even a hidden debug setting is fine), but for a polished system, expose it with clear labels.
- *Bluetooth Networking*: Implement or finish implementing the `BluetoothNetworkingService`. Likely this uses Bluetooth Classic (RFCOMM socket) because BLE doesn’t have the bandwidth for video (though for just sync signals BLE could suffice). Ensure that pairing two phones via Bluetooth works: one phone (master) listens for a BT connection, the other (slave) initiates a connection to the master’s Bluetooth name/ID [141] [142]. You might integrate an in-app device list (similar to the Shimmer pairing) for selecting which phone to connect to as slave. Once connected, verify that the phones can exchange basic messages. At minimum, they should share sync signals (like “start recording now” commands). Potentially, the master phone could also receive data events from the slave if designed (though in our architecture, it might suffice that each phone just records locally and maybe the master aggregates minimal data). Test two phones: set one as master, one as slave, connect via BT, then press start on master. Both should start recording near-simultaneously. Adjust any logic (like maybe a small countdown to account for BT latency).
- *Wi-Fi Direct Networking*: This is more complex but provides high bandwidth. The code likely sets up one device as a Wi-Fi Direct Group Owner (GO) and others connect as clients. Implement the workflow: on one phone (master), call something like `startAsMaster()` for Wi-Fi service which initiates group formation [143]. On another phone (slave), call `startAsSlave()` which will scan for the Wi-Fi Direct group. We likely need a UI to select the peer device from discovered peers (or auto-select if only one found) [144] [145]. Then call connect. Once Wi-Fi Direct is connected (IP networking established between phones), the app’s networking service can use a socket or something to send data or commands. Test by sending some dummy data or just rely on it for sync. Given that ultimately the PC will likely be involved with Wi-Fi as well, the main thing is to get the phones in the same network and able to identify each other. We will ensure that our networking manager can handle multiple slaves (maybe not immediately, but plan for it).

- *Data Exchange*: Decide what data to exchange between phones in multi-device mode. For basic sync, one phone could be the time master and broadcast sync signals (like "ping" or "start now"). The code design might already envision that (the NetworkingManager selecting master or slave mode). Confirm that in master mode, the phone can send its timestamp or a sync packet to slaves periodically [146]. If not implemented, add a simple handshake: e.g., when a slave connects, have it send a "HELLO with current time" to master; master compares with its time and computes an offset, sends back maybe. Or simpler, designate master's clock as reference and have slaves adjust their timestamps by the difference. Implement a message like: *SYNC_TIME {slave_time, master_time}*. Actually, one easy approach: Master on connect immediately sends its current monotonic time; slave receives it and compares to its own `elapsedRealtimeNanos()`, thus gets offset. Use that offset to adjust all outgoing data timestamps on slave if needed (or just for display). Also, ensure that when master says "start recording", either it's immediate or scheduled – perhaps master can send "START at timestamp X" where X is a little in the future (a few hundred ms) so slaves can line up [26]. This way even if command receipt is slightly delayed, they all start at the agreed time.
- *Testing with Two Phones*: Rigorously test synchronization between two phones. For example, start them in multi-device mode and do an action like a hand clap that both phones' microphones and cameras record. After the session, compare the data (if we have logs or using PC to collect streams) to see if timestamps align. Aim for differences <50 ms ideally. Fine-tune the sync logic if necessary (e.g., if using BT which has unpredictable latency, maybe do a countdown sync or a couple of "time alignment" exchanges before start). Also verify that if master stops, slave stops. And if a slave disconnects mid-session, the master handles it gracefully (perhaps stops that device's data or warns).

- Note: Ultimately, if a PC is present, the PC can act as master, but it's still useful to have phone-to-phone sync in case of PC-less use.

- **Integrate with PC (Data Streaming)**: Enable the PC application to receive data from the phone(s) in real-time. This will likely involve the **LSL integration** or socket streaming. Given our plan, using LSL is a priority (Phase 2 of monorepo blueprint covers this). Here's the plan:

- On the Android side, ensure the `LSLStreamingService` is functional. That means including the LSL library (if not in Phase 1 already) and actually creating LSL outlets for each data type (thermal, RGB, GSR, audio). The code might have placeholders for this. Implement those: give each stream a name (e.g., "GSR", "Thermal", etc.) and a unique source ID (perhaps the phone's device ID or name) [147] [148]. Start the LSL outlets when recording starts. They will broadcast on the network (assuming phone and PC are on same Wi-Fi or Wi-Fi Direct network).
- On the PC side (which will be handled in the PC refactoring blueprint too), use the pylsl library to resolve these streams and record or display them. For now, ensure the PC can at least find and subscribe to one phone's streams. The PC might run LabRecorder or our own code; we can test initially with LabRecorder (since it can show when streams are found and record them to file). This verifies that the end-to-end pipeline works: phone -> LSL -> PC.
- If not using LSL (or in addition), implement a simple TCP socket connection from phone to PC for control. For example, the PC could send a "START" command to the master phone via a socket. But since we plan for PC to possibly just use LSL for data and maybe a separate channel for commands, decide on one approach. Perhaps easiest: use LSL for data and still use maybe Bluetooth or a separate TCP for commands if PC is in Bluetooth range. However, since PC likely not use BT, probably the PC will connect via Wi-Fi (if on same Wi-Fi or via Wi-Fi Direct group).
- *Synchronized Start via PC*: Leverage the above phone-to-phone sync or extend it such that the PC can trigger start on the master phone, which then relays to others. Or PC could directly send start to all phones if it has connections to each. In an LSL scenario, we might not do commands

over LSL easily, so possibly we use a small custom TCP server on master phone. One approach: since Wi-Fi Direct will connect the phones and PC at IP level, the PC can open a socket to the master phone's listening port. Define a simple protocol ("CMD START" etc.). The DualPhoneManager on PC side can handle that (Phase 3 of Windows plan covers a unified start/stop mechanism) [149]. Implement such that when PC sends "START", master phone starts and also tells slaves to start (the code might already do that if master gets a command).

- Test PC integration: Connect one or more phones to PC (maybe have PC join the Wi-Fi Direct group or all on same router network). Start a session from the PC interface and see that phones respond and data flows to PC. Check synchronization: the PC might merge streams (if using LSL, LabRecorder can save them combined). Evaluate if the data from multiple phones aligns in time on the PC side. If not, adjust the offset logic. Possibly incorporate an LSL time correction or PC clock exchange as mentioned earlier (e.g., PC requests each phone's clock to compute offset and instruct them to adjust). By end of this, the **multi-device, PC-coordinated recording scenario should be operational**.

- **Time Sync Verification and Calibration**: After implementing the above, perform a thorough verification of cross-device synchronization accuracy. Use the continuous alignment features: for instance, ensure that when a slave phone connects, we do perform the offset measurement handshake [146]. Maybe log the computed offset for debugging. Also, implement maybe a periodic sync message: e.g., every minute, master phone sends a sync signal which slaves compare to their time and slightly adjust if needed (though abrupt adjustments are not ideal, but minor drift could be handled gradually). LSL will handle this if used for data (since each sample has its own timestamp), but for sanity, having our own sync upkeep for command timing is good.

- **Scalability to 2+ devices**: Modify the code to handle more than one slave if not already. For example, the master might maintain a list of connected slaves (for BT and Wi-Fi) and loop through to send them commands. The PC should be able to manage multiple incoming LSL streams or socket connections. On PC side, perhaps implement a simple manager that keeps track of multiple devices' data (Phase 5 of blueprint covers multi-device stream handling on PC) [150] [151]. In Android code, test with 3 devices if possible (that might be beyond immediate availability, but at least structure the code to allow it).

- **Edge Cases & Fail-safes**: Add any needed fail-safe mechanisms for multi-device mode. For example, if a slave loses contact (BT disconnect or Wi-Fi drops), ensure the master and others don't hang indefinitely. Maybe implement a timeout: if a device doesn't respond to a sync or command, log it and possibly drop it from the session (the recording on that phone might continue locally but master stops expecting data from it). These are advanced details; implement as much as reasonable. Also ensure stopping a session stops all phones even if one had an issue (like master should send stop to all known devices even if one might have disconnected).

Outcome of Phase 3: The system will be capable of **multi-device synchronized recording**. For instance, you could have two phones and the PC all capturing data together: two phones collecting thermal/RGB/GSR, and the PC collecting its webcam and aggregating streams, all time-aligned. We will have demonstrated that pressing "Start" on the PC (or master phone) causes all devices to record in lockstep, and that the data from all devices can be merged by timestamp after (or even live via LSL on the PC). We will also have a working networking setup with both Bluetooth and Wi-Fi options, though in practice Wi-Fi Direct will be preferred for heavy data (Bluetooth might only be used for small sync messages due to limited bandwidth) [152] [141].

This fulfills the original project requirement of supporting **multiple smartphones and a PC base station** in one synchronized system. It also sets the stage for a **unified monorepo** in which the Android and PC code coexist – at this point we know what pieces are needed on both sides, which informs how we integrate the repositories (which is done in the Monorepo Integration phase next).

After Phase 3, the focus will shift from adding features to integrating and packaging the entire system (e.g., combining codebases, writing documentation, etc.), but as far as the Android app's concerned, by Phase 3 it will have all major features implemented: real sensor support, optimized performance, polished UI, and networking sync. We will proceed to merge this with the PC application development to complete the project.

Refactoring Plan for Windows Application (FYP-GSR-Windows)

(The Windows desktop app, which is written in Python, will be refactored in phases to improve maintainability, performance, and synchronization with the Android app. The end goal is to merge it with the Android project into a single monorepo. Below is the phase-by-phase plan, tailored for a solo developer timeline.)

Phase 1: Migrate GUI from PyQt5 to PySide6

Objectives: Update the Windows app's GUI framework from PyQt5 (Qt5) to PySide6 (Qt6). This ensures long-term support (Qt6) and avoids the licensing limitations of PyQt (PySide6 is LGPL, PyQt5 is GPL/commercial) [153] [154]. The aim is to make this transition with minimal changes to functionality – the UI should behave exactly as before, just using the new library backend.

Key Steps:

- **Upgrade Dependencies**: Remove PyQt5 from the project requirements and add PySide6 (likely via `pip`). In the codebase, replace all `PyQt5` imports with `PySide6` equivalents [155]. For example:
- `from PyQt5.QtWidgets import ...` -> `from PySide6.QtWidgets import ...` (and similarly for QtCore, QtGui, etc.).
- If the code used `pyqtSignal`, `pyqtSlot`, or `pyqtProperty`, replace those with `Signal`, `Slot`, `Property` from PySide6 (which are in QtCore) [155].
- **Adapt to Qt6 Differences**: Qt6 has some minor changes from Qt5 that could affect the code:
- In PySide6, some APIs might use slightly different names or enums. For instance, `QHeaderView.setSectionResizeMode()` exists in Qt5 and Qt6 but ensure usage is correct.
- One noted difference: PySide6 (like PyQt5) still often uses `.exec_()` for starting the Qt event loop (only PyQt6 changed to `.exec()` without underscore) [156]. So this might not need change, but confirm if `.exec_()` is correct. It likely is, since PySide6 aligns with Qt6 which retained `.exec()` in C++ but PySide might keep `exec_` for Python compatibility – actually PySide6 might accept both. In any case, verify how the main event loop is started and adapt accordingly.
- **Update Signals/Slots**: Convert any PyQt-specific idioms. For instance, if the code uses `@pyqtSlot` decorator, change it to `@Slot` (from `PySide6.QtCore`). If any `connect` calls used the old syntax, they should continue to work in PySide6 as it's similar. PySide6 might not require explicitly marking slots, but it's good practice to use the decorator if used before.
- **Test UI Functionality**: Run the application using PySide6 and systematically test every GUI feature:
- Windows and Dialogs: do all windows open correctly? (e.g., main window, settings dialogs, etc.)

- Buttons, Menus, Actions: clicking each button/menu does the intended action (open a file, start capture, etc.).
- Real-time displays: the app likely has a video display widget, maybe a plot for GSR, etc. Ensure that these still update properly (no errors in console about painting or threads).
- Check for any warnings or errors on the console related to Qt when running (PySide might output things like missing libraries or deprecated functions).
- Pay attention to any subtle behavior differences (e.g., focus policies, default button signals). Ideally, it behaves identical to before.
- **Dependency Compatibility**: Verify that other libraries used in the Windows app (OpenCV, NumPy, etc.) are still compatible with the environment (especially with a possibly new Python version, if any). If the app was Python 3.x already, likely fine. Confirm that PySide6 works with our Python version (should, if using 3.8+).
- **Documentation Update**: Note in the README or developer docs that PySide6 (Qt6) is now required to run the app [157]. Remove any mention of PyQt5. This way, anyone setting it up fresh will install the correct GUI toolkit.

Rationale: Using PySide6 aligns the app with Qt6, which is the current and future Qt version (Qt5 is on its way out). PySide6 being LGPL means we avoid any licensing issues that PyQt could pose [154]. Additionally, moving to Qt6 early prevents double-work later – we won't have to refactor again when Qt5 becomes unsupported. It also might offer better support for high-DPI displays and newer Windows features. The expectation after Phase 1 is that the application **runs on PySide6 with feature parity to the PyQt5 version** [158]. This provides a stable, modern UI foundation for subsequent refactoring phases.

Expected Outcome: The Windows app should launch and operate normally using PySide6. There should be no regressions in UI functionality. Internally, this change sets us up to more easily integrate with the rest of the project (since we can use Qt6 across platforms, and PySide6 might integrate better with some IDEs and tools like Qt Designer if needed). Essentially, by end of Phase 1, the GUI migration is complete and we can move forward without worrying about deprecated frameworks.

Phase 2: Modular Architecture Reorganization

Objectives: Reorganize the Windows app codebase into a cleaner, more modular structure, following a "clean architecture" or layered design. We want to clearly separate the UI layer from core logic, and create self-contained modules for distinct functionalities (capture, data processing, networking, etc.). This makes the code more maintainable and also parallels the Android app's structure, easing future integration.

Key Tasks:

- **Directory Restructure**: Reshape the source tree into logical packages (as outlined in the System Architecture section). For example [76] [77]:
- `ui/` package for all GUI-related code (windows, dialogs, widgets).
- `core/` (or `backend/`) for core application logic that doesn't depend on Qt directly (session management, state machines, data logging).
- `capture/` for data acquisition components – e.g., interfacing with the C++ camera engine or any sensors connected to PC.
- `network/` for communication code – e.g., classes handling connection with Android devices (Bluetooth, Wi-Fi, or LSL clients) [78].
- `config/` for configuration and settings management (reading config files, storing user prefs).
- `utils/` for utilities and helper functions.

- Possibly `data/` for data models or processing algorithms if needed (though Python might not need as explicit separation for small things).
- Adjust the import paths in the code to match the new module locations. Use relative imports within the project where appropriate for clarity (e.g., `from core.session import SessionManager`) [159].
- **Decouple UI from Logic**: Audit each part of the code to ensure separation of concerns:
- Move business logic out of UI classes. For example, if the main window directly started threads to capture data, refactor so that logic resides in a controller class in `core/`, and the UI simply calls a method on that controller [160]. The UI should not contain complex logic, only user interaction handling and presentation.
- Conversely, ensure core modules do not directly manipulate UI elements. They should communicate results via signals, callbacks, or a mediator. For instance, if the capture module gets a new frame, it could emit a signal that the UI layer's slot is connected to, rather than calling a UI update method directly [161] [160].
- Introduce Qt signals in core classes if needed to notify UI (PySide6 allows defining signals in Python easily). Or use a messaging system / event bus if preferred.
- **Introduce Interfaces/Abstractions**: Where appropriate, define clear interfaces or base classes for components. This is partly to mirror Android's architecture:
- For example, define an abstract base class or protocol for a **DataSource** that could be either *Android integration* or *local capture* [162]. The idea: if the PC can get data from an Android phone or from a local webcam, both implement the same interface so the rest of the system can treat them uniformly. Similarly, for receiving GSR data either from an Android's stream or a local device.
- Another case: if there's a "SessionManager" controlling a recording session, it could have an interface that doesn't depend on whether sources are remote or local.
- Essentially, mimic how the Android app has `CaptureModule` for each sensor – on PC we might have something like `CaptureSource` for each input (like an Android phone connection vs local camera) [163].
- **Update Import Paths**: After moving files, update all import statements throughout the project to match the new structure [164]. This is mostly mechanical but must be done thoroughly to avoid import errors. Utilize IDE refactoring or careful find-replace.
- **Maintain (or Create) Tests**: If any automated tests exist, reorganize them under a tests folder corresponding to new modules. If none exist, consider writing a few basic tests for core modules once reorganized (e.g., a test for a data parser or config loader). This ensures the refactoring didn't break core logic.
- **Incremental Refactoring**: Perform the reorganization in stages, module by module, and run the app frequently to ensure nothing breaks drastically [165]. For instance, move `session` and `capture` code into core/capture first, adjust imports, run; then move networking, etc. This reduces the risk of ending up with a completely broken state that's hard to debug.
- **Documentation**: Write some developer docs or comments explaining the new structure. A short README in the repo or in code comments that outlines what each package is for can help future maintainers (or your future self).
- **Parallel with Android**: As a guiding principle, try to structure it so that analogous components have analogous structure to Android. E.g., Android's `NetworkingManager` vs PC's network manager, Android's capture modules vs PC's capture sources. This isn't strictly necessary but will help when one person (you) is working across both – easier mental mapping, and will make eventual monorepo integration smoother because both sides will have similarly named parts where appropriate [166] [167].

Rationale: A modular, clean architecture makes the system easier to understand, test, and modify [166]. By structuring code by feature and layer, we reduce tight coupling. The Windows app will follow the

same separation-of-concerns philosophy used on the Android side, where each sensor and the sync logic live in decoupled components [167] [168]. This not only makes the PC app cleaner, but also simplifies maintaining both PC and Android apps in tandem – you can implement a feature on Android and have a clear place to implement the counterpart on PC.

Expected Outcome: A **cleaner project layout** with well-defined modules. Core functionality remains the same as before (we're not adding new features in this phase, just refactoring), but code readability and maintainability are greatly improved [169]. It will be easier to locate relevant code (e.g., all networking code is in one place). Changes in one module will have minimal side effects on others, reducing regression risk. New developers (or yourself after a break) can quickly grasp where things are, and the structure will accommodate new features (like adding another sensor or integrating something) with less hassle. Essentially, by end of Phase 2, the Windows app's codebase should be well-organized and "future-ready," aligning with modern Python project practices and ready for the deeper sync improvements and integration to come.

Phase 3: Improve Data Synchronization Reliability

Objectives: Fix and enhance the synchronization of data streams between the Windows app and the Android devices. In this phase, we focus on ensuring that timestamps and recording start/stop events are tightly aligned for all modalities (RGB video, thermal, GSR, etc.) across the PC and phones. This involves establishing a unified time base and robust start/stop coordination.

Key Tasks:

- **Unified Time Base between PC and Phones**: Implement a mechanism for the PC and Android devices to synchronize their clocks when connected [170]. For example:
- When an Android phone connects to the PC (over Wi-Fi or Bluetooth or any network), have the PC immediately perform a clock sync handshake. The PC could send a "TIME_REQUEST" to the phone; the phone responds with its current monotonic clock timestamp; the PC compares it with its own timestamp upon receipt and computes the offset.
- Alternatively (if using LSL), retrieve the offset that LSL calculates (LSL provides a function to get time correction for a stream). But building our own is fine too for command sync.
- Use this offset to adjust incoming phone data timestamps to the PC's timeline (or vice versa). Possibly, define the PC's clock as master reference (since ultimately data might be combined on PC). So if a phone's elapsed time is, say, 50 ms behind PC, add 50 ms to all its data times when logging or streaming on PC.
- We may maintain this offset in the `network` module for each connected device.
- **Sync Start Commands**: Utilize the (now separate) DualPhoneManager or network manager to coordinate recording commands [149]. Specifically:
- When the user clicks "Start Capture" on the PC UI, instead of immediately starting local capture, have the PC send a **synchronized start command** to the phone(s). For instance: PC sends "START_AT t0" where t0 is a timestamp a short time in the future (maybe 2 seconds from now) so that all devices can prep and then start together at t0.
- The Android apps will receive this and use their SyncManager to start at the given time (they can compare t0 to their own clock; since we did clock sync, they know what t0 is in their terms). They then actually begin capturing at that exact moment [171].
- Simultaneously, the PC schedules its own capture (webcam, etc.) to start at t0 as well. That may mean implementing a small delay or timer on PC side to begin capturing from the webcam at the coordinated moment.
- This approach ensures PC and phones all begin recording near-simultaneously (within a few milliseconds).

- If immediate start is fine (maybe network delays are negligible over Wi-Fi), we could also just broadcast "START NOW" and trust all devices to hit go nearly together, but scheduling is safer for precision.
- Implement similar for "STOP" – though stop is less timing-critical, ideally all streams end together too. A straightforward way is PC sends "STOP_NOW" to all when user stops; slight differences of a few ms won't matter much for stop.
- **Timestamps Alignment**: As data comes into the PC from various devices (including PC's own data), adjust their timestamps by the known offsets so that they align to the unified time base [170]. For example, if phone A was 5 ms behind PC, add 5 ms to all its data points' timestamps as they are recorded or displayed on PC. This way, when saving or analyzing, 12:00:00.100 on PC data corresponds to 12:00:00.100 on phone data.
- Ensure that these adjustments are done consistently in the data logging or stream handling code. Possibly encapsulate in the DeviceManager that handles multiple inlets (from blueprint) – when it pulls a sample from a stream, apply offset before storing [172] [173].
- **Testing Multi-Modal Sync**: After implementing, test with one phone and PC: start a capture that includes a common event (like a clap with sound and motion). Verify that the PC's audio/vid and the phone's data have that event at the same timestamp in their respective logs. Then test with two phones + PC, etc., to confirm alignment.
- If any drift is observed over time, consider adding periodic re-sync. But if using LSL time correction or if sessions are not extremely long, it might be negligible. LSL typically keeps streams aligned by constantly adjusting.
- **Synchronization Feedback**: Possibly provide some diagnostics, e.g., log or display the calculated time offsets, so we know our sync is working. Maybe in a debug panel show "Phone1 offset = -5ms, Phone2 offset = +2ms" etc.
- **Robustness**: Ensure that sync mechanism accounts for slight delays. For example, when sending "START_AT t0" commands, ensure all devices got the message (maybe wait for ack or just choose a sufficiently in future time). Also handle if one device fails to start (maybe retry or report).
- **Local Data Coordination**: If the PC app itself has multiple threads (one grabbing webcam frames, another logging GSR from a device, etc.), ensure those are also using a consistent clock. Usually, they'd all use PC's system clock via `time.time()` or similar. Might consider using high precision timer if needed. But Python's time should suffice (~ms precision).
- **LSL Integration** (if not already fully used): If we're employing LSL to ferry data, note that LSL inherently tags data with the source timestamp and does sync. But LSL would align everything to PC's local clock when pulling samples (each sample gets an arrival timestamp corrected). We might use that directly for logging. However, LSL's internal sync is good; in our code, we might actually rely on that rather than manual offsets. The manual system is more for command sync and for non-LSL transports (like direct socket). Since blueprint indicates using LSL, maybe lean on it for the actual data alignment, and just focus here on command/time sync for starting/stopping and any data not going through LSL (like if PC sends commands outside of LSL).
- **Document/Log Sync Behavior**: Add comments or documentation on how sync is achieved (for future maintainers). Possibly in user documentation, note that device clocks are auto-synced on connect (for transparency).

By the end of Phase 3, the PC and Android parts of the system will operate on a **shared timeline** [170]. When a recording is initiated, all devices start near-simultaneously and their data is inherently aligned in time. This is a crucial capability for any meaningful analysis of combined data.

Concretely, if the PC records a webcam video and a phone records thermal video and GSR, the frames and samples will all carry timestamps that can be compared directly. For example, one could merge the data sets and find that at time T, these are all the sensor readings across devices.

This phase addresses one of the trickiest parts of multi-device systems – it ensures no device's data is drifting or offset unbeknownst to us. The approach of exchanging timestamps and coordinating commands should get us to within a few milliseconds of accuracy, which is likely more than sufficient for our use case (physiological signals and video, where a few ms is negligible).

It's worth noting that external factors like camera exposure times or Bluetooth latencies might introduce slight jitters, but those are not systematic and are expected even in a single device scenario.

At this point, the "distributed" aspect of the system is robust: we trust that if something happens in real life (like a sudden event), all devices' data reflecting that event can be lined up and analyzed together with confidence in the timing.

With synchronization sorted, we can proceed to the final refinements and integration tasks, such as combining the repositories and adding any packaging/documentation needed (Phases 4–6 in the Windows plan and blueprint, which deal with code quality, performance, and then monorepo integration).

Phase 4: Decouple UI and Enhance Code Quality

Objectives: Further reduce coupling between the UI and core logic in the Windows app (beyond what was done in Phase 2) and fix any architectural or code quality issues identified. Essentially, phase 4 is a cleanup and fine-tuning pass: ensuring the UI is completely isolated from business logic, and addressing any "code smells" (overly long functions, unclear naming, etc.) that might hinder maintainability.

Key Tasks:

- **Review UI interactions**: Go through each UI class (windows, dialogs) and check that it doesn't directly manipulate core state. If any UI element still calls deep into logic, consider introducing a mediator or moving that logic. By end of this phase, the **UI layer should ideally only talk to core via well-defined interfaces or signals**, not reach into core objects' internals [174] [160]. For example, instead of `ui.button.clicked.connect(self.camera.startRecording)`, maybe connect it to a method in a Controller that handles starting recording (which then triggers camera, network, etc.). This might have been done in Phase 2, but Phase 4 is to catch any stragglers.
- **Further Abstract Connections**: If not already done, use signals for communication. For instance, if core needs to notify UI of something (like "frame ready" or "device connected"), ensure it emits a signal that UI listens to, rather than calling UI methods directly [161].
- **Identify Code Smells**: Search for any technical debt or messy code:
- Overly long functions: break them into smaller ones with clear purposes.
- Repeated code: refactor into utility functions to avoid duplication.
- Unclear variable or function names: rename them for clarity.
- Comment any tricky sections to explain the intent.
- If there are any remnants of old PyQt5-specific hacks, remove them or update them properly.
- **UI Responsiveness**: Ensure that all heavy tasks are off the main thread (should have been by design, but double-check). E.g., if saving a large file, do it asynchronously or at least ensure it doesn't freeze UI. If any UI operation is sluggish, look for ways to optimize or move to background.
- **Documentation & Comments**: Write docstrings or comments for core classes and methods now that the architecture is stable. Possibly create a small developer guide describing key classes and their interactions. This is the polishing step to make the codebase easily understandable.

- **Testing & Verification**: Run through all app features after these adjustments to ensure nothing broke. Because this phase might involve small changes across the code, do a thorough test of functionality: capturing data, connecting/disconnecting devices, opening files, etc. If any bugs are introduced, fix them.
- **Prepare for Integration**: Since we know monorepo integration is upcoming, think ahead if any further decoupling is needed to allow merging with Android code. For example, ensure no hard-coded paths or OS-specific assumptions (monorepo might be multi-OS). Possibly ensure that the code can be imported as a module (since in monorepo, we might call PC functionalities from a combined context). This might mean adding `__init__.py` files to packages, etc., which likely done in Phase 2 but just verify.

Rationale: This phase addresses the subtle issues that might not have been fully handled in earlier phases. By **further isolating the GUI from logic**, we ensure the application design is solid and maintainable long-term [174]. Any remaining tight coupling is removed, which reduces the chance of bugs where UI actions unexpectedly affect core state or vice versa in unintended ways. Also, by cleaning the code, we reduce technical debt – making future enhancements or debugging easier.

Expected Outcome: After Phase 4, the Windows app's code should be **clean and robust**. The architecture issues identified in the initial audit or during refactoring have been resolved. The GUI code is thin (mostly event handling and display), and the core logic is clear and well-documented. The app's performance and responsiveness should also benefit from removing any lingering inefficiencies. Essentially, we've eliminated the "hard to maintain" parts – overly entangled code, magic numbers, etc. – resulting in a codebase that a new developer could navigate with relative ease.

This sets an excellent stage for both final performance optimizations and the integration of the Windows and Android codebases. It's much easier to combine two well-structured codebases than two messy ones. So Phase 4 ensures the Windows side is in tip-top shape before monorepo integration.

Phase 5: Optimize Camera Stream & Performance Bottlenecks

Objectives: Fix known performance issues in the Windows app's data streaming, particularly the camera feed handling, and eliminate any other major bottlenecks. Ensure that the system can capture and display data in real-time on the PC without lag or excessive resource usage. Essentially, this is about boosting throughput and efficiency so the PC can keep up with multiple high-frequency streams.

Key Tasks:

- **Efficient Frame Handling (Camera Feed)**: The audit revealed an inefficiency: the current method for the PC's webcam frames likely writes frames to disk and then reads them for display [175] [176]. This is obviously slow if done for every frame. We need to implement a faster pipeline:
- Use a **shared memory buffer** or direct in-memory transfer for frames instead of the filesystem [175] [177]. For example, allocate a shared memory region (e.g., using `multiprocessing.shared_memory` in Python or a memory-mapped file) that the C++ camera capture code writes each new frame into. Then the Python UI can directly create an image from that memory address without intermediate file I/O. This might require modifying the C++ capturing component to support this (if we have source or can wrap it). If modifying C++ is not feasible, consider at least using an in-memory stream.
- Alternatively, use an in-memory stream via Qt or sockets: e.g., the C++ code could send JPEG-compressed frames over a local TCP socket, and Python reads from the socket and updates the image [178]. This avoids hitting the disk. It might add a tiny overhead for compression, but JPEG compression can actually be quite fast and the reduced data size might be beneficial.

- Either approach would remove the expensive step of writing a frame to disk for each video frame.
- Implement whichever is easier given the existing code: if the C++ engine can be changed to expose frames via shared memory or a direct callback to Python (maybe using Ctypes or Pybind), do that. Otherwise, implement a simple local socket server in C++ and connect from Python.
- **Frame Rate Throttling**: The app currently likely caps the preview frame rate at ~30 FPS (perhaps using a timer in `VideoDisplayWidget`) [179]. Ensure this mechanism remains but perhaps make it dynamic/configurable [179]. For instance, allow reducing the preview rate on slower PCs to maintain responsiveness. Possibly put a setting like "Preview FPS limit" or automatically drop to 15 FPS if performance is an issue.
- Also ensure that if frames back up in a queue, we drop older frames rather than try to catch up (similar to Android drop policy).
- **Parallel Processing**: Confirm that data processing tasks are distributed across threads. For instance:
- If one thread is handling image frame updates (feeding the GUI), and another is logging data to disk, that's good.
- If any heavy computations (like computing heart rate from PPG on PC side, if any) are being done on the main thread, offload them to a worker thread or use `QtConcurrent.run` for example [180].
- The plan suggests possibly using QtConcurrent or thread pools for tasks like computing PPG heart rate or applying colormaps to thermal images [180]. On PC side, we might not do much processing, but if we do (say, computing HRV or running some analysis), ensure it's multi-threaded.
- Also ensure writing to disk is done in a separate thread (so UI doesn't freeze during file I/O).
- **Optimize Data Logging**: Review how the PC app writes data (if it logs to CSV or similar). Batching writes is recommended [181]. For example, instead of writing each GSR sample to file one by one (which could be hundreds per second), accumulate in a buffer and write in chunks. If using Python's file I/O, use buffered I/O or use something like Pandas to write periodically. If writing video files, ensure we aren't doing it frame-by-frame in Python (prefer to let OpenCV or the capture library handle encoding).
- Essentially, reduce the overhead of frequent small writes by using larger buffered operations [182].
- **Memory Management**: Check for any memory leaks or high memory usage. Python generally handles memory, but watch out for storing entire video frames list in memory if not needed. If we use OpenCV or PIL images, make sure to drop references when done to allow GC. Also ensure the shared memory or buffer approach doesn't leak if the capture restarts (i.e., reuse or clean up properly).
- **Test Under Load**: After optimizations, test the system under a high load scenario: e.g., two phones streaming high-res video + GSR to PC, plus PC capturing its webcam, for an extended period (say 10 minutes). Monitor CPU usage (it should be high but stable), memory usage (shouldn't continuously grow), and check for any frame drops or UI jank.
- If CPU usage is maxing out one core while others idle, consider ways to better parallelize (maybe the GIL will limit, but C++ code can release GIL during frame processing).
- If video stutters, see if we need further optimization like using GPU or lower resolution for preview (but hopefully not needed).
- **Continuous Profiling**: Use tools like cProfile or PyInstrument on a test run to identify any remaining slow spots in code. For example, if handling each incoming data event has overhead we can trim, do it.
- **Final Tweaks**: Implement any minor performance-related options in the UI, like an option to disable live preview if user only cares about logging (reducing overhead). Or a toggle for "low

performance mode" that maybe drops frame rate or turns off some visuals to accommodate less powerful PCs.
- **Outcome**: By the end of Phase 5, the Windows app should be able to handle the intended data rates **in real time, without lag**. The preview should be smooth at ~30 fps for video, the GSR/PPG data should plot or log without delay, and the system should not be bottlenecked by slow I/O or unnecessary processing. CPU usage will still be significant (because we are doing a lot), but it should be spread across cores and not cause UI freezing. Any needed throttling mechanisms will prevent the app from grinding down if overloaded – it will gracefully drop data as needed to maintain responsiveness.

Expected Outcome: The PC app runs **efficiently**. Notably, the camera preview now updates directly from memory instead of via disk, providing a big boost [183] [177]. Real-time video display should no longer be the bottleneck. If someone were observing resource usage, they'd see that we utilize multi-core CPUs effectively (e.g., one thread for video decode, one for network receive, one for UI). The improvements implemented here will ensure the system scales to at least 2-3 devices as intended, and can potentially handle higher data rates if needed (within reason).

With performance concerns mitigated, we can move on to the final phase of integrating the codebases and preparing the entire project for deployment. Essentially, Phase 5 ensures that when we run everything together (multiple phones + PC), the PC app will cope with it and present the data in real-time without hiccups.

Phase 6: Prepare Monorepo Integration with fyp-gsr-android

Objectives: Finally, reorganize and package the project to combine the Windows and Android applications into a single **monorepo**. Ensure both components can coexist in one repository, share documentation, and be built/released together. Also perform end-to-end integration testing to confirm everything works in concert when combined.

Key Tasks:

- **Repository Restructure for Monorepo**: Create a new top-level repository structure that contains both the Android and PC projects side by side [184] [185]. For example:

```
fyp-gsr-monorepo/
├── android-app/    # Android Studio project for the Kotlin app
(formerly fyp-gsr-android)
│   ├── app/ ...    # (source code, resources, etc.)
│   └── build.gradle, settings.gradle, etc.
├── pc-app/         # Python PC application code (formerly fyp-gsr-
windows)
│   ├── fypgsr/ ... # (the Python package with ui/, core/, etc.)
│   ├── pyproject.toml or requirements.txt  # (for Python dependencies,
perhaps using Poetry)
│   └── ... (maybe a setup.py if needed)
├── proto/          # (optional) Protocol Buffer definitions if using,
e.g., messages.proto for command/data schemas [186] [185]
├── docs/           # Shared documentation (like README, usage guide,
etc.) [187]
```

```
├── README.md       # Combined README explaining both parts
└── ... (any common assets or config files)
```

Move or copy the content of the existing repos into these subfolders accordingly [188]. Preserve commit history if possible: one could use Git subtree merges or similar to bring in history from the two separate repositories. If not possible, at least document the merge clearly in commit messages.

- **Configure Version Control & CI**: Initialize git in the monorepo (if not done) and ensure `.gitignore` covers both Android build artifacts (e.g., `*.apk`, `build/`) and Python artifacts (`*.pyc`, `__pycache__`, maybe `venv/`). Update any continuous integration or build scripts to account for new structure (for instance, if using GitHub Actions, have jobs for building Android and testing Python).
- **Build/Test Environment Setup**:
- For Android: ensure that the Android project still builds in Android Studio in its new location (update any relative paths if needed in gradle settings).
- For Python: decide how to handle dependencies – possibly use Poetry or requirements.txt. Since this is monorepo for internal use, a simple `requirements.txt` might suffice listing needed packages (PySide6, pylsl, etc.). Confirm one can set up a virtual environment in `pc-app/` and run the app.
- Consider adding a top-level build script that can, for example, build the Android APK and package the Python app (maybe into an executable via PyInstaller) for release. This could be in `docs/` or a CI pipeline.
- **Shared Protocols/Code**: If using Protocol Buffers or any common definitions (like message schemas for network commands), put them in a `proto/` folder and have both projects use them [185]. For example, a `messages.proto` defining the structure of commands or LSL stream info. Integrate into build: maybe use Gradle's protobuf plugin for Android and Python's `protoc` for PC to generate code from that. This ensures both sides literally share the exact same definitions of any communication packets.
- If not using protobuf, ensure at least documentation of the protocol is shared, or maybe share a simple config (maybe the source_id naming scheme, etc.). But protobuf would be neat if time permits.
- **Documentation**: Create a unified README and additional docs as needed:
- The top-level README.md should give an overview of the entire system, how to set it up, and how to run it (covering both Android and PC) [187]. It can link to sub-docs for specifics (like an Android usage guide or a PC user guide).
- Include instructions for development: e.g., "to build Android app, open `android-app/` in Android Studio; to run PC app, see `pc-app/README.md` for setup".
- Document the architecture (maybe move parts of this comprehensive document into the repo docs).
- Provide usage examples: how to start a multi-device session, etc.
- **Integration Testing**: After merging, treat it as one product and do a full test:
- Setup a fresh environment (simulate a new user): e.g., clone the monorepo, build the Android app, install on phones; install Python requirements on PC, run the PC app. Ensure the instructions provided are accurate by following them.
- Run a full multi-device recording with the monorepo code to verify everything still works in this unified context. This is basically a regression test to ensure that merging didn't introduce any path issues or such.
- If any issues found (like Android app expecting assets in a different path, or PC code assuming some working directory), fix them by making paths more robust (maybe use relative paths based on code location).

- **Release Packaging**: If the project is to be delivered, consider packaging:
- Android: generate a signed APK for release or share via an internal store.
- PC: possibly use PyInstaller to create an executable so end-users don't need Python environment (optional if this is just for internal use, but if expecting non-developers to run it, an EXE would be convenient).
- Provide batch/shell scripts if needed to launch things easily (like a script to start the PC app with correct settings).
- **Preserve Git History** (if possible): Merging repos can be tricky, but one approach is to add the Android repo as a subtree in `android-app/` and same for PC. This way, commit history from each is preserved within that directory's history. If that's not done, at least archive the old repos and mention their last commit so history isn't completely lost. But the plan suggests preserving history if possible [188].
- **Final Checks**: Verify that nothing got lost in the move: check that all files (source, configs, assets, images) from both projects are present in the monorepo. Also ensure license notices or credits from each repo are carried over appropriately.
- **Unified Versioning**: Set a version number that applies to the whole system (maybe in a top-level file or at least in docs). Now that it's one project, we might use a single version scheme (e.g., v1.0 for first integrated release).
- **Combined README**: Emphasize how the combined system is used: e.g., "Install the Android app on all phones, run the PC app, ensure network connectivity, etc." Include screenshots or diagrams if helpful.
- **Note for future**: mention how to add new features in monorepo (so devs know to update both sub-projects accordingly).

Rationale: Combining into a monorepo has many benefits: single source of truth for documentation, easier synchronization of changes (you can update protocol on both sides in one commit), and simpler issue tracking. It also makes it easier for a solo dev to manage (one repo instead of two). By preparing for integration meticulously, we avoid chaos that can come from merging projects (like mismatched versions or forgetting a crucial piece).

Expected Outcome: A single repository named something like `gsr-rgbt-system` or similar, containing everything needed to build and run the entire system [189] [190]. Both the Android app and PC app can be launched from this codebase. The repository will have a professional structure with a clear README, possibly even a one-step setup for the PC (like `pip install -r requirements.txt`). It will be the repository that can be shared with others or open-sourced if desired (assuming any sensitive parts are fine), containing the unified code.

Integration testing will confirm that a fresh setup works, meaning the instructions are correct and all dependencies are accounted for. The monorepo is essentially the final deliverable of the project – everything in one place, properly organized and documented.

At this point, the solo developer (and future collaborators) can use this monorepo to maintain the project. Adding a feature that affects both Android and PC can be done in one coherent commit/pull-request. The blueprint and plans we followed ensure that the architecture and performance are solid, so going forward, development is about extending capabilities (like adding new sensors or improving algorithms) rather than fighting structural issues.

Finally, having both sides together simplifies user distribution: one could imagine a release bundle that includes the APK and the PC installer together, with instructions, which is convenient for end users of this research toolkit. Overall, Phase 6 brings the project to completion as a unified, maintainable codebase ready for use in research or demonstrations.

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 21 22 25 28 29 30 31 33 34 35 Revised Requirements for Multi-Device GSR and Dual-Video Recording System.pdf
file://file-2ADcpVAXY6Tr8swSd5tVn7

20 23 24 27 32 41 43 44 45 46 49 51 52 53 54 56 57 58 59 60 61 62 64 65 67 68 69 70 71 72 73 74 75 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 152 Technical Audit and Upgrade Plan for the FYP-GSR-Android Application.pdf
file://file-K52ojE6FaoXWsG6amNz756

26 76 77 78 79 80 149 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 Refactoring Plan for fyp-gsr-windows (Windows App) toward a Unified Monorepo (1).pdf
file://file-BSAy3ES9ituEh8pMaXGtFZ

36 37 38 39 40 42 47 48 50 55 66 README.md
https://github.com/buccancs/fyp-gsr-android/blob/c34c2cbda374e614ad8c317ad82a991b851badb9/README.md

63 GitHub - ShimmerEngineering/ShimmerAndroidAPI
https://github.com/ShimmerEngineering/ShimmerAndroidAPI

81 147 148 150 151 172 173 Monorepo Integration Blueprint_ fyp-gsr-windows & fyp-gsr-android.pdf
file://file-DUqd5f3M1h4ynxa9y7WQnS