ChatGPT

Building a Synchronized GSR + Dual-Video Multimodal System (Step-by-Step Guide)

Overview and Architecture

In this guide, we will build a **multimodal data acquisition system** capable of synchronized recording of **Galvanic Skin Response (GSR)** and **dual video streams** (RGB and thermal), using multiple Android devices and a central Windows PC. The system architecture consists of:

- **Android capture nodes** (mobile apps) that record RGB video (camera), thermal imagery (Topdon TC001 thermal camera), physiological data from a Shimmer GSR+ sensor, and audio.
- A **PC controller application** (Python/PySide6) that interfaces with a webcam (for an additional RGB stream) and optionally its own GSR sensor, provides a GUI for monitoring, and remotely controls the Android devices.
- **Cross-device communication** over Wi-Fi (TCP/IP networking or Lab Streaming Layer) and/or Bluetooth to synchronize operations and data streams. This allows *centralized control* of multiple phones and unified timestamping of all data [1] [2].
- A **unified data logger** and time-synchronization mechanism to ensure all modalities (video frames, thermal data, GSR samples, audio, etc.) share a common timeline for later analysis [3].

By the end of this guide, you will have a monorepo project containing an Android app, a PC app, shared resources, and documentation. Both the mobile and desktop components will be configured in a single IntelliJ IDEA workspace for convenience. We will proceed step-by-step, from initial project setup to implementing each feature, integrating SDKs, and testing the end-to-end system.

1. Project Setup and Monorepo Structure

a. Repository Structure: Begin by creating a new directory for the project and initializing a git repository there. Inside this root, set up a monorepo-style structure with clearly separated components:

```
gsr-multimodal-system/
├── android-app/    # Android mobile application (Kotlin + Gradle)
├── pc-app/         # PC controller application (Python + PySide6)
├── shared/         # Shared assets, configs, or documentation
└── docs/           # Documentation (setup guides, user manuals, etc.)
```

This layout groups the Android and PC codebases together for easier management. The `shared/` folder can contain any common resources (e.g. a README, JSON schemas for data, etc.), and `docs/` will hold Markdown/PDF guides or configuration files. Initialize a Git repository in the root and create a `.gitignore` (you can use standard templates for Android, Python, and IntelliJ). For example, ignore build outputs like `android-app/build/`, Python virtual envs, and IDE files. Commit the initial structure.

b. Version Control and GitHub: If you use GitHub, create a new repository and push the monorepo. Organizing both apps in one repo simplifies issue tracking and ensures all changes across Android/PC stay in sync. (Later, we will add vendor SDK files to the repo as needed – see Section 9.) At this stage, also consider setting up a README in the root to summarize the project and a `docs/README.md` or other docs outlining hardware requirements and architecture for future reference.

c. IntelliJ IDEA Workspace: Open IntelliJ IDEA (Ultimate Edition is recommended, since it has both Android and Python support). Use **File > Open** and select the root `gsr-multimodal-system` folder. IntelliJ should recognize the subdirectories; we will configure them as separate modules: - **Android Module:** If IntelliJ doesn't automatically detect an Android Gradle project, you can manually create one. Right-click `android-app` folder (or use **File > New > Module** if available) and choose **Android Application**. Follow the wizard to create a basic Android app in that directory (choose Kotlin as the language, a minimum SDK appropriate for Camera2/CameraX – e.g. API 26 or higher). This will generate an `android-app` submodule with its own `app/` structure, `build.gradle`, etc. Verify that `android-app` has a `build.gradle` (or `build.gradle.kts`) and `settings.gradle` linking the module. You should be able to open the Android Manifest and App code in IntelliJ.
- **Python Module:** Ensure the **Python plugin** is enabled in IntelliJ. Now configure the PC app as a Python module. Go to **File > Project Structure > Modules**, click "+" to add a new module, and select **Python**. Set the module content root to the `pc-app` folder. IntelliJ will prompt to create a new Python SDK (virtual environment) for this module – you can either let it create a virtualenv or use Poetry (discussed below). After this, `pc-app` should be marked as a Python module in the IDE.

IntelliJ will now treat this as a multi-module project: one Android, one Python. This allows you to edit and run both applications side by side. In the Project view, you'll see both modules. You may also create separate run configurations – one to launch the Android app (on an emulator or device) and one to run the Python app. Keeping everything in one IDE streamlines development and ensures you can easily jump between Android and PC code.

d. Python Environment Setup: Inside `pc-app/`, set up a Python project structure. For example:

```
pc-app/
├── main.py           # Entry point for the PC GUI application
├── requirements.txt  # (or pyproject.toml if using Poetry)
├── app/              # (optional) put your Python source files here for
clarity
│    ├── gui/ ...     # GUI modules, dialogs, etc.
│    ├── network/ ... # networking modules (for device comm)
│    ├── sensors/ ... # sensor integration (e.g., GSR handling)
│    └── __init__.py
└── ... (other files)
```

You can use **Poetry** for dependency management or a classic virtual environment + requirements file. Using Poetry: run `poetry init` in `pc-app` to create a `pyproject.toml` and specify dependencies (PySide6, opencv-python, numpy, pylsl, pybluez, etc.). If using venv + pip: create a venv (`python -m venv .venv`), activate it, then use `pip install` for dependencies, and freeze to `requirements.txt`. Ensure IntelliJ uses this environment for the Python module (Project Structure > SDK > add the interpreter from `.venv` or the one Poetry created).

e. Hardware and Software Prerequisites: This project assumes you have at least one Android phone (with USB-C port for the thermal camera) and a Windows PC with Bluetooth and Wi-Fi connectivity. For multi-phone setups, ensure up to 2 devices can connect (the system supports two phones simultaneously [1]). The PC should have a Bluetooth adapter (internal or USB dongle) if you plan to use Bluetooth communication [4]. All devices should be on the same Wi-Fi network for TCP/IP communication. Install manufacturer drivers or SDKs if needed (e.g. the Topdon camera might require enabling USB host mode on the phone; the Shimmer sensor may need to be paired via Bluetooth). Finally, **synchronize the system clocks** as much as possible (via internet time or manual sync) – this will simplify timestamp alignment later if not using a specialized sync mechanism.

With the initial scaffolding in place, we can now implement each component of the system in turn.

2. Developing the Android Capture App (Kotlin)

We will now build the Android application that runs on each mobile device to collect data. The app will capture four modalities in parallel: **RGB video**, **thermal video**, **GSR sensor data**, and **audio**. It will also handle sending real-time previews and data to the PC and responding to remote control commands. We'll construct the app step by step:

2.1 Android Project Configuration and Dependencies

Open the `android-app` project in IntelliJ. In the `build.gradle` (Module: app) file, add the necessary permissions and dependencies:

- **App Permissions:** In `AndroidManifest.xml`, request `CAMERA`, `RECORD_AUDIO`, `WRITE_EXTERNAL_STORAGE` (if saving files to external storage), and `BLUETOOTH`/`BLUETOOTH_ADMIN` (for older Android) or `BLUETOOTH_CONNECT` (for newer Android 12+), `BLUETOOTH_SCAN` if needed, and `INTERNET` (for Wi-Fi communication). Also add `android:usesCleartextTraffic="true"` in application manifest if you plan to use raw TCP sockets with an IP (not needed for LSL).

- **Gradle Dependencies:** Add libraries for camera and networking:

- If using **CameraX** for ease of camera handling: include `androidx.camera:camera-core` and `camera-camera2`, `camera-lifecycle`, `camera-view` artifacts with appropriate versions.
- For video recording, include `androidx.camera:camera-video` or use the older MediaRecorder APIs.
- Add the **Topdon TC001 SDK** library. The Topdon SDK may be provided as an `.aar` package. If you have an AAR or JAR for it, place it in `android-app/app/libs/` and add `implementation files('libs/TopdonSDK.aar')` in Gradle. Alternatively, if Topdon provides a Maven dependency, add that repository and dependency. (For example, some IR camera SDKs use `com.infisense:...` – check vendor docs.)
- Add the **Shimmer Android SDK** for GSR. Shimmer provides an Android API (ShimmerAndroidInstrumentDriver, ShimmerBluetoothManager, etc.). You can include it via Maven JitPack/JFrog. For instance, add the Shimmer repository and dependency in Gradle. According to Shimmer's documentation:

```
repositories {
    maven { url "https://shimmersensing.jfrog.io/artifactory/
```

```
ShimmerAPI" }
}
dependencies {
    compile group: 'com.shimmersensing', name:
'ShimmerAndroidInstrumentDriver', version: '3.0.71Beta', ext: 'aar'
    implementation group: 'com.shimmersensing', name:
'ShimmerBluetoothManager', version: '0.9.42beta'
    implementation group: 'com.shimmersensing', name: 'ShimmerDriver',
version: '0.9.138beta'
}
```

This will pull the Shimmer SDK AARs from Shimmer's repository [5] [6]. (Ensure you exclude any conflicting transitive libs as noted in their instructions.) The Shimmer API will greatly simplify connecting to the GSR sensor.

- Optionally, include **LabStreamingLayer (LSL)** for Android if you plan to stream data via LSL. LSL on Android requires compiling the liblsl C++ library for Android and a Java wrapper. This is advanced: you might skip initially, but if needed, add `liblsl-android` AAR or compile from source [7] [8]. Alternatively, you may rely on your own TCP networking and not use LSL on the phone.
- Other utilities: You might use Kotlin coroutines (include `org.jetbrains.kotlinx:kotlinx-coroutines-android` for background threads), and maybe OKIO or similar for efficient data streaming.

After adding dependencies, sync the Gradle project. Also set compileSdk and targetSdk to a recent version (e.g. 33). We will use **Kotlin** and Android Jetpack libraries for modern development.

2.2 Implementing RGB Video Capture (Camera Preview & Recording)

Start by implementing the phone's **RGB camera capture** with live preview and recording. Android's CameraX library offers a simple way to preview and record video. Alternatively, use Camera2 for fine-grained control. Here we outline a CameraX approach for simplicity:

- **UI Layout:** In `activity_main.xml`, add a `PreviewView` (from CameraX) that will show the camera feed. Also add UI elements for status (like a TextView for GSR readings) and buttons to start/stop recording. For example:

```
<androidx.camera.view.PreviewView
    android:id="@+id/previewView"
    android:layout_width="match_parent"
    android:layout_height="200dp" />
<Button android:id="@+id/btnRecord" ... text="Start Recording"/>
```

(The thermal camera preview can be a separate ImageView which we will update with frames.)

- **Camera Initialization:** In your MainActivity (or a CameraFragment), use CameraX's lifecycle APIs:

```
val cameraProviderFuture = ProcessCameraProvider.getInstance(this)
cameraProviderFuture.addListener({
    val cameraProvider = cameraProviderFuture.get()
```

```
    val preview = Preview.Builder().build().also {
        it.setSurfaceProvider(previewView.surfaceProvider)
    }
    val recorder = Recorder.Builder()
        .setQualitySelector(QualitySelector.from(Quality.HD))
        .build()
    val videoCapture = VideoCapture.withOutput(recorder)
    // Select back camera
    val cameraSelector = CameraSelector.DEFAULT_BACK_CAMERA
    try {
        cameraProvider.unbindAll()
        cameraProvider.bindToLifecycle(this, cameraSelector, preview,
videoCapture)
    } catch(exc: Exception) {
        Log.e(TAG, "Camera initialization failed", exc)
    }
}, ContextCompat.getMainExecutor(this))
```

This sets up the preview. For recording, when the user (or remote command) triggers start, create an output file and start the recorder:

```
val videoFile = File(getOutputDir(), "video_$
{System.currentTimeMillis()}.mp4")
val mediaStoreOutput = FileOutputOptions.Builder(videoFile).build()
val recording = videoCapture.output.prepareRecording(this,
mediaStoreOutput)
    .withAudioEnabled()  // include microphone audio
    .start(ContextCompat.getMainExecutor(this)) { recordEvent ->
        // handle events like stop or error
    }
```

This will capture video **with audio** from the phone's microphone and save to the file. When stopping, call `recording.stop()`.

- **Permissions at Runtime:** Use `ActivityCompat.requestPermissions` for CAMERA and RECORD_AUDIO before starting camera. Ensure the preview and recording only start after permissions are granted.

With this in place, you should be able to preview the camera on the phone and record an MP4 file with synchronized video/audio. Test this basic camera functionality on the device. Once working, integrate the other sensors.

2.3 Integrating the Thermal Camera (Topdon TC001)

The Topdon TC001 thermal camera connects via USB-C. Typically, it acts as an external UVC camera with a specialized SDK for retrieving thermal data. Using the Topdon SDK (or a generic UVC library) we need to initialize the camera and get frames in real-time:

- **USB Permissions:** Android requires user permission to access USB devices. The SDK likely includes a mechanism to request permission via `UsbManager`. You might need to register a

BroadcastReceiver for `UsbManager.ACTION_USB_DEVICE_ATTACHED` and request permission for the Topdon device (identified by vendor/product ID). Alternatively, if using their SDK, call their init function which might handle permission internally.

- **SDK Initialization:** Follow Topdon's documentation. For example, if the SDK provides a class `IRCamera` or `IRUVC`, you would do something like:

```
val usbManager = getSystemService(Context.USB_SERVICE) as UsbManager
// Find the device
val deviceList = usbManager.deviceList
val tc001Device = deviceList.values.find { it.vendorId == TOPDON_VID &&
it.productId == TOPDON_PID }
if(tc001Device != null && usbManager.hasPermission(tc001Device)) {
    IRUVC.initialize(usbManager, tc001Device)  // hypothetical
initialization
}
```

Once initialized, the SDK might provide callbacks or a method to read frames. For example, Topdon's sample code or API might deliver a thermal image in YUV or a special format. Many thermal cameras provide two images: a visible (or IR grayscale) image and a matrix of temperature data. In the case of Topdon TC001 (256×192 sensor), a known approach is that it returns a YUYV image where one half of the frame contains IR data [9]. The vendor SDK likely abstracts this, giving either a thermal bitmap or separate arrays.

- **Display Thermal Feed:** Create an `ImageView` in the UI for thermal preview. In code, when you receive a thermal frame, convert it to a Bitmap. If the SDK directly gives a Bitmap (e.g., a false-color thermal image), simply post it to the ImageView:

```
runOnUiThread { thermalImageView.setImageBitmap(frameBitmap) }
```

If the SDK gives raw temperature data, you might need to convert it to a color map (perhaps using OpenCV or custom code) – check the SDK documentation. Some SDKs also provide a function to get a thermal image with a palette (e.g., ironbow, rainbow coloring). Use whichever is simplest to visualize in real-time.

- **Record Thermal Video:** Recording the thermal stream is more complex, since it's not a standard camera source. A simple approach is to record the thermal frames as a sequence of images (e.g., save each frame as PNG with a timestamp) and later convert to a video. However, a better approach is to encode on the fly: you can use Android's `MediaCodec` to encode an MP4 from byte arrays. For example, use an **MP4Muxer** or the new CameraX `Recorder` if it can accept a custom data source (CameraX doesn't natively handle external camera yet). As an interim solution, you might forego real-time encoding and instead rely on the phone to send the thermal frames to the PC for recording, or record as image sequence. Given this is a step-by-step from scratch, you might initially skip saving thermal video and just ensure the frames are available for display and streaming. We will later consider syncing it via timestamps.

Ensure to run a test: plug the TC001 into the phone, launch the app, and verify you see a thermal image updating. You may need to allow the app to use the USB device (a dialog will prompt). If the Topdon

official app works on your phone, that means the hardware is fine – try to replicate minimal functionality with the SDK in your app.

2.4 Reading GSR Data from Shimmer Sensor (Bluetooth)

Now integrate the **Shimmer GSR+ sensor** to collect physiological data. The Shimmer3 GSR unit communicates via Bluetooth (Classic SPP). We will use the Shimmer Android API for robust implementation:

- **Shimmer API Setup:** After adding the dependencies in Gradle, you will have access to classes like `Shimmer` or `ShimmerBluetoothManager`. Typically, you instantiate a Shimmer device object and connect by its MAC address. For example (pseudo-code):

  ```
  val shimmerDevice = Shimmer(DeviceBluetoothAddress)
  shimmerDevice.connect()
  ```

  With the Shimmer API, there might be a `ShimmerBluetoothManager` where you register listeners for data. Consult the Shimmer API quick start guide. In general, you will need to:
- **Pair the Shimmer sensor** with the Android device via Bluetooth settings (so you have its MAC address).
- Request `BLUETOOTH_CONNECT` permission on Android 12+.
- Use the API to connect. For instance:

  ```
  shimmerManager = ShimmerBluetoothManager(applicationContext)
  shimmerManager.connectShimmerThroughBT(shimmerMacAddress)
  ```

  (The actual API calls may differ; Shimmer's example code is the best reference.)
- Subscribe to GSR data: The API likely provides a callback or data packet event. You will configure the Shimmer to enable the GSR channel. E.g.,

  ```
  shimmerDevice.setSamplingRate(128.0)
  shimmerDevice.enableGSR(true)
  shimmerDevice.startStreaming()
  ```

  Then listen for incoming samples.

- Each GSR sample typically includes a timestamp and the conductance or resistance value. The Shimmer3 GSR+ also can provide PPG and accelerometer, but we focus on GSR.

- **Display and Record GSR:** For real-time feedback, update a TextView on the UI with the latest GSR value (e.g., in microsiemens). For recording, accumulate the data points (timestamp and value) in a buffer or write to a CSV file on the phone storage. For example, append lines like `timestamp, gsr_value` to a file. The timestamp can be the device clock in milliseconds or the Shimmer's internal timestamp (Shimmer packets often include their own timestamp). Using device clock (System.currentTimeMillis or `System.nanoTime`) ensures consistency with video times (provided clocks are synced).

- **Error Handling:** Implement callbacks for connection status. If the sensor disconnects or fails to connect, handle it (e.g., show message and allow retry). The Shimmer API should provide status updates that you can monitor [10].

Test the GSR integration by turning on the Shimmer sensor, pairing it, and running the app. You should see connection status changes (perhaps log them) and incoming GSR readings. For example, log or display something like "GSR = 430µS" to verify data is coming through.

2.5 Audio Capture

For audio, since we already enabled microphone input in the video recording (if using CameraX Recorder with `withAudioEnabled()`), the phone's ambient audio will be recorded into the RGB video file. If you prefer to record audio separately (or if not using CameraX for video), you can use Android's `AudioRecord` or `MediaRecorder` directly.

A simple approach: - Use `MediaRecorder` to capture audio to a file (like a WAV or AAC). Set source as `MIC`, output format AAC or WAV, and start recording when the session begins. - Alternatively, use `AudioRecord` to capture raw PCM in a background thread if you need to stream audio data in real-time to PC (LSL could also be used for audio streams).

However, to keep things manageable, it's recommended to let the video recording handle audio (so your RGB video file has audio track) – this synchronizes audio/video on the phone automatically.

2.6 Implementing Local Preview & Recording Logic

At this point, all modalities (camera preview, thermal image, GSR, audio) are individually working. Next, implement the coordination logic in the Android app:

- **UI Controls:** In the app's main activity, have a "Start" and "Stop" button (or a toggle). This will start or stop **recording** of all modalities simultaneously. When "Start" is pressed (or a remote start command is received, as we'll handle later), do the following in code:
- Start the RGB video recording (as in 2.2, using CameraX's Recorder or MediaRecorder).
- Start the thermal frame capture thread. If possible, also start recording those frames (e.g., by initializing a buffer or opening a `MediaCodec` encoder if implemented).
- Start GSR streaming from Shimmer (e.g., `shimmerDevice.startStreaming()` if not already started) and create a new file to log GSR data for this session.
- Start audio recording if separate from video.

Mark the start time (e.g., `sessionStartTime = System.currentTimeMillis()`) for synchronization reference.

- **Real-Time Preview:** The user holding the phone should see the RGB preview (already shown by `PreviewView`) and possibly the thermal feed (your ImageView). They can also be shown a small plot or numeric display of GSR. (For a quick graph, you could use a simple `SurfaceView` to draw a rolling waveform, or update a ProgressBar to indicate GSR magnitude.) Keep these previews running while recording so the operator knows it's functioning. Ensure these UI updates happen on the main thread (use `runOnUiThread` or LiveData/observables).

- **File Management:** Decide a file naming scheme and storage location. For example, create a new folder per session (timestamped) in the phone's storage (e.g., `/Pictures/GSRSession/`

`2025-07-01_10-00-00/`). Save `video.mp4`, `thermal.mp4` (or images), `gsr.csv`, and `notes.txt` (if any metadata) in that folder. This way, after a session, you can easily retrieve all files. You might integrate this with Android's MediaStore or just use direct file APIs. Remember to handle storage permissions on Android 10-11 (use `requestLegacyExternalStorage` in manifest or scoped storage APIs).

- **Stopping Recording:** When "Stop" is pressed, stop all streams:

- Stop the video recorder (`recording.stop()` or `mediaRecorder.stop()`).
- Stop the thermal capture thread (and finalize the video or save the collected frames).
- Stop Shimmer streaming (`shimmerDevice.stopStreaming()`) and close the GSR data file.
- Stop audio if applicable.

Ensure that all resources are closed properly to flush data to disk. After stopping, you have a complete set of recorded data files on the phone.

- **Timing and Sync Considerations:** Because each data type is recorded separately, we rely on timestamps to sync. For example, every GSR sample in the CSV should have an absolute timestamp (Unix epoch or time-of-day). The video file frames will be timestamped in its metadata (PTS), and the thermal images can be timestamped by filename or a separate log. Later, we can align these using the timestamps. For now, ensure everything uses the same reference clock (the phone's SystemClock). We will introduce more robust sync using the PC as reference in Section 4.

At this stage, **test the Android app standalone**: launch it on a phone, start a recording for a short duration, then stop. Verify the outputs: - Play back the RGB video to see if it recorded properly (with audio). - Check if thermal data was saved (if implemented). - Open the GSR CSV to see data points and timestamps. - Make sure no crashes occur when sensors are started or stopped. Adjust as needed (e.g., some flows may need delays or specific ordering, especially for the Shimmer sensor start/stop).

If all looks good, the Android capture node is ready. We will later add code to handle remote commands (start/stop triggered by PC) and streaming previews to the PC in Section 4. But first, let's set up the PC controller.

3. Developing the PC Controller App (Python/PySide6)

Now we focus on the Windows PC application which will control and synchronize multiple Android devices and capture its own data. We choose **PySide6 (Qt for Python)** for the GUI to get a flexible interface, and use Python libraries for sensor integration (webcam, GSR, networking). Ensure you have the required Python packages installed in your `pc-app` environment: - PySide6 (for GUI) - OpenCV (`opencv-python` for webcam access) - numpy (for image processing) - pylsl (LabStreamingLayer, if using LSL for sync) - pybluez (for Bluetooth communication, if needed on Windows) - pyserial (for serial COM port access, possibly for Shimmer or Bluetooth on Windows) - any other needed (e.g., PyQtGraph or matplotlib if plotting signals live).

Check that these are listed in `requirements.txt` or pyproject. In our reference system, similar dependencies are listed (PyQt5, OpenCV, numpy, pylsl, pybluez) [11] [12].

3.1 PC App Structure and GUI Design

App Structure: In `pc-app/main.py`, we'll create a PySide6 QApplication and our main window. It's wise to organize the code into modules (as hinted in the structure above). For instance, you might have: - `gui/MainWindow.py` – defines the main window class and UI layout. - `gui/DevicePanel.py` – a custom widget or section of UI to represent one phone's status (preview, indicators, controls). - `network/server.py` – code for network server (listening for phone connections). - `sensors/webcam.py` – code to interface with local webcam. - `sensors/shimmer_pc.py` – code to connect to a Shimmer over serial (if using PC-side GSR). - etc.

Design the **GUI layout** to include: - A video preview area for the PC's own webcam. - Sections for each connected Android device (if two phones, have two panels side by side or in tabs). Each section might show the phone's camera preview (small), maybe thermal preview, and status of recording. - Controls: Buttons to connect/disconnect devices, start/stop recording (for all devices simultaneously, and possibly individually), and maybe a sync status display. - Menu or settings to configure IP addresses, etc.

For simplicity, you could use Qt Designer to mock up the UI and then load the `.ui` file in PySide6. Or build it dynamically in code. The key elements: - A `QLabel` or `QGraphicsView` for each video feed (PC webcam and phone previews). - `QPushButton` for actions (connect, start, stop). - Maybe a `QListWidget` or similar to list discovered devices. - Text areas or labels for GSR values.

Example Layout: You might have a main window with a top-level horizontal splitter: left side for PC's own video, right side for a tab widget containing "Phone 1" and "Phone 2". Each phone tab contains a video preview (QLabel showing frames from phone's camera), maybe another label for thermal image, and some stats (like "GSR: -- µS", "Status: connected/recording").

3.2 Capturing PC Webcam (RGB Video)

Implement the PC's local video capture: - Use **OpenCV** to access the webcam. For example:

```
import cv2
cap = cv2.VideoCapture(0)  # 0 is default camera index
if not cap.isOpened():
    print("Error: Webcam not found")
```

You may need to set resolution or frame rate if defaults aren't suitable (use `cap.set(cv2.CAP_PROP_FRAME_WIDTH, ...)` etc). - Set up a timer loop or separate thread to grab frames from `cap` and display in the GUI. In PySide6, one approach is using a QTimer that calls a slot e.g. every 33ms (30 FPS) to retrieve a frame:

```
def updateWebcamFrame():
    ret, frame = cap.read()
    if ret:
        # Convert BGR (OpenCV) image to RGB for Qt
        rgb_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
        h, w, ch = rgb_frame.shape
        bytes_per_line = ch * w
        image = QImage(rgb_frame.data, w, h, bytes_per_line,
QImage.Format_RGB888)
```

```
        ui.webcamLabel.setPixmap(QPixmap.fromImage(image))
timer = QTimer()
timer.timeout.connect(updateWebcamFrame)
timer.start(30)  # roughly 30 ms for ~33 FPS
```

This will continuously update a QLabel (`webcamLabel`) with the current webcam frame. - **Recording PC Video:** If the PC also needs to save its webcam feed, you can use OpenCV's VideoWriter to record to a file. Alternatively, integrate with the rest of the system such that when the user hits "Start", you open a VideoWriter (`cv2.VideoWriter('pc_video.avi', fourcc, 30, (width,height))`) and inside the update loop write each frame to the video file as well. Stop and release the writer on "Stop". Ensure the fourcc and file path are appropriate (e.g., 'XVID' or 'MJPG' codec, or use MP4 if available).

Test this by running the PC app and confirming that your webcam video is visible and can be recorded to a file.

3.3 Integrating a GSR Sensor on PC (Optional)

If you want the PC to capture its own GSR (for example, if a Shimmer is connected to the PC or a different sensor), you have two options: - **Via Bluetooth/Serial:** Pair the Shimmer to the PC. On Windows, this typically creates a COM port for the device. You can then use the Python `serial` library (pyserial) to read data. Shimmer devices output a binary packet stream. Unless you have the spec or a library (like *pyshimmer*), decoding could be complex. Another approach is to run the Shimmer in *LSL mode* (Shimmer has an app or you write a small script) and subscribe via pylsl. However, given complexity, an easier route is to rely on the phone for GSR to avoid duplicate effort. - If you do implement PC-side GSR: Use `pyserial.Serial('COM3', baudrate=115200)` (find the COM port from Bluetooth settings) and read lines or bytes. The Shimmer3 can be configured to stream ASCII or binary. If ASCII, you might get comma-separated values including GSR. If using the Shimmer's own software, they might have to be started via a command. Refer to Shimmer documentation if needed. For now, this step is optional and not critical since the phone will handle GSR.

In summary, we will assume **phones provide the GSR data**, and PC simply receives it, so PC-side GSR capture is not mandatory. We will revisit data aggregation in the sync section.

3.4 Network Communication: Connecting PC and Android Devices

One of the most crucial parts is enabling communication between the PC app and the Android app(s). We need this for two reasons: **remote control** (sending start/stop commands from PC to phones) and **real-time data preview** (receiving video frames, GSR readings, etc., from phones to display on PC). We outline a TCP/IP based approach, with notes on alternatives:

- **Server on PC:** The PC will act as a **server**, listening on a known port for incoming connections from phones (clients). Using Python's socket library or an async library:

```
import socket
server_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_sock.bind(('0.0.0.0', 5000))  # listen on port 5000
server_sock.listen()
```

This opens port 5000 on the PC. You might want to make the port configurable. Ensure Windows Firewall allows this port or disable it for testing.

- **Client on Android:** On the Android app, when it starts up (or when "Connect to PC" is pressed), have it open a socket to the PC's IP and port:

```
val socket = Socket(pcIpAddress, 5000)
outputStream = socket.getOutputStream()
inputStream = socket.getInputStream()
```

You may want to do this in a background thread (using Kotlin's `GlobalScope.launch` or a Thread) to avoid blocking the UI. If the network is unavailable, handle exceptions (e.g., show "Failed to connect").

- **Protocol Design:** Define simple messages to exchange. Text-based protocols (like sending lines of JSON or CSV) are easier to debug. For example:

- PC to phone commands: `"CMD START"` or `"CMD STOP"` could be sent to trigger recording. Or JSON: `{"type":"cmd", "action":"start"}`
- Phone to PC status: `{"type":"status", "gsr": 420.0, "recording": true}` periodically, or specific messages:
  - GSR data: `"GSR, <timestamp>, <value>"`
  - Frame data: for video frames, sending raw frames is heavy; instead, send a compressed JPEG image periodically for preview. One approach: the phone can take the camera preview frame, compress to JPEG (`bitmap.compress(Bitmap.CompressFormat.JPEG, 50, outputStream)`) and send that bytes with a header. However, designing a custom binary protocol for frames is non-trivial. An easier but more bandwidth method is to use **Lab Streaming Layer (LSL)**: have the phone publish video frames (or better, just timestamps and perhaps a lower-res stream) as an LSL stream, and the PC subscribe. LSL would automatically handle timestamps and you can use LabRecorder to record if needed [1] [13]. But implementing LSL on Android is advanced. Alternatively, consider using an existing streaming solution (even RTSP or WebRTC), but that may be overkill.

Given complexity, **a reasonable compromise**: Use the network link primarily for control commands and small data (like GSR values or events). For real-time video preview, you can either: - Rely on looking at the device screens (if in the same room) – not ideal for remote. - Implement a low-rate frame sender: e.g., send one frame every second or on demand. This could be enough to verify framing without saturating network. - Or integrate an open-source streaming library later.

For now, implement **command exchange**: - When the phone connects, have it send an identifier, e.g., `"HELLO PHONE1"` (where PHONE1 could be device name or ID). The PC server thread accepts the connection, reads the hello, and associates that socket with the corresponding device panel in the GUI. - Once connected, the PC can enable the "Start" button for that phone. - On PC "Start All", you will iterate over connected phone sockets and send `"CMD START\n"` to each. On PC "Stop All", send `"CMD STOP\n"`, etc. For single phone control, send to respective socket. - The phone's listening thread will parse incoming commands. When it sees "CMD START", it will trigger the same logic as if the user pressed start on the phone (but since we want remote control, you might not even show a start button on phone UI, and exclusively control via PC in field use). - The phone can send back acknowledgment:

e.g., after starting, send `"STATUS RECORDING\n"`; or send updates like `"STATUS GSR=420.0\n"`. The PC can parse these to update the UI (e.g., show "Recording..." or update a GSR display).

Bluetooth alternative: In scenarios without Wi-Fi, you could connect PC and phone via Bluetooth. This is more complex: you would need to use PyBluez on PC to act as a Bluetooth server (SPP profile) and have the Android app connect via Bluetooth socket (BluetoothDevice.createRfcommSocket). It is possible, but data rates are lower and connectivity management is tricky. Since Wi-Fi is more straightforward and given the requirement of possibly multiple phones, TCP/IP is recommended. (The system is designed to support both Wi-Fi and Bluetooth connections for flexibility [14] [15], but implementing both is a lot. We'll proceed with Wi-Fi and note where Bluetooth could fit in.)

- **Threading/Async on PC:** Use Python `threading` or `asyncio` to handle multiple device connections. For example, spawn a thread for each connected phone to listen for messages:

```
def handle_client(sock, addr, phone_id):
    with sock:
        buffer = b""
        while True:
            data = sock.recv(1024)
            if not data:
                break  # client disconnected
            buffer += data
            if b"\n" in buffer:  # assume newline-terminated messages
                line, buffer = buffer.split(b"\n", 1)
                message = line.decode('utf-8')
                process_phone_message(phone_id, message)
        print(f"Phone {phone_id} disconnected")
```

  The `process_phone_message` function would update the GUI accordingly (use signals or QTimer to safely update UI from thread).

- **Latency and Sync:** The network approach introduces some latency (tens of milliseconds). But since the phone records data locally, a slight delay in command is okay. We will ensure the PC sends the start command to all phones nearly simultaneously. Each phone will note its start time. We will use NTP/RTCP or an initial time sync to align clocks – see Section 4.

At this point, implement and test the networking on a small scale: - Hardcode the PC's IP in the phone app for now (or allow input). Launch the PC app (server) and then the phone app. Confirm that the phone says "connected" on PC side. - Click a "Start" on PC targeting the phone – verify the phone begins recording (check the phone's UI or logs). Then "Stop" – verify it stops. - Try with two phones if available: ensure both can connect and start/stop together when commanded.

This achieves **centralized remote control** of devices [16]. The PC operator can start/stop all recordings in sync with one button, rather than manual on each phone.

3.5 Real-Time Data Streaming (Preview on PC)

As an enhancement, incorporate at least a basic preview of what each phone is recording: - **Video Preview:** We can repurpose the phone's camera preview frames. One idea is to use the Android

`ImageReader` in parallel with recording: e.g., for every Nth frame, compress to JPEG and send to PC. This could be done in the camera capture callback or using the `PreviewView` output (if accessible). Another method: if using Camera2, add a `ImageReader` on a secondary target for the capture session. But that gets complex. A simpler hack: periodically take a snapshot of the `TextureView` (which can produce a Bitmap via `textureView.bitmap`) and send that. For example, every second or two, do:

```
val bmp = previewView.bitmap
if(bmp != null) {
    // compress and send
    val stream = ByteArrayOutputStream()
    bmp.compress(Bitmap.CompressFormat.JPEG, 50, stream)
    val jpegBytes = stream.toByteArray()
    // send length + bytes or some framing protocol
    outputStream.write(("FRAME "+jpegBytes.size+"\n").toByteArray())
    outputStream.write(jpegBytes)
}
```

On PC side, interpret a message "FRAME X" to know the next X bytes are an image, then read that many bytes and display the image in the phone's panel (convert bytes to QImage).

This is a rudimentary custom protocol, and you must be careful with socket reading to not mix with text commands. You might use separate sockets: one for control (text) and one for streaming (binary). To simplify, you could decide not to intermix binary data on the command channel.

- **GSR Preview:** For each GSR sample or perhaps averaged over 1s, have the phone send a line `"GSR <value>\n"`. The PC thread will parse it and update a label or even plot a live graph using PyQtGraph. The GSR data is low-bandwidth, so sending at, say, 10 Hz is fine. This gives the researcher feedback that physiological data is coming in (e.g., "GSR: 420 µS").

- **Thermal Preview:** Similar to video, you can occasionally send a thermal image. Perhaps even less frequently due to size. If thermal is 256x192, a JPEG of that is small (<50KB), so it's feasible. The phone can compress the thermal frame (if it has a bitmap of it) and send it. The PC can display it in the phone's panel.

Keep the frequency of frame updates modest to avoid network flooding (especially with multiple phones). E.g., 1 FPS preview per phone. This is enough to verify camera framing and general status [17] [18].

Test the previews: - When the phone is connected and capturing, see if an image appears on the PC GUI for that phone. There might be a slight delay. - Check that GSR values on PC match what's on phone (if you have both displaying). - Network usage should remain reasonable (monitor if possible, e.g., a few hundred KB/s).

4. Time Synchronization and Data Alignment

Synchronizing the timestamps of data from multiple devices is critical. Our goal is that all data streams can be merged post-hoc with minimal offset. We address sync on two levels: **between PC and phones**, and **between different sensors/modalities** on each device.

a. NTP Time Sync: The simplest baseline is to ensure all devices' clocks are in sync via NTP (Network Time Protocol). If the phones and PC have internet, they likely already sync to NTP servers (usually within ±10 ms). You can also run an NTP server on the PC and have phones sync to it (on rooted devices or via special apps) – but this might be overkill. For our purposes, ensure date/time are correct on all devices before recording. This gives a common reference (UTC or epoch time).

b. Lab Streaming Layer (Optional): If using LSL for streaming, LSL automatically handles time synchronization by estimating clock offsets between endpoints [19]. For example, each phone's LSL outlet would provide its local timestamps, and the PC's LSL inlet can query the offset to convert them to the PC's time base. If you went the LSL route, leverage that: each sample (frame, GSR) gets an accurate global timestamp [13]. In analysis, you can simply align on those. (LSL also allows a separate **clock sync test** – not needed if functioning.)

c. Manual Timestamp Alignment: Even with NTP, there could be small offsets (10s of ms). To improve: - Use a **sync event**: For instance, when the PC operator clicks "Start", the PC can send a timestamp with the command. E.g., `"CMD START 1698812345.123"` (a UNIX epoch with fractional seconds). The phone, upon receiving, can note the difference between its own clock and the PC timestamp:

```
val pcTime = receivedTimestamp
val phoneTime = System.currentTimeMillis()/1000.0  // get seconds.sss
timeOffset = phoneTime - pcTime
```

This `timeOffset` (which could be positive or negative) tells how much ahead/behind the phone's clock is relative to PC. Save it. Then for each recorded data point on the phone, you could store a PC-aligned timestamp = phone_timestamp - timeOffset. (If you want to be precise, use `SystemClock.elapsedRealtimeNanos()` on Android for a steady clock and synchronize that to PC's reference time.)

- Alternatively, perform a **round-trip ping** from PC to phone to estimate offset. E.g., PC sends "SYNC?" with its time, phone replies immediately with its current time, PC receives and compares. A simple average can give offset. This is essentially how NTP works. If you implement this at start of session, you can get maybe ±5–10 ms accuracy on a good Wi-Fi network.

- **Synchronization across devices:** Since the PC sends the start to all phones almost simultaneously, and assuming network latency to each is within a few ms on the same network, the start events are effectively synced to that precision. Each phone might actually start recording a fraction of a second after receiving the command (processing delay). To measure any drift or offset, you could do a quick test: e.g., start all devices and have a *physical synchronization event* like a clap or a flash of light in view of all cameras and maybe touching the GSR sensor. Later, in the data, align those events. In practice, our software approach (with NTP and simultaneous commands) should achieve on the order of tens of milliseconds sync, which is often sufficient for GSR vs video alignment (the reference system aimed for sub-millisecond sync with a more complex setup [20] [21], but our simpler approach may be within ~50 ms which is acceptable for many applications).

- **Timestamp Logging:** Ensure that in each recorded file, the timestamps are either absolute or relative to a known start. For example, in the GSR CSV, it might be easier to store relative time since session start (0 at start, then 0.0078, 0.0156, … seconds). In the video, the first frame timestamp could be considered 0. As long as you know how to line them up, it's fine. A common

strategy: have the PC log a master record of the session start time (PC clock) and send that to all devices. Each device also notes its local start time. Later, you use those to align.

Summarizing, to implement sync in code: - When PC operator hits "Start All", capture `master_start_time = time.time()` (PC epoch time). Send `CMD START <master_time>` to each phone. - On phone, upon receiving start, record `sessionStartEpoch = <master_time>` and also `sessionStartLocal = System.currentTimeMillis()/1000.0`. The difference is offset. - When writing data, each phone can either: - Convert each timestamp to epoch (e.g., if a GSR sample taken at local_time, its epoch ~ `local_time - sessionStartLocal + sessionStartEpoch`). - Or store relative time and note the offset somewhere.

- On PC, also note when it started recording its own data (which ideally is immediately at master_start_time). If PC's webcam thread started slightly later, note that delay.

With this procedure, after the session, all data can be converted to the same timeline (the master PC clock timeline).

5. Multi-Device Coordination and Remote Control

The PC application now serves as a **central control hub** for potentially multiple Android devices. Here are additional considerations for managing multiple nodes:

- **Device Management:** If two phones are used, ensure you label them (e.g., in the UI, "Phone A" vs "Phone B"). This could be via config (user enters which IP corresponds to which participant or body location). Our PC GUI can have a tab or panel for each phone. The server can differentiate clients either by the order they connected or an ID string they send. Implement a simple registration: when a phone connects, have it send e.g. `"ID PHONE_A\n"`. Map that socket to a name.

- **Simultaneous Start/Stop:** Provide a **global Start/Stop** button that affects all connected phones and the PC's own recording at once. This will invoke the logic from Section 3.4 to send commands and start local capture. The operations should be as concurrent as possible. (In Python, sending to each socket sequentially is usually fast enough – a few milliseconds difference. If you want, spawn threads to send simultaneously, but it's likely unnecessary.)

- **Status Monitoring:** The PC should listen for status updates from phones. If a phone unexpectedly disconnects (network drop or app crash), handle that: the thread will end, so update UI to show that phone is offline. If a phone sends periodic heartbeats or status (you can implement a `"ALIVE"` message every few seconds), use that to update a last-seen timestamp in UI. The PC can also poll phone status via commands if needed (e.g., send `"CMD STATUS"` to query battery, etc., if you extend protocol).

- **Bluetooth Phone Control:** Although we focus on Wi-Fi, if needed, you could implement a fallback: For instance, if Wi-Fi is not available outdoors, you could tether via a Bluetooth connection. The code structure would be similar but using RFCOMM sockets. PyBluez on PC would do `sock = BluetoothSocket(RFCOMM); sock.bind((HOST_ADDR, CHANNEL))` etc., and on Android use `BluetoothAdapter.getRemoteDevice(mac).createRfcommSocketToServiceRecord(UUID)`. This is advanced and platform-specific, so ensure Wi-Fi first. (The design supports both per requirements [14] [22], but implementing both concurrently might be a future improvement.)

At this point, we have a functional system where the PC can control multiple Android capture nodes and gather data streams.

6. Recording, Saving, and Data Management

When the user triggers a recording session (via PC or manually), all data is recorded locally on each device. After stopping, you will have: - On each phone: video file(s), GSR data file, etc. - On PC: a video file (for webcam) and perhaps any PC sensor data.

You should consider how to **aggregate or store these session files**: - A straightforward way: after stopping, have the phones automatically send their data files to the PC over the network. For example, the phone could initiate a file transfer (via the same socket or an HTTP POST to a server on PC). However, large video files could be slow to transfer wirelessly. If immediate analysis isn't needed, you might manually copy them later (USB or SD card). - Alternatively, ensure the filenames include device identifiers and timestamps so that if you dump all files in one folder later, you can distinguish them. For instance: `session123_phoneA_video.mp4`, `session123_phoneA_gsr.csv`, `session123_phoneB_video.mp4`, etc., and similarly `session123_pc_webcam.mp4`. - You can also create a **session manifest** (JSON or CSV) that lists all files and their start times, offsets, etc., to facilitate data merging. The PC could generate this since it knows the sync info. For example:

```
{
  "session_id": "2025-07-01_103000",
  "phones": {
     "PHONE_A": {
         "video_file": "session123_phoneA.mp4",
         "thermal_file": "session123_phoneA_thermal.mp4",
         "gsr_file": "session123_phoneA_gsr.csv",
         "start_epoch": 1698812580.123,
         "offset": 0.015  // phone clock offset to master
     },
     "PHONE_B": { ... }
  },
  "pc": {
     "webcam_file": "session123_pc_webcam.mp4",
     "start_epoch": 1698812580.123
  }
}
```

This manifest (which you can save in `shared/` or output directory) will help in analysis or later automation (like a script to merge data).

- **Local vs Central Recording:** In future, one might stream all raw data to the PC and record centrally. Our design currently **records locally on each node** to ensure full quality and reliability (network hiccups won't stop the local recording). The PC is just commanding and previewing. This is generally safer for high bandwidth data (video). If you have ample network capacity and want to centralize recording, you could send the video streams to PC and write them there, but that requires robust streaming code (or using something like RTSP/RTMP servers).

- **LSL Note:** If you did implement LSL streaming on phones, the PC could actually act as the recorder by subscribing to those streams and using LabRecorder to save them. In that case, the data would be recorded on PC with all streams in one XDF file, automatically time-synced [1] [23]. The trade-off is that streaming raw video via LSL can be extremely bandwidth-heavy. So likely, keeping as files is wiser.

7. Testing and Validation

Now that the system is built, thorough testing is important:

7.1 Component Testing

- **Android unit tests:** Test each sensor individually on the phone:
- Open the Android app on the device with the Shimmer off to see how it handles no sensor (it should not crash; maybe it shows disconnected status).
- Turn on the Shimmer, ensure it connects and GSR values update. Verify sampling rate roughly (e.g., log timestamps to see ~128 Hz or whatever set).
- Test camera preview in various lighting. Try record video for a short time and play it back (check resolution, audio).
- Test thermal camera connection: plug/unplug while app is running to see if it handles it (perhaps add a reinitialize if disconnected).

- Check that starting and stopping recording multiple times works (no resource leaks like camera not reopening).

- **PC component tests:**

- Run the PC app without any phones: it should still show the webcam and not freeze.
- If possible, simulate a phone connection: you can use `telnet` or netcat to connect to the PC server port and send a pretend `"HELLO PHONE_X"` to see if the UI adds a device entry. This can test the server thread handling.
- If you have two webcams or a dummy video source, try changing the code to see if multiple video streams overlay (not typical, but just ensure UI is flexible).
- Test the PC's Start/Stop when no phone is connected – it should maybe just record PC webcam.

7.2 Integration Testing

- **Single-Device End-to-End:** Use one phone and the PC:
- Start the PC app, connect the phone (over Wi-Fi). Ensure the PC indicates the phone is connected.
- In the PC UI, click "Start Capture". Verify simultaneously: the phone starts recording (look at the phone's screen or LED if any) **and** the PC starts its recording (maybe show a timer or red dot on PC GUI).
- Perform some actions for sync: e.g., clap in view of phone camera and PC camera at the same time, or use a flash. Also maybe tap the GSR electrodes at that moment to create a spike.
- Click "Stop" on PC after, say, 10 seconds.
- Check results: The phone should have saved files, PC saved its file. The PC UI might show a message like "Files saved on device" if you implemented feedback.
- Transfer the phone's files to PC (manually or if you implemented auto-transfer).
- Analyze timestamps: Verify that the clap is at the same global time in both videos (play them side by side, or check metadata if available). Check that the GSR data around that time shows a

disturbance corresponding to the clap (if it caused any). They might not exactly, but you can at least see if GSR timeline aligns roughly with video timeline using the recorded start times.

- If misalignment is more than, say, 100 ms, investigate the offset and adjust the sync approach.

- **Multi-Device Sync Test:** If you have two phones, do the above with both:

- Connect both phones to PC (over Wi-Fi or one Wi-Fi, one Bluetooth to test both channels).
- Start recording on both via PC.
- Do an event like clap visible to both phone cameras (get them in one scene if possible) and PC camera.
- Stop, gather files, and compare.

- Also, verify that PC successfully started both and stopped both. Check if one started late or any error was reported.

- **Stress Testing:** Try longer recordings (several minutes) to ensure no memory leaks or buffer issues. GSR and video generate a lot of data; ensure the phone can handle it (Shimmer at 128Hz for minutes is fine, video is heavier on storage). Check that file sizes are reasonable and that the app doesn't crash on large files.

- If possible, fill a short survey during recording (just as activity) to simulate real usage and see if any interference (not likely relevant unless the app goes background – ensure the Android app is set to keep screen on and perhaps use a Foreground service if you want to guard against Android pausing it).

7.3 Validation of Synchronization

To be confident in sync: - Use **analysis scripts** (could be a part of `shared/` or separate Jupyter notebooks) to load the recorded data and overlay them. For instance, find the frame in phone video where clap happens and note its timestamp (if you instrumented the video recording to log start time and you know frame index, etc.), and do the same for PC video. Or simply manually align by looking at videos in an editor. - If using LSL, you can use LabRecorder to verify all streams; but assuming not, manual analysis is fine.

- Check **GSR vs video** sync: GSR changes with a slight delay to stimuli (physiologically, GSR lags by 1-3 seconds for a stimulus). But if you have an obvious event (like touching the sensor or disconnecting it briefly), see that the timestamp aligns with when that happened in video.

If you find discrepancies, you might need to refine the offset calculation: For example, if phone video appears consistently 0.5s later than PC video, perhaps the phone started half a second late. You could incorporate a fixed offset or improve command dispatch (maybe measure the actual time gap and compensate).

At this stage, the **system should meet the requirements**: synchronized multi-modal capture, remote control, previews, and proper data logging. The PC GUI provides a central hub for up to 2 phones [2] [24], and all data streams are timestamped for synchronization [25].

8. Including SDKs and External Libraries in the Monorepo

We have mentioned external SDKs (Topdon, Shimmer, LSL). It's important to properly include these in your project and repository:

- **Topdon SDK:** If Topdon provided you with an SDK archive, it might contain an `.aar` for Android and maybe some documentation. Add the `.aar` file to `android-app/app/libs/` and version-control it (unless its license forbids; check that). Note the version or source of this SDK in your docs. If the SDK came with sample code, keep that in `docs/` for reference.

- **Shimmer SDK:** We integrated it via Gradle dependency (JFrog). You don't need to store the library in the repo since Gradle will fetch it. But do document the version in your `docs/` or README (e.g., "Using Shimmer Android API v3.0.74beta"). For the PC side, if you end up using any Shimmer libraries (unlikely, as we used generic serial), list those in requirements.

- **LSL:** If you compiled liblsl for Android, that might produce a `.so` file and a Java wrapper. You could include those in `android-app/app/src/main/jniLibs/` (under armeabi-v7a/arm64-v8a folders). Or include as an AAR if provided. On PC, pylsl is just a pip dependency, so that's handled. If not using LSL yet, you can omit it for now.

- **Git LFS:** If any SDK files (like Topdon .aar or FLIR SDK if used) are large (>100 MB), consider using Git LFS to track them. But ideally, keep things lightweight. Topdon's AAR should be a few MB.

- **Documentation:** In the `docs/` directory of the repo, include:

- A **Setup Guide** (which could be derived from this content) explaining how to install and run the system.
- A **User Guide** for an operator (how to use the PC GUI, how to prepare phones, etc.).
- Any vendor manuals or relevant research papers (if this is for academic purposes).

- A **configuration file** example (the PC app might use a JSON config for IP addresses of phones, etc., which you load on startup).

- **README:** Your repo README should highlight the project architecture and link to the docs for detailed instructions. For inspiration, note how the existing project described integration of LSL, dual phone control, etc., in its README [1] [26] – you can do similar for your project.

9. Running the Complete System

Finally, you are ready to run full sessions with the system:

Step-by-Step Operation: 1. **Launch PC Controller:** Run `pc-app/main.py` (IntelliJ run config or `python main.py` in terminal). The GUI window appears. Verify the local webcam view is working and the interface shows "Waiting for devices" or similar. 2. **Launch Android Apps:** Start the `android-app` on each phone. Ideally, have an initial screen where you configure the PC server IP (unless you coded it static). Enter the IP and tap "Connect" (or the app auto-connects on start). The phone should indicate "Connected to PC". On the PC GUI, you should see each phone listed (e.g., a tab or panel becomes active, showing the device name). 3. **Prepare for Capture:** On each phone, attach the sensors: - Plug in the thermal camera (if not already). The phone app might automatically start the thermal preview (or perhaps waits until recording starts to initialize it to save resources). - Turn on and connect the Shimmer

GSR sensor. The phone app might auto-connect if it has the MAC saved, or provide a UI to select the device. Ensure GSR data is streaming (maybe an indicator on phone, or wait for PC to show GSR values if you wired that through). - Make sure the phone's camera is seeing the subject properly. You should see preview on phone; if you set up preview streaming, also glance at the PC's small preview for each phone. - Ensure participants are ready (if you have participants wearing sensors). 4. **Start Recording:** In the PC app, click **Start** (the global start). This will lock the UI (optionally, you might disable the button or change it to "Recording..." during capture). It sends commands out: - Phones receive start, begin recording all modalities and perhaps send back confirmation. - PC starts its webcam recording and any local data logging. - The PC GUI might show a timer or status "Recording in progress..." for each device [16]. - Monitor the live feeds: you should see video frames updating (at the reduced preview rate) and GSR values changing. This is a good time to observe if any device is lagging. 5. **During Recording:** Conduct your experiment or data collection. All the while, keep an eye on: - GSR plots (if implemented) for any flatlines (could indicate disconnection). - Video preview to ensure framing is correct. - The PC sync monitor (you might implement a simple "drift" calculator that compares device clocks occasionally). In advanced setups, you could have a dedicated sync visualizer [27], but initially just ensure everything *appears* steady. 6. **Stop Recording:** Click **Stop** on PC. This sends stop commands: - Phones stop their recordings and close files. - PC stops its recording. - The UI should reflect that recording stopped (timer stops, maybe a "Ready" status returns). You can re-enable the Start button for a new session if needed. 7. **Data Offloading:** If you implemented automatic file transfer, the PC might now request each phone to send its files. This could take time for large videos. Alternatively, you just inform the user to collect the files manually. Ensure that the files are saved and not corrupted. On Android, a common issue is file not finalizing if recording wasn't stopped properly – our code stops properly, so it should be fine. 8. **Post-Session:** If a manifest or log was generated, review it. For example, a console log on PC might have printed "Phone A started at 10:00:00.123, Phone B at 10:00:00.130, offset ~7ms". This is good info to note. - Disconnect devices if needed. The phone apps can either stay running waiting for next session or auto-quit after stop (up to you).

1. **Verification:** Load a sample of the data to quickly check alignment. You might open the videos side by side, or use a Python script with pylsl or OpenCV to read frames and GSR CSV to ensure the data makes sense. This is more for your confidence that the system works reliably every time.

Throughout the process, maintain **documentation** for the user. For instance, in `docs/user_guide.md`, describe how to perform the above steps, including screenshots of the PC GUI and Android app. This will help others (or your future self) use the system correctly.

10. Conclusion and Next Steps

Congratulations – you have constructed a comprehensive multi-device, multi-modal recording system from scratch! You now have: - An Android app capturing synchronized RGB video, thermal imagery, audio, and GSR, with real-time preview and data logging. - A PC application that centrally controls multiple Android devices, receives live previews, and records its own data, all while maintaining synchronization across streams. - A monolithic repository in IntelliJ IDEA that allows you to develop and debug both the mobile and desktop components in one place. - Integrated use of vendor SDKs (Topdon, Shimmer) and standard libraries (CameraX, PySide6, OpenCV) to achieve the system functionality. - A strategy for time synchronization and a verified level of accuracy suitable for research purposes (the design aimed for high precision with unified timestamps across modalities [25]).

Next Steps and Improvements: - You may consider refactoring parts of the code for better maintainability. For instance, on Android, integrating a dependency injection framework like Hilt and separating concerns (ViewModel, repository for hardware as seen in our refactoring plan [28] [29]) can

make the app more robust. On PC, you might modularize further or add unit tests. - Enhance the GUI/UX: e.g., add indicators for battery level of devices, allow configuration of sampling rates from the PC, or display synchronization error metrics in real time (perhaps using an LSL-based time sync test). - Implement data **export/analysis tools** in the `shared/` folder. For example, a script to merge phone and PC data into a single CSV or HDF5 file for analysis, or even to feed into a machine learning pipeline. - **Error handling and reconnection:** improve how the system deals with dropped connections. E.g., if a phone app crashes mid-session, the PC could detect loss and possibly even re-initiate recording on that node. - **Cross-platform considerations:** While our target was Windows for PC, much of the PC code (PySide6, OpenCV, etc.) can run on Linux or macOS as well. If needed, test on those platforms. Android app of course runs on Android devices. This means the solution is quite flexible. - **Scaling up:** The design supports 2 phones as specified [2]. If you wanted more, you could try, but bandwidth might be a limiting factor. Two is usually enough for stereoscopic setups or covering two subjects. - **Documentation and open-source:** If appropriate, consider publishing the documentation or code (respecting SDK license restrictions). Others in the research community might benefit, and you could get feedback or contributions.

By following this guide, a developer starting with nothing should be able to gradually build the entire system and have it functioning. Each step builds on the previous, and at each milestone you have a working subset that you can verify. Good luck with your synchronized GSR & dual-video recordings, and happy data collecting!

Sources:

- Buccan's GSR-RGBT project README (features and architecture) [1] [26] [30] – provided inspiration for multi-device support, LSL integration, and unified timestamp logging.
- FYP system usage notes on dual phone control and time sync [2] [24].
- Shimmer Android API documentation – for proper integration of the Shimmer GSR sensor [5].
- Les Wright's TC001 research – understanding thermal camera data handling (256×192 sensor specifics) [9].

---

[1] [2] [3] [4] [11] [12] [13] [14] [15] [16] [17] [18] [20] [21] [22] [23] [24] [25] [26] [27] [30] README.md
https://github.com/buccancs/fyp-gsr-windows/blob/2d41c241dfeccbb9f5dc0b582255f8b67c8e0ec6/README.md

[5] [6] GitHub - ShimmerEngineering/ShimmerAndroidAPI
https://github.com/ShimmerEngineering/ShimmerAndroidAPI

[7] labstreaminglayer/liblsl-Android: Android bindings for liblsl - GitHub
https://github.com/labstreaminglayer/liblsl-Android

[8] mvidaldp/liblsl-android-builder: LSL (LabStreamingLayer ... - GitHub
https://github.com/mvidaldp/liblsl-android-builder

[9] tc001v4.2.py
https://github.com/leswright1977/PyThermalCamera/blob/b19821a1a25e081594c666022a3d64e2b9cf41cc/src/tc001v4.2.py

[10] shimmer_usage_example.kt
https://github.com/buccancs/gsr-android-dual-video-stream/blob/91ed504b1bae3336b870e9452bc3ea2bae9b5c42/shimmer_usage_example.kt

[19] Time Synchronization — Labstreaminglayer 1.13 documentation
https://labstreaminglayer.readthedocs.io/info/time_synchronization.html

28 29 10_POINT_REFACTORING_PLAN.txt

https://github.com/buccancs/gsr-android-dual-video-stream/blob/91ed504b1bae3336b870e9452bc3ea2bae9b5c42/docs/10_POINT_REFACTORING_PLAN.txt