ChatGPT

Technical Audit and Upgrade Plan for the FYP-GSR-Android Application

1. Full Technical Audit of FYP-GSR-Android

Overview: The FYP-GSR-Android application (also known as the GSR-RGBT Android Data Capture app) is a multimodal mobile app designed for synchronized recording of **thermal infrared video**, **RGB video**, **audio**, and **Galvanic Skin Response (GSR)** sensor data [1]. It forms the Android side of a research toolkit intended to collect physiological data in sync, with a companion PC application (FYP-GSR-Windows) for data reception and analysis. Below is a detailed audit of its code structure, architecture, hardware integration, communication protocols, and current limitations.

1.1 Code Structure and Architecture

The application is written in **Kotlin** with a modular architecture. The codebase is organized into clear packages by functionality, following a Model-View-Controller-esque separation. Key modules and their roles are:

- **Capture Modules (`capture/`):** Each sensor modality has its own capture module implementing a common `CaptureModule` interface [2]. For example:
- `ThermalCaptureModule` – handles the Topdon TC001 thermal camera integration [3].
- `RGBCaptureModule` – manages the phone's Camera2 API for RGB video [4].
- `GSRCaptureModule` – intended for the Shimmer GSR+ sensor (currently using a simulation subclass) [5].

- `AudioCaptureModule` – records microphone audio input (with fallback to simulation if mic unavailable).

- **Synchronization Manager (`sync/`):** A `SynchronizationManager` provides a unified time base for all data streams. It timestamps each sensor event using Android's monotonic clock (`elapsedRealtimeNanos`) to achieve frame-level synchronization between modalities [6]. This manager also queues events and distributes them to logging or networking systems.

- **Data Model (`data/`):** Data from each modality is encapsulated in **timestamped event classes** (subclasses of `DataEvent`) such as `ThermalFrameEvent`, `RGBFrameEvent`, `GSRDataEvent`, `AudioDataEvent`, etc [7] [8]. These events carry both the sensor data (or metadata) and a timestamp in nanoseconds. They can be serialized (using Kotlinx serialization) to JSON for network transmission [9] [10] or written to files.

- **Recording Controller (`controller/`):** The `RecordingController` orchestrates the lifecycle of all capture modules and coordinates start/stop across them [11] [12]. It uses a `CountDownLatch` to ensure all modules begin capturing in unison for synchronized start [13]. It also manages a **LocalDataRecorder** (for saving data to storage) and interacts with the Networking layer for multi-device sync.

- **Networking (`networking/`):** There are networking service classes to enable multi-device recording and data streaming:

- `NetworkingManager` – a high-level coordinator that selects between **Bluetooth**, **Wi-Fi Direct**, or **Lab Streaming Layer (LSL)** modes and manages connection state [14] [15] .
- `BluetoothNetworkingService` – (BLE or Bluetooth Classic) service for phone-to-phone data sync and coordination.
- `WiFiNetworkingService` – uses Wi-Fi Direct (P2P) to allow higher bandwidth data exchange between devices (one device acting as group owner/server) [16] [17] .
- `LSLStreamingService` – integrates with Lab Streaming Layer, creating LSL outlets for each data stream to broadcast data to a PC running an LSL client [18] [19] .

- All networking services implement a common `NetworkingService` interface which defines methods for starting as master or slave, discovering peers, sending data events, sync commands, etc. [20] [21] .

- **UI Layer (`ui/`):** The main user interface (`MainActivity`) provides a preview and control panel. Key UI elements include:

- A **TextureView** for camera preview (switchable between RGB and thermal views).
- Status text overlays for each module (showing e.g. "Connected", "Capturing", or error states).
- A primary **Start/Stop button** to control recording.
- A toggle to switch camera view mode (RGB, Thermal, or hybrid overlay mode).
- Menu actions for opening a **Settings screen** and a **Device Pairing dialog** (to scan for BLE sensors).

The **app architecture** can be visualized as multiple parallel capture threads feeding into a synchronization hub, which then dispatches data to storage and/or network. This is consistent with the design documentation, which illustrates thermal, RGB, and GSR modules all providing timestamped data to the SynchronizationManager, and from there either to local logs or out over the network [22] [23] .

Each capture module runs in its own coroutine or thread with appropriate priority: - The Thermal and RGB capture threads are set to **high priority** to minimize frame drops (as stated in documentation and reflected in usage of `Dispatchers.Default` with thread priority hints) [24] . - The GSR module uses BLE callbacks or a simulation loop, and the Audio module uses an `AudioRecord` read loop. All modules push data to the SyncManager, which tags each event with a unified timestamp.

Module Lifecycle: The RecordingController handles initializing each module (via an async `init()`), then starting all nearly simultaneously. It monitors their status flows to update the UI. On stopping, it signals each module to stop and then waits for termination. It also ensures resources are released (e.g., camera released, file writers closed, threads joined) to avoid leaks.

Error Handling: The code includes robust error handling placeholders. For example, if a module fails to init or loses connection, the UI status is updated and often a fallback is used. The Thermal module will enable **simulation mode** automatically if no USB camera is detected [25] . The GSR module currently *is* a simulation by default (until real integration is provided), ensuring the app can run without the physical sensor. The architecture is meant to be resilient: inoperative hardware should not crash the app thanks to try-catch blocks and status flags.

Simulation Mode: Simulation code is present for thermal, GSR, and even audio. In the absence of hardware, the app generates synthetic but "realistic" data: - Simulated thermal frames: an algorithm produces a moving heat pattern as a ByteArray representing thermal pixel values [26] [27]. - Simulated GSR values: a sinusoidal variation plus noise to mimic physiological GSR trends (1–20 μS range), and a pseudo-PPG waveform for heart rate [28] [29]. - Simulated audio: if microphone permission is missing, the audio module sets `simulationMode` and doesn't actually record (could generate silent or dummy audio, though current implementation just reports no mic).

These simulations allow testing the synchronization and data pipeline without all hardware present, but they are placeholders and not meant for production data collection.

1.2 Hardware Integration and Dependencies

The application integrates with two main external devices: **Shimmer3 GSR+ sensor** and **Topdon TC001 thermal camera**, as well as the phone's built-in sensors (camera and microphone). Key dependencies and their status:

- **Shimmer3 GSR+ (BLE GSR & PPG sensor):** This is a wearable sensor providing Galvanic Skin Response and Photoplethysmography (PPG) data via Bluetooth Low Energy. The app intends to use Shimmer's official BLE API or a custom parser to stream GSR data at 128 Hz [30]. **Current status:** *Not fully implemented.* The codebase includes a `GSRCaptureModuleSimulation` but no concrete class for real Shimmer data capture (likely planned but incomplete). There is a **DevicePairingDialog** that can scan for BLE devices (filtering by "Shimmer" device type) and allow the user to select one [31] [32]. However, after selection, the app does not yet initiate a connection to the device – the code contains a **TODO** note to "connect to selected device via GSR module" where integration logic would go [33]. This indicates that while the UI for choosing a sensor exists, the actual BLE data streaming (using Shimmer's provided services/characteristics) is not done. In effect, the app still uses simulated GSR values for now, meaning **GSR and PPG data are placeholders** until the Shimmer integration is completed.

- **Topdon TC001 Thermal Camera (USB-C):** This is an infrared thermal camera attachment that streams thermal images (resolution 256x192, up to ~25 Hz). Integration is done via the **Topdon Thermal SDK** (an `.aar` library provided by the manufacturer). The repository includes a stub `ThermalSDK` class [34], but in practice the real SDK .aar (and a supporting lib `libusbirsdk.aar`) must be included in `app/libs/` for full functionality [35]. The `ThermalCaptureModule` uses the SDK to initialize the camera, register a frame callback, and start the thermal stream [36] [37]. **Current status:** Partial integration. The code attempts to find a connected USB device matching the camera (by USB Vendor/Product ID possibly), and if found, calls `initializeTopdonSDK()` (which in turn likely loads the native library) [38] [39]. If the camera isn't found or SDK init fails, it falls back to simulation mode [40]. The frame callback in `runRealCapture()` receives each frame's byte data and spawns a coroutine to process it (timestamp and enqueue to SyncManager) off the SDK thread [41] [42]. **Limitations:** The code for rendering the thermal preview (`renderThermalFrameToSurface`) is not fully implemented (it contains a stub comment) [43], so the live thermal view might currently be blank or not properly converted to a visible image. Also, the Topdon SDK initialization and usage might require the device to grant USB permission; the app does prompt the user when connecting (by default Android USB permission dialog). This integration likely requires testing on-device; it's assumed to work given correct SDK, but any SDK version mismatch or phone USB host issues could cause it to quietly revert to simulation.

- **Phone RGB Camera:** Uses Android's **Camera2 API**. The RGBCaptureModule sets up a camera session for the rear camera, configured for 1920x1080 at 30fps by default (with potential to adjust resolution via settings) [44]. It employs manual exposure and focus settings to avoid frame timing jitter [24]. Frames are captured in a background thread. The code likely captures in a YUV format (as indicated by `format: "YUV_420_888"` in events) [45]. Those frames are not saved as images in this app (the DataWriter logs only metadata like width, height, and perhaps file path if implemented; see **Data Logging** below). The main purpose is providing a preview and synchronized timestamp for each frame. The RGB integration is standard and should be stable on devices supporting Camera2. **Android version compatibility:** The app targets Android 8.0+ (Oreo, API 26) [46], and uses Camera2 (which is supported on almost all modern devices). There is manual permission handling for CAMERA in the app.

- **Phone Microphone (Audio):** The AudioCaptureModule uses **AudioRecord** to capture PCM audio (default configuration 16-bit mono at 16 kHz, though it supports up to 44.1 kHz stereo) [47] [48]. It reads audio in small chunks (100 ms segments by default) and timestamps each segment to align with other sensor data [49] [50]. If microphone permission is denied or not present, it goes into simulation mode (i.e., doesn't record actual audio) [51]. **Current status:** Audio recording is implemented and considered *complete* in documentation [52]. However, a potential issue found is that the app's permission request list does **not include** `RECORD_AUDIO` (microphone permission), meaning unless the user manually grants it via settings, the app will always run audio in simulation mode by design (since `checkMicrophonePermission()` will fail). This appears to be an oversight: the `requiredPermissions` array in MainActivity includes camera, Bluetooth, location, storage, but no audio [53]. For full audio functionality, adding microphone permission is necessary. Once granted, the audio module should produce WAV data saved to storage (the README mentions WAV file export, though in practice the implementation logs PCM segments to CSV and the actual WAV file path isn't explicitly handled in the provided code).

Local Data Logging: The app can locally store captured data for later analysis. The `CsvDataWriter` creates a session folder per recording and writes multiple CSV files: `thermal_frames.csv`, `rgb_frames.csv`, `gsr.csv`, `audio.csv`, and `sync_markers.csv` [54] [55]. Each contains timestamped entries of that data type (with metadata like frame indices or data sizes, but not the full raw frame data to keep file sizes manageable) [56] [57]. For example, each thermal frame entry logs timestamp, frame index, dimensions, data size, and some computed stats (min/max/avg temperature from that frame) [58]. RGB frames log timestamp, frame #, resolution, pixel format, and byte size [59]. GSR entries log timestamp, sample index, GSR value (in μS), the sensor's own timestamp if available, and PPG value if present [60]. Audio logs timestamp, segment index, sample rate, channel count, segment duration, an audio level (computed RMS) and data size [61]. This design prioritizes capturing synchronized *timestamps* and summary info rather than storing raw signal waveforms in the CSV. The actual raw data (like the audio PCM or image bytes) are not written to CSV; however, audio could be reconstructed from PCM segments if needed, and frames are either discarded after logging or only kept in memory briefly. This approach prevents huge file sizes (the README estimates ~1-2 GB/hour including video [62], likely assuming compressed video recording; CSV metadata alone will be much smaller).

Architecture Assessment: Overall, the code structure is well-thought-out and modular. Each sensor's integration is encapsulated, which aids maintainability and future extension (e.g., adding another sensor by implementing `CaptureModule`). The use of a centralized SynchronizationManager is a strength, as it abstracts the timing and makes it easy to add new sinks (like networking or local logging) without changing capture code. The inclusion of simulation paths shows foresight for development and testing. However, some parts of the architecture remain unfinished or only partially tested, which we will detail as limitations next.

1.3 Communication Protocols with FYP-GSR-Windows

Aside from local recording, FYP-GSR-Android can stream data to a PC (the FYP-GSR-Windows application) for real-time monitoring and saving on a computer. This is critical for use cases like live biofeedback or when a more powerful PC is used as a central logger for multiple mobile devices.

There are two primary mechanisms for communication implemented:

- **Socket Networking (TCP/UDP):** In "Network" mode, the Android app can act as a **client or server** to send data events over a network socket. The PC application runs an `AndroidStreamReceiver` server listening on a TCP port (default 8080) [63] [64]. The protocol is as follows:
- Android serializes each DataEvent to a JSON string (with fields like `type`, `timestamp`, and `data`) [65] and sends it over the socket. Over TCP, a message length header (4-byte int) precedes each JSON message [66]; over UDP, messages are sent as standalone datagrams.
- The JSON includes a `type` field that identifies the data (e.g., `"thermal_frame"`, `"rgb_frame"`, `"gsr_data"`, `"audio_data"`, or sync/events) [67]. The PC's AndroidStreamReceiver dispatches messages based on this type (it expects types like `"gsr"`, `"rgb_frame"`, `"thermal_frame"`, etc.) and parses the payload accordingly [68] [69]. Each data type handler either processes the data or simply logs/plots it. For instance, receiving a `thermal_frame` might include a file path or byte count rather than the whole image due to size concerns [70].
- **Data format compatibility:** One finding in the audit is a slight mismatch: the Android `GSRDataEvent` is tagged with serial name `"gsr_data"` [71], whereas the PC expects type `"gsr"` in its handler mapping [68]. This inconsistency would need aligning (either on Android to label it just "gsr" or on PC to accept "gsr_data") so that GSR messages are recognized. Other types appear consistent (Android uses "rgb_frame", "thermal_frame", which PC also references).

- **Sync and control messages:** The protocol also allows special event types. The PC code listens for events like `"event"` with subtypes such as `"session_start" or "session_stop"` to know when a recording began/ended [72]. The Android app likely sends these when recording is started or stopped. Additionally, a `"SyncCommand"` message type exists for time synchronization commands (to align clocks) [73], and `"SensorQuery"/"SensorResponse"` for on-demand sensor info queries [74] [75]. The NetworkingManager on Android has methods to send sync commands (with a timestamp) to connected devices [76], which the PC's AndroidSyncManager could use to calculate offset between PC and phone clocks. In practice, implementing a robust clock sync (similar to NTP or using LSL's clock) would improve cross-device alignment. The current approach likely uses the first connection or a ping exchange to estimate offset (the audit did not find the exact method, but the presence of time offset storage in AndroidSyncManager suggests such calibration).

- **Connection setup:** For Wi-Fi streaming, the simplest scenario is the Android phone and PC are on the same network. The PC listens on a known IP/port and the phone, acting as client, could connect to that IP. However, the code appears more focused on phone-to-phone (where one phone is master). There isn't an explicit UI to enter a PC IP on the app, which suggests either (a) using Wi-Fi Direct between phone and PC (not typical), or (b) relying on LSL for PC connectivity. It's possible the intended route is that the Android app uses LSL for PC, and reserves socket networking for phone clusters only. If socket to PC is desired, the app might need a small addition (like reading a config for server IP or scanning the LAN for the PC host). In summary, the socket protocol is there and matches PC's expectations aside from minor naming tweaks; using it may require a manual configuration step not currently in the UI.

- **Lab Streaming Layer (LSL):** The app contains an `LSLStreamingService` which publishes data streams via LSL, a common framework in research for syncing multimodal data across devices. If enabled, the phone will act as an LSL source:

- The service (in simulation mode unless liblsl native libraries are present) defines LSL **outlet streams** for GSR, RGB frames, Thermal frames (and possibly PPG as a separate stream) [77] [78]. Each stream is named (e.g., `"GSRRGBT_GSR"` for GSR) and has a channel format (e.g., 2 channels for GSR+PPG values, double format) [79].
- When data events occur, the LSL service's `sendData()` will push samples to these outlets. For example, a GSR event sends a double array of [gsr, ppg] with the event's timestamp [80]. Frame events send a JSON string of metadata (timestamp, frame index, etc.) as a sample on their stream [81] [82]. Because sending full image frames over LSL is impractical, this design only streams metadata and potentially a path or size, requiring the actual images to be stored or transmitted separately if needed.
- The advantage of LSL is that the PC can use standard LSL libraries (e.g., pylsl in Python, which the FYP-GSR-Windows likely uses, given references to `pylsl` in its docs [83]) to receive streams with automatic timestamp alignment. LSL inherently handles clock offset calibration between devices. The FYP-GSR-Windows "LSL Integration" was noted to be activated by simply installing pylsl [83]. So if the Android app enables LSL mode and the PC is listening for those streams, the data should flow with minimal custom networking code.
- **Current status:** The LSLStreamingService code in Android is mostly functional but runs in a simulated mode unless actual LSL native libraries are bundled (the code checks for `liblsl.so` and logs a warning if not found [84]). It does attempt to load `System.loadLibrary("lsl")` and a custom JNI `gsrrgbt_lsl_jni` library if available [85]. This indicates that for full use, the project expects the LSL native binaries to be compiled into the app (which may not have been done yet, given the simulation fallback). On the PC side, however, using LSL might currently be easier than sockets because the PC app is already configured to create LSL streams for its own sensors and can incorporate the phone's streams simply by discovery. The audit suggests verifying if the Android app has an option in settings to turn on LSL; if not, adding one would be beneficial.

Sync Markers and Coordination: The integration between phone and PC should also consider **synchronization markers**. The Android app generates `SyncMarkerEvent` events (e.g., when recording starts, or on user-triggered markers) [86]. These are timestamped events labeled with a `markerType` like "start", "stop", or "calibration" [87]. On the PC, these might be logged or used to align timelines of different data logs. For multi-device phone-to-phone scenarios, the NetworkingManager broadcasts sync commands (for example, one phone can send a "START" command to another so they begin recording simultaneously) [76]. In a phone-to-PC scenario, since the PC is not capturing its own video, the sync is mainly about aligning the phone's data clock with the PC's log clock. There is an `AndroidSyncManager` on PC that likely calculates an offset (by having the phone send a timestamp and comparing to PC time) [88] [89]. Including periodic sync pulses or common reference events (like both devices logging an external signal or manual marker) is recommended to verify alignment drift over time.

Testing Protocol for Coordinated Streaming: To ensure the Android and Windows apps work in unison, a testing strategy is needed: - **Initial Connectivity Test:** Start the PC server (or LSL receiver) and attempt to connect the Android app in each mode (Bluetooth, Wi-Fi Direct, or Wi-Fi TCP, or LSL) and verify the PC UI shows a connection established. For TCP, the phone may need an IP:port input (could be added temporarily in code or via Wi-Fi Direct auto-discovery if both on same network). - **Synchronization Test:** Generate a known simultaneous event visible to both systems. For example, start a recording and perform a quick action like tapping a finger in view of the camera while simultaneously touching the GSR electrodes (causing a spike) – then stop and compare the PC's recorded data: the video frame of the tap and the GSR spike timing should align to within the expected sync error (<50 ms). The PC's logs with sync markers can help – e.g., ensure PC logged "Android session start" and used that to align with any PC-side data. - **Data Throughput Test:** Stream for a sustained period (e.g., 5 minutes) and monitor for dropped packets or delays. The PC's `AndroidStreamReceiver` keeps stats on bytes and packets received [90] [91]. If using UDP for frames, allow some tolerance for lost packets; if TCP, ensure the PC receives all events eventually (no infinite buffering on phone side). - **Multi-Phone Scenario:** (if relevant) Pair two phones (one as master, one as slave via Bluetooth or Wi-Fi Direct) and also connect the master to the PC. Then start a multi-device recording. Check that PC receives data from both (the PC's receiver identifies clients by address). This tests the NetworkingManager's ability to handle multiple streams and the PC's ability to separate them (currently, PC identifies by client address string).

In summary, the Android app is capable of streaming data off-device, but configuration and testing are needed to fully utilize it. For a single-device demo, one could simplify by using LSL (if integrated) or by enabling a mode where the phone writes everything locally and the data is transferred after. If real-time PC monitoring is required, minor fixes (like the GSR type name and adding a way to specify the PC host) will be high priority.

1.4 Current Limitations and Unimplemented Features

The audit identified several **limitations, placeholders, and partially implemented features** in the current codebase:

- **GSR/PPG Sensor Integration:** As noted, the Shimmer GSR+ support is not completed. The `GSRCaptureModuleSimulation` explicitly states it's a working implementation "while Shimmer integration is being developed" [5]. No code exists yet to connect to the actual BLE GSR device, meaning **no real GSR data** can be recorded by the app at this time. This is a critical feature gap since the GSR is central to the project's purpose. The risk is that any demo relying on physiological signals must use simulated data, undermining realism. Implementing and testing the Shimmer BLE data stream (handling BLE services, characteristics, and ensuring 128 Hz reliable streaming) is outstanding work.

- **Device Pairing Workflow:** Following from the above, the UI path for selecting a BLE device does not lead anywhere. After selecting a device in the `DevicePairingDialog`, the app should initiate a connection and possibly reinitialize the GSR module with the target device. The code to do this is marked as TODO and not present [33]. Thus, even if a Shimmer device is discovered, the app won't actually start receiving its data. This whole workflow needs finishing. Additionally, scanning for BLE requires **Location permission** (which is requested) and on Android 12+, BLUETOOTH_SCAN permission (requested) – those are in place [53]. The app does not currently remember or auto-reconnect to a previously paired sensor; that could be a future enhancement once basic connectivity works.

- **Thermal Camera Preview and Calibration:** The integration with the Topdon thermal camera is in place, but some aspects may be limited:

- The preview rendering for thermal is not implemented (the code to draw the thermal image to the TextureView's Surface is incomplete) [43]. For a demo, this means the user might not actually see the thermal image on the screen, which is a major UX issue. They would see only the RGB or a blank view when toggled to thermal. Implementing a basic grayscale or pseudo-color render from the raw thermal frame bytes (which likely represent temperature per pixel) is needed for a meaningful demo.

- There may be **calibration or temperature scale issues**: the Topdon SDK provides raw temperature data; converting that to a human-viewable image or specific temperature values might require applying a colormap and knowing the sensor's range. Currently the DataWriter even calculates min/max/avg temperature per frame [92], but without display, that info is only in logs. If absolute temperature accuracy is needed (for example, to detect fever-level temperatures or subtle physiological changes), further calibration and testing with the device would be needed. For now, the focus might be just visualizing relative changes (for stress, the interest is likely in perspiration on skin which correlates with GSR, or blood flow changes).

- Another limitation: **USB Permission handling**. The user must grant permission when the camera connects; if not, the app will not see the device and will go to simulation. The code doesn't explicitly show requesting this (likely it relies on the system broadcast when the Topdon is attached). Ensuring the app gracefully handles being started with the camera already plugged vs plugging in while running is an edge case to test.

- **Hybrid Camera Mode:** In the UI, there is mention of a `HYBRID` camera mode (to possibly overlay thermal and RGB) [93]. However, there is no implementation provided for how the hybrid view works. The `HybridPreviewView` object is declared but not clearly defined in the snippet we saw. This suggests that hybrid mode (fusing thermal and visible imagery in one view) was a planned feature but not realized. It's a nice-to-have (not critical for core functionality) but currently a user selecting "Hybrid" might either do nothing or cause an error. This can be marked as an unimplemented feature for future development once basic modes are solid.

- **Settings and Configurability:** A `SettingsActivity` exists, presumably to adjust parameters, but in the RecordingController the actual usage of settings is rudimentary (many values are just defaulted with a comment that the module should be enhanced to use them) [94] [44]. For example, the code hints at support for different RGB resolutions (720p/1080p/4K) and different GSR sample rates (64Hz, 128Hz, etc.), but currently the RecordingController does not pass those to the modules (the modules themselves also have fixed internal constants for frame size and sampling). So effectively, the settings UI (if implemented) is not wired up. This is a limitation because any options shown to the user might not actually do anything. Also, the Settings may allow toggling network mode (local vs network recording), selecting network type (Bluetooth vs Wi-Fi Direct vs LSL), participant ID input, etc., but these need to be verified. Without a working settings flow, the app is stuck in default configurations (which are fine for demo, but if the demo scenario required, say, turning off audio or switching network mode, that might not be possible from UI).

- **Stability and Error Handling Gaps:** While many try-catch and status flags exist, some potential stability issues include:

- **Stopping and restarting recording** quickly might not be well-tolerated. For instance, after stopping, some threads might still be shutting down (the code uses coroutines which cancel jobs and waits for them in some cases). If the user hits Start again too soon, or if an error prevented a clean stop, it could cause a crash (e.g., camera already open exception, or file writer still busy). There is no explicit cooldown or "recording in progress" lock beyond a boolean.
- **Memory usage:** Storing video frames in memory could be heavy. The design avoids queuing too many frames by processing them as they come (the SyncManager immediately records or streams them), but if the PC or disk writing can't keep up for some reason, there is a risk of memory buildup. The DataStreamer uses a channel with capacity 1000 for streaming events [95]

[96], meaning if the network is slow, up to 1000 events might buffer (this could be ~1000 video frames or dozens of audio segments). In extreme cases, this could cause OOM on phone. There is a warning in code to drop events if channel is full [97], which is good, but then data loss occurs. It's a trade-off to monitor.

- **Threading issues:** Some parts (like writing files, or handling socket callbacks) use `withContext(Dispatchers.IO)` appropriately for long operations [98] [61]. However, interactions with Android UI (if any) should be on the main thread. The provided code segments show UI updates likely done via LiveData/StateFlow observers in the MainActivity (which uses `lifecycleScope.launch` and repeatOnLifecycle to collect status flows). That is a correct approach. But if any module tries to directly update a TextView from a background thread, that would be a bug. No such direct calls were seen, implying the design is sound in this regard.

- **Permissions on newer Android:** Android 11+ restricts background location access and changes how Bluetooth scanning works (needs fine location + BLUETOOTH_SCAN runtime permission). The app requests the needed ones, but another possible issue is **external storage write**. They request WRITE_EXTERNAL_STORAGE (for saving files), which on Android 11+ might not work unless the app targets an older SDK or uses the app-specific directories (they use `getExternalFilesDir`, which should be fine without special permission). The CsvDataWriter indeed uses `context.getExternalFilesDir("recordings")` which is within the app sandbox on external storage [99], so it's okay. Thus, no critical storage permission issues as long as the user grants it on older devices; on newer, it might not even prompt since app external dir is accessible by default to the app.

- **No Video Compression or Export**: The app does not compress or encode the RGB or thermal frames into video files (e.g., MP4). It only logs metadata and possibly leaves the frames in memory or sends them out. This means that after a recording, you don't directly get a playable video file from the device – you would need to reconstruct it from frames or rely on the PC to store video if it was streaming to PC software that does recording. The README's mention of "compressed video" and MP4 might be aspirational [62]. In the current implementation, if we need actual video clips, an upgrade is needed to encode frames (using Android's MediaCodec or MediaRecorder to record the camera output to MP4, for example) or at least save frame JPEGs. Not doing so is a limitation because a research user might expect to get a synchronized video+sensor dataset from one device. However, this is somewhat mitigated if the PC app is used as the primary recorder (the PC could combine the streams and produce outputs). For a standalone usage, adding video recording would be necessary in the future.

- **Multi-Device Sync Complexity:** The app claims to support multi-device synchronized capture (across multiple phones) via Bluetooth or Wi-Fi Direct networking [100] [101]. While the code exists to form connections and exchange data events between phones, this is a complex feature that may not have been fully tested. Potential limitations here:

- If using Bluetooth: phones communicate with one master. BLE is low-bandwidth (not ideal for video), so it likely uses Bluetooth Classic SPP if at all. The code appears to use Bluetooth for coordination and possibly small data (maybe sync signals) and suggests using Wi-Fi Direct for the heavy data. But managing two network types simultaneously is tricky. The NetworkingManager can choose one at a time in current code (NetworkType enum) [14].
- Wi-Fi Direct: Not all devices handle Wi-Fi P2P reliably. Group formation might fail on some phones, or reconnection logic might not recover if one device drops off. The code does have event listeners for failures and retries, and the documentation mentions "automatic reconnection with retry logic" [102], but those need real-world testing.

- The multi-device sync ultimately relies on the fact that each phone's data is timestamped to its own clock plus periodic sync commands. Without a hardware sync trigger, there will always be some small offset (they attempt software sync within 50 ms). That's likely acceptable for the project (e.g., aligning GSR peaks to video within a few milliseconds is fine), but the limitation to acknowledge is that perfect sync (like you'd get with a shared trigger or a single device) is hard to achieve with entirely separate devices. The approach is reasonable but demands a thorough validation if multi-phone data is going to be used for scientific measurement.

In summary, while the core architecture is in place, **several important features are still placeholders or minimally tested**: real GSR sensor support, the thermal preview, robust multi-device operation, and the settings UI. These limitations inform which upgrades are most urgent, as described in the plan section below.

1.5 Stability, Compatibility, and Requirements

Finally, some notes on the app's stability and environment requirements:

- **Android Version Compatibility:** The app requires Android 8.0 or above [46]. It makes use of Camera2 API (requires API >= 21, so fine) and foreground service for remote control (likely uses modern APIs). One thing to watch is that on Android 13, new Bluetooth permissions for advertising and connects are needed, but the app only requests SCAN and CONNECT which should suffice for BLE operations. Storage permission is not needed for the default save location, as discussed, but since it is requested it's fine. The app should be tested on at least one device running a recent Android (11 or 12) to ensure there are no surprises with changed permission models (the code is mostly aware of them).

- **Bluetooth LE Requirements:** Using the Shimmer sensor over BLE requires that the phone supports Bluetooth 4.0+ and that Location is on (for scanning). The app checks for Bluetooth enabled and will show a message if not [103]. Also, scanning is timed out after 10 seconds by the dialog code [104] [105]. Stability issues that can arise:

- BLE scan not finding the device (common if sensor not advertising or if phone's BLE is flaky). This might confuse users; adding a refresh or ensuring the Shimmer is in pairing mode is necessary.
- BLE connection dropping: the plan (from documentation) was to implement auto-reconnection with exponential backoff [106]. It's unclear if that is implemented yet on Android (the PC side summary mentioned adding reconnection logic for COM ports). The Android code doesn't obviously show a reconnection loop for GSR (since it's not implemented at all, it's moot right now).

- Throughput: 128 Hz data + some PPG and perhaps battery info is not a lot (couple KB/s), so BLE can handle that easily. The main requirement is making sure the BLE operations are properly on a background thread and not blocking the UI (most BLE callbacks are fine).

- **Performance Observations:** According to the README and design documents, the system should handle:

- Thermal ~25 fps, RGB 30 fps, GSR 128 Hz, audio continuous, all without dropping samples in a typical 15-minute session [107]. These numbers are plausible given modern smartphones, but pushing the phone (especially doing thermal image processing and video simultaneously) might cause high CPU usage and heat. The code's use of separate threads and avoiding heavy processing (no real-time computer vision or such in this app) suggests it can meet these rates. If anything, the thermal camera might internally be limited to 9 Hz if region-locked, but Topdon claims 25 Hz so we assume that.

- The app tries to maintain frame sync within 50 ms [108]. This implies that the time difference between an RGB frame and the nearest thermal frame or GSR sample is <50 ms. The use of a single clock for timestamps and starting all captures together supports this. However, scheduling delays could occasionally introduce jitter (e.g., GC pause or a slow disk write). Testing would reveal if any modality lags. For instance, GSR is continuous while frames are discrete; you might see up to ~20 ms difference if a GSR sample arrives just before a frame timestamp, etc. Overall, <50 ms is a reasonable goal (roughly within 1-2 video frames).

- **Battery life:** The README suggests ~2 hours continuous use on battery [62]. This will vary by device, but running two cameras and BLE and writing to storage is heavy. The phone will likely warm up. It's advisable to test under expected demo conditions (e.g., 10 minutes recording) to ensure the phone doesn't thermal throttle or the app doesn't slow down. Lower-end or older phones might struggle with simultaneous thermal+RGB due to I/O bandwidth.

- **Known Issues:** No major crashes are documented in the repository (no GitHub issues section appears to be used for this). But from code review, some potential issues to be aware of:

- If the Topdon SDK is not present or fails, the app goes to simulation without notifying the user clearly that "Thermal camera not found, using simulation". It sets a status string, but an average user might not realize. This could be mitigated by a toast or dialog – currently not done.
- If the external libraries (Topdon, LSL) are not added, the code still compiles (thanks to stub classes) but obviously features won't work. This is fine for development but for deployment one must remember to include those AARs. The risk is forgetting the Topdon AAR – in which case the ThermalCaptureModule would always be in simulation even if camera is attached (since stub ThermalSDK `initialize()` always returns true without actually doing anything).
- The application expects certain permissions to be granted. If the user denies some (say, location or Bluetooth), the app might not properly handle it beyond a generic "permission denied" toast. It doesn't disable features in the UI, so pressing "Start" with missing permissions could cause exceptions. The code does check and show a message if not all permissions granted on startup [109], so it likely prevents starting until fixed, which is good.
- There is no in-app guidance or tutorial – which is okay for an engineering audience, but for a polished product one would add instructions in the UI. For now, assume a knowledgeable operator.

Conclusion of Audit: The FYP-GSR-Android app is an ambitious integration of multiple sensing modalities on Android. The code architecture is largely sound and aligns with the project's design goals (modularity, synchronization, real-time streaming). The primary issues are that some crucial components are left in a prototype state (notably the GSR sensor support and certain UI/UX elements like the thermal view), which could impede a successful demonstration if not addressed. Android and hardware compatibility seem largely considered, but will require real device testing. The subsequent section outlines an upgrade plan prioritizing these findings to make the application demo-ready and robust for future use.

2. Phase-by-Phase Prioritized Upgrade Plan

To improve the FYP-GSR-Android application, we propose a **phased upgrade plan** with each phase addressing a set of issues in order of importance. The plan is structured to ensure that the most critical fixes and features (those needed for a successful upcoming demo and core functionality) are implemented first, while later phases focus on enhancements for robustness and future capabilities.

Each phase is described with its objectives, rationale (why it's critical), and guidance on implementation including design decisions or trade-offs.

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 demo. This phase targets the "blockers" that currently force the app to use simulated data or could cause a demo to fail.

Key Tasks:

1. **Implement Shimmer GSR+ Sensor Support (Replace Simulation):** This is top priority because real GSR data collection is central to the project. Use the Shimmer Android BLE API (or directly interact with GATT services if not using their SDK) to connect to the GSR+ device. Specifically:
2. Initialize a real `GSRCaptureModule` class that scans for the device (by name or MAC, possibly utilizing the device selected in the pairing dialog) and connects via BLE. Upon connection, subscribe to the GSR and PPG data streams provided by the Shimmer (typically, Shimmer streams data by notifications on specific characteristics).
3. Parse incoming BLE packets to extract GSR (in microSiemens or raw ADC counts) and PPG values. Shimmer's documentation will define the packet format and how frequently data comes (128 Hz means ~ every 7.8ms a packet or perhaps packets of multiple samples). Ensure that each reading gets timestamped with `SystemClock.elapsedRealtimeNanos()` at reception (or optionally use the Shimmer's timestamp and adjust to phone clock – but initial simpler approach is use phone time on arrival, as documented in research paper [110]).
4. Feed the GSR readings into the existing syncManager via `syncManager.onGSRData(sampleData)`, similar to how simulation does it [111] . This will create and queue `GSRDataEvent` instances that include the real sensor values.
5. **Auto-reconnection:** Implement basic logic to handle if the BLE connection drops. For demo, at least notify the UI "Reconnecting…" and attempt a reconnect a couple of times (exponential backoff as planned). This prevents having to manually re-pair mid-session if interference or distance causes a dropout.
6. **Testing & Calibration:** Test this with the actual Shimmer device in hand before the demo. Check that the values make sense (e.g., if you touch the GSR leads, does the app display a rising GSR value on the UI?). Adjust unit conversion if needed – Shimmer might send raw values that need scaling to μS. Also verify heart rate calculation from PPG (if planned) – though that might be a Phase 2 item (implementing a peak detection algorithm if not provided by Shimmer).
7. *Trade-offs:* Using Shimmer's official API vs custom BLE code. The official API might simplify subscription but could be heavy; given time constraints, directly handling BLE might be quicker if sample code is available. The key is reliability: ensure we don't drop GSR samples. Even if we reduce sampling to, say, 64 Hz to be safe, that's fine; 128 Hz is ideal but quality of GSR analysis likely won't suffer if a bit lower, as long as sync is maintained.
8. **Fix Permission and Initialization Issues:** Some basic fixes that can drastically improve stability:
9. Request **RECORD_AUDIO permission** on startup (add to `requiredPermissions` and manifest). Without this, the audio module will always simulate. After adding, test that the app actually records real audio (one can speak or clap during a test recording and later examine the audio file or stream).
10. Ensure the app handles the USB permission for the Topdon camera. On launch, or when starting thermal module, check `UsbManager.hasPermission(device)`; if not, request permission via

`UsbManager.requestPermission(device, PendingIntent)`. This might prompt a dialog the first time. Doing this in Phase 1 ensures the thermal camera can turn on smoothly in a demo scenario without manual fiddling in system dialogs.

11. In the UI, gray out or hide features that aren't ready to prevent accidental use. For example, if LSL or multi-device won't be used in the demo, and clicking those might cause confusion, it may be best to disable those controls (or at least ensure they're not throwing errors). This is a temporary demo-hardening step.

12. Validate that all required runtime permissions (Camera, Location for BLE, etc.) are indeed being requested and handled. We saw location, BT, storage are requested – after granting, the app should be fully operational. If the user denies any, handle gracefully (e.g., a pop-up "App cannot run without X permission").

13. **Thermal Camera Preview Display:** Make the thermal camera feed visible in the app:

14. Implement the `renderThermalFrameToSurface()` method in ThermalCaptureModule [43]. A simple approach: take the ByteArray `frameData` (which presumably contains 16-bit temperature values for each pixel) and convert it to an 8-bit grayscale bitmap. If the Topdon SDK provides a utility (sometimes thermal SDKs give you a YUV or RGB888 image directly), use that. Otherwise, manually map temperature range to 0-255. For example, find min and max in the frame (we already compute those in `analyzeThermalFrame()` for logging [92]) and scale each pixel = (value - min)/(max-min)*255. This will produce a grayscale image highlighting temperature differences.
15. Use Android Canvas or Bitmap APIs to draw this image to the `Surface` associated with the thermal TextureView. This can be done by constructing a Bitmap of size 320x240 (interpolated dimensions) and filling it with pixels from frameData. Then post the bitmap to the TextureView via `Surface.lockCanvas` or by using an ImageReader with the surface. Given time, even an approximate solution (like mapping the 16-bit to one channel) is fine – the goal is the demo operator and audience can see something representing heat.
16. **Result:** Now toggling to "Thermal" mode in the app should show the live thermal video. This is critical for demo, since demonstrating the thermal camera is likely a highlight (e.g., showing heat patterns on someone's face or hand).

17. *Note:* The color palette could be grayscale or a simple pseudo-color (blue-to-red). Fancy palettes or high accuracy can be Phase 3; initially, any visible output is better than none. The design trade-off is balancing frame rate vs processing: converting 25 fps of 256x192 to a bitmap on CPU is okay (it's not a lot of pixels), but ensure to do it in the background thread to not block UI. The ThermalCaptureModule's coroutine can handle this.

18. **Verify and Stabilize Data Recording:** After making the above changes, conduct a full integration test of a recording:

19. Connect the real hardware: plug in the Topdon camera, pair with the Shimmer, grant all permissions.
20. Start a recording for a short period (1-2 minutes). Stop it. Then check that:
    - All CSV files are created and contain data. Particularly, GSR CSV should now have real values (not just the simulated pattern), Audio CSV should have data_size non-zero and audioLevel varying if there was sound, etc.
    - No crashes or freezes occurred during start/stop. If any exceptions appear in logcat (e.g., null pointer on stop if something wasn't initialized), fix those. For instance, if the new

GSRCaptureModule wasn't properly stopped or a Bluetooth thread remains open, ensure `release()` closes it.

21. This testing might reveal small bugs (like maybe forgetting to call `thermalSDK.release()` after stopping capture – but the code has a `releaseTopdonSDK()` doing it [112] [113]). Pay attention to battery and performance during the test as well (the PerformanceMonitor if enabled can log CPU or FPS). If you notice any module lagging (e.g., thermal dropping frames or audio stuttering), note it for Phase 2 optimization.
22. Also test the UI under these conditions: Does the status text for each module properly show "Capturing" or errors if any? Does switching the preview from RGB to Thermal mid-recording work? (It should, since both are capturing anyway; if not, we might restrict mode switching during capture to simplify.)

Justification: Phase 1 addresses the fundamental capability of the system – recording actual synchronized multimodal data. Without real GSR and a visible thermal stream, the demo would be a "simulation", which undermines the purpose. By focusing on these, we ensure that by the end of Phase 1 the app can truly showcase **all four modalities (RGB, thermal, GSR, audio)** working together. This will greatly enhance demo credibility: for example, one can show their hand in front of the camera and see both the thermal imprint and a spike in GSR as they get nervous, all in real-time. Fixing permissions and minor bugs avoids embarrassing moments where the app might crash or silently fail in front of an audience. Essentially, Phase 1 transforms the app from a mostly theoretical implementation to a practically usable tool.

Phase 2: Demo-Readiness and Performance Optimization

Objective: Now that core functionality is in place, Phase 2 focuses on refinements that ensure the demo runs smoothly and the data quality is sufficient. This includes optimizing performance (to prevent lag or drops), improving the user interface feedback, and implementing any features critical for the demo scenario (like basic analysis or visualizations if needed to illustrate the project's goals).

Key Tasks:

1. **Optimize Frame Handling and Throughput:** To guarantee stable frame rates and synchronization:
2. **Network/Bandwidth adjustments:** If the demo involves streaming to a PC, consider the volume of data. For instance, sending raw frame bytes as JSON (even if Base64 encoded) is inefficient. A short-term optimization: do not send the full image over TCP; instead, send just metadata (already the code seems to do that, but double-check). Possibly integrate a mechanism on PC to fetch images from phone if needed, or skip image sending entirely if local saving is enough. For purely on-phone demo, disable network streaming to save CPU.
3. **Thread priorities:** On Android, the thermal and RGB threads already run in Dispatchers.Default which should map to background threads. We could experiment with using `Thread.setPriority(Thread.MAX_PRIORITY)` for those threads to avoid UI or other processes interfering. However, be cautious: too high priority might starve other tasks. Usually, just ensuring they're not on the main thread is enough.
4. **Drop policies:** It's better to drop frames than to lag and accumulate latency. The DataStreamer already drops events if the channel is full [97] . We might want to adjust the channel size or drop threshold. If, say, network is slow, we prefer to drop some thermal frames rather than queue 1000. We could lower `STREAM_BUFFER_SIZE` from 1000 to something like 100 for demo, since real-time display doesn't need that backlog. This is a trade-off between completeness of data vs real-time performance, but for demo, real-time is king.

5. **Memory profiling:** Use Android Studio's profiler or simple logging to check memory usage over time during a test recording. Ensure that after stopping a recording, memory usage drops (meaning no leaks). If any large objects persist (e.g., bitmaps, buffers), explicitly null them or use `System.gc()` at safe points (not generally recommended, but for demo could be acceptable after stopping recording to quickly free memory).

6. If the phone runs hot, consider reducing some loads: e.g., limit thermal to 15 fps if full 25 fps isn't needed, or reduce RGB to 720p if 1080p isn't visibly different in demo context. These can be quickly toggled via the SettingsRepository or constants. The expected outcome is a smoother, cooler run at the cost of some resolution—an acceptable trade-off if stability is improved.

7. **UI/UX Improvements for Demo Clarity:**

8. **Real-time Data Display:** In a live demo, it's powerful to show the sensor readings updating. The app already displays real-time GSR value (in μS) and heart rate on the UI according to README [114]. Make sure these are indeed wired up. If not, implement a simple binding: e.g., have the GSRCaptureModule (real one) update a `MutableStateFlow<Double>` for current GSR value which the MainActivity observes and displays in a TextView. Similarly for heart rate if computed. If time allows, even a simple line graph or trend indicator for GSR could impress – but that might be beyond scope now. A numeric value updating is still good (maybe update it once per second to be readable).
9. **Status Indicators:** Ensure that each modality's status (Connected/Ready, Capturing, Error) is visible. Possibly use small icons or color coding (e.g., a green dot next to "Thermal" label when capturing, red if error). The groundwork is there with `statusOverlay` TextView and such. Just confirm that the RecordingController's status flows are being observed and reflected. This helps the presenter quickly diagnose if something is not running (for example, if GSR sensor disconnects mid-demo, an on-screen warning should appear).
10. **Stream Start/Stop Feedback:** When the user taps "Start Recording," give immediate feedback. If initialization takes a second (especially now connecting BLE etc.), consider showing a loading spinner or a countdown. The code uses a CountDownLatch and likely starts nearly instantly, but if any delays occur, a brief "Starting…" status (which ThermalCaptureModule does set [115]) should be seen. Likewise, after tapping stop, maybe change the button to "Stopping…" until fully stopped. This prevents confusion of multiple taps.
11. **Thermal/RGB Toggle Behavior:** If the user toggles to thermal view when the camera isn't connected, currently it says "Simulation Mode" on status but showing a blank screen might confuse. It might be wise to either hide the toggle if no real thermal camera (or label it differently like "Simulated Thermal"). But since Phase 1 aimed to always have the camera, this is less an issue. Still, in case the camera disconnects mid-demo, handle that gracefully: e.g., catch the error in frame callback and set status "Thermal camera disconnected".

12. **Prevent Mistakes:** During demo, you likely want to avoid the operator accidentally hitting settings or device pairing and losing time. If demo script doesn't require pairing on the fly (we'll pre-pair the sensor beforehand), you might disable the "Device Pairing" menu during an active recording or altogether for the demo build. Similarly, if multi-device sync isn't used, maybe hide the networking controls (if any in UI). These temporary UX changes reduce risk of user error in a high-pressure scenario.

13. **Initial Data Analysis (optional, if critical to project goals):** Depending on the project's focus (for example, if it's about estimating stress or correlating signals), it might strengthen the demo to have a minimal real-time analysis:

14. **Heart Rate from PPG:** If the Shimmer provides a PPG waveform, implement a simple heart rate calculation. The README claims real-time HR is included [116]. Possibly Shimmer's firmware can compute HR, or we have to do peak detection. A quick hack: measure time between successive PPG peaks (maybe by checking when PPG value crosses a threshold or using a basic derivative) and calculate BPM. Even if not super accurate, demonstrating the app showing "Heart Rate: 85 BPM" next to GSR can be impressive. Ensure to smooth it (average over last 5-10 seconds) to avoid it jumping wildly.
15. **Thermal ROI Measurement:** Another possible demo-oriented feature: if the use-case is stress, one might want to track temperature on the participant's forehead or nose (common stress indicators). Doing full computer vision face tracking is complex, but as a proxy, maybe allow a simple selection of a region in the thermal view and calculate average temp there continuously. This could be a stretch goal for Phase 2 if it directly supports the research question (like demonstrating how thermal correlates with stress sweat on skin). If not central, skip it.
16. **Data Logging Confirmation:** After a demo recording, it's useful to quickly verify data was captured (maybe even show an excerpt). While the app can't easily show CSV contents on-device, one could implement a summary popup: e.g., "Recording saved: 300 RGB frames, 300 Thermal frames, 18000 GSR samples, 120 audio segments" by reading the SessionStatistics [117] [118]. This reassures that the sensors ran correctly. It's not strictly necessary, but if the demo format allows, showing that summary or sending it to the PC live (the PC UI could indicate packet counts etc. which it does [90]) demonstrates completeness.

Justification: Phase 2 ensures the app isn't just working, but working **well** and impressively. A demo scenario likely involves live operation of the app in front of observers – thus performance (no lag, good frame rate) and clarity (what's happening is visible) are paramount. The optimizations here reduce the chance of things like frame drops or app slowdowns which could disrupt synchronization or cause awkward pauses. The UI improvements make the system's functioning transparent, which is important for convincing observers that the multimodal capture is actually happening in sync (they can see values update together, etc.). Additionally, addressing any smaller bugs discovered in Phase 1 tests is part of stabilization – we want to walk into the demo with high confidence in the app's reliability. Trade-offs made: some heavy or risky features (like advanced analysis or multi-device) are deferred to ensure we don't introduce instability at this stage. Phase 2 is about polishing what we have and making it **demo-ready** in terms of user experience and data fidelity.

Phase 3: Comprehensive Multi-Device Synchronization and Networking

Objective: With the single-device use-case solid, Phase 3 expands the capabilities to fully support **multi-device scenarios and PC integration**. This phase is about fulfilling the system's networking promises – enabling synchronized recording from multiple phones and seamless data streaming to the PC base station (FYP-GSR-Windows). It also involves testing and refining the time synchronization across devices.

Key Tasks:

1. **Finalize Wi-Fi Direct and Bluetooth Master/Slave Features:** Ensure that two or more Android devices can truly work in sync:
2. **Connection UI:** Provide a way for the user to initiate networking mode. Perhaps in Settings or on the main screen, a toggle "Multi-Device Mode" with options: Master or Slave and via Bluetooth or Wi-Fi Direct. In master mode, the device will wait for connections; in slave, it will scan for the master.
3. **Bluetooth Networking:** Useful for small data and coordination. For multi-angle scenarios (maybe two phones each capturing video+GSR), Wi-Fi Direct is preferred due to video bandwidth,

but Bluetooth could be a fallback for sync signals or if only GSR is on one device and camera on another. Test the `BluetoothNetworkingService` by trying to pair two phones. Likely it uses Bluetooth classic SPP (since BLE wouldn't handle video). Make sure pairing and socket connection works. Also implement any needed user flow (like selecting the other phone from a list of discovered Bluetooth devices – similar to how pairing the sensor works).

4. **Wi-Fi Direct:** This is more complex but powerful. The code sets up a group, but to test it:
   - On one phone (Master), call `startAsMaster()` for WiFi service [119]. On another (Slave), call `startAsSlave()` and then some method to connect to the master's device (perhaps the service broadcasts or lists peer devices – the code listens for peers and emits `DeviceDiscovered` events [120]). We might need to implement a UI to select the master from available peers, then call `connectToDevice(deviceAddress)` on the slave [121].
   - Once connected, verify that data events are indeed exchanged. The design likely is that each phone's SyncManager sends data events to the active service which then transmits to others [122]. Confirm that, for example, the slave phone's thermal frames are received on master and maybe saved or forwarded to PC if connected.
5. **Time Sync Between Phones:** One phone is master time source. The SyncManager might be updated to adjust timestamps from the slave by an offset. If not already, implement a simple sync message: when a slave connects, have it send a "hello" with its current `elapsedRealtimeNanos`. Master compares with its own time and computes offset. Then master instructs slave to apply that offset to all its event timestamps (or master adjusts on receipt). This way, all events in the combined dataset share a time basis. Use one of the NetworkEvent types (like `SyncCommand`) for this coordination. Since Wi-Fi Direct latency is low (<50ms), doing a quick ping-pong to calculate offset is fine. Repeat this occasionally (e.g., every minute) if recording is long, to correct any drift.

6. **Testing:** Try a scenario: Master phone capturing thermal, slave phone capturing RGB (just for testing multi-device). Start recording simultaneously via the network control. Stop after some time. On master, check that it received data from slave (maybe it logs in CSV or at least prints something). More practically, you might unify data saving: e.g., if devices are networked, maybe only master saves to CSV (including events from slaves). The code appears to lean that way (NetworkingManager could feed remote events into local SyncManager by generating NetworkEvent.DataReceived events [74], which could then call syncManager.recordEvent() on master). Ensure this flow works – you might need to add handling in NetworkingManager or SyncManager to integrate remote DataEvents into local logging. This is tricky but important if the goal is one combined dataset.

7. **Robustness:** Implement reconnection attempts for Wi-Fi Direct similar to what was planned for BLE. Wi-Fi Direct can occasionally drop; having the slave try to reconnect or at least not crash if master disconnects is key. The NetworkingService.NetworkEvent events (ConnectionStatusChanged, etc.) should be observed to update UI (like "Disconnected, retrying…"). Set appropriate timeouts and perhaps allow manual retry via a button.

8. **Enhance PC Streaming Compatibility:** Now focus on connecting the Android network with the PC:

9. **Choose Streaming Method:** Decide whether to use **Sockets or LSL** primarily. Each has pros:
   - *Sockets:* Already implemented on PC (AndroidStreamReceiver) and on Android (via NetworkingService in either Wi-Fi or BT mode). This could allow a master phone to send all data to PC in real-time, which PC then logs or displays. It's straightforward but requires the devices be on the same network or have direct connection.

    - *LSL:* More flexible for research; PC just listens for streams. But we need to package liblsl in Android for a full solution (which might be an effort).
10. For immediate integration, **use sockets with Wi-Fi**: e.g., have the master phone, when in master mode, also open a TCP client to the PC's IP (if known). Or simpler: run the phone in *slave* mode connecting to PC as master (like PC is a server). The PC's `AndroidStreamReceiver` already acts as a server on port 8080 [123]. We can leverage that:
    - Implement in NetworkingManager or a new small routine: a **PC connect mode** that opens a TCP socket to a configurable IP (the PC's). The UI can provide an input for PC IP, or temporarily one can hardcode it for demo if IP is known. This can reuse the JSON DataEvent serialization (the DataStreamer is actually already structured to send to multiple clients via `clientSender` callback [124] [125]). We can treat the PC as another "client" of the DataStreamer: i.e., when PC connection is established, add it to the DataStreamer's clients and then all events will be `invoke()`d to that socket.
    - Alternatively, use the existing NetworkingService but that's designed for phone-to-phone. Perhaps simpler is to just create a minimal `PCClient` class inside the app that connects to PC IP:port and sends events outside of the phone-to-phone NetworkingManager. This isolates PC comms and won't interfere with phone sync logic.
11. **Align Data Format:** Fix the earlier mentioned inconsistency (send `"gsr"` instead of `"gsr_data"` or update PC to accept both). Also ensure any new event types (if we added heart_rate or others) are either hidden from PC or also supported. Ideally, PC should be updated to mirror final Android event schema to avoid data loss.
12. **PC Coordination:** If using sockets, ensure the PC knows when a session starts and stops. That's done via sending an `"event": "session_start"` message at the beginning (the RemoteControlService on Android possibly already sends commands to start PC recording). If not, implement that: when user hits start on phone, also send a control message to PC to start its logging. PC already listens for `session_start` events and marks them [126]. On stop, do similarly. This way PC can create a new file per session and not just stream into one long log.
13. **Testing with PC:** Do a test run where you start the phone, connect to PC, then record something. Check on PC:
    - Does it receive data from all modalities (there should be callbacks printing or plotting data, e.g., PC code has `_handle_rgb_frame()` stub and others [127] – perhaps extend PC code to actually save images or at least count them).
    - Measure latency: perhaps log on PC and phone a sync marker and see difference. It likely will be small (<100 ms).
    - Test with multi-phone: PC should get data from both (the connected master might forward slave data as its own, or maybe each phone connects separately to PC if on same network). Decide architecture: maybe easier if **only master connects to PC** and it relays other phones' data (so PC sees one stream of unified data). Or each phone could separately connect to PC; then PC would treat them as separate sources with their own timestamps. The latter could complicate analysis because you then need to merge logs with offsets; the former centralizes merging on master phone. Given the NetworkingManager design, centralizing on master seems intended (master collects and could send out combined events).

14. If time permits and it's beneficial, integrate **LSL properly**: compile or include `liblsl.so` for Android (there exist builds for arm64). Then test that `LSLStreamingService.initialize()` returns true [84]. If working, then the phone would broadcast to any PC running LabRecorder or custom LSL script. The PC app in fact had config for `lsl_enabled: true` [128], meaning the PC can take advantage of that. LSL would automatically handle time alignment; it might even be simpler ultimately. But doing this requires NDK steps – if not comfortable, skip to avoid delaying the project.

15. **Improve Data Management for Long Sessions:** In multi-device or PC connected mode, consider the following:

16. **File management:** If multiple devices record, do we want one combined dataset or separate per device? It might be easier to keep separate (each phone writes its own CSVs). But for analysis, merging needed. If Phase 3 aims for a polished research tool, maybe have the master phone collect and write one set of files including all devices' data (with device ID tags in each line or separate columns). This could be done by modifying DataWriter to include deviceId on each entry when in network mode, or by having separate CSV files for remote data (like "gsr_device2.csv"). This is an architectural decision: central vs distributed logging. A compromise: each phone logs its own, and the PC can later merge by timestamp using the sync offsets. Given time, implementing central logging on master is cleaner but complex.
17. **Scalability:** How many devices do we intend to support? If more than 2, test that scenario. Wi-Fi Direct supports multiple clients in a group, but bandwidth and phone CPU could limit if say 3 phones all send video to one master. Possibly restrict to 2 in practice.
18. **User Interface for Networking:** Provide clear information in the app when in multi-device mode: e.g., show "Master – connected to 1 device" or "Slave – connected to Master X". Provide a way to disconnect gracefully. Possibly add a notification when network is active (since these can run in background, a foreground service might be warranted for long captures to avoid Android killing the process – but for now, assume app stays foreground).
19. **Error cases:** If the PC disconnects mid-session (Wi-Fi drop), ensure the phone doesn't crash. Maybe it should attempt reconnect or at least continue local recording so data isn't lost. Similarly, if a slave phone goes out of range, master should log an error but continue its own recording.

Justification: Phase 3 elevates the system from a single-device demo to a **full research platform** capable of multi-device studies and integration with existing lab tools. This is likely important for the project's ultimate use (collecting data from multiple angles or syncing with PC-based sensors). Doing it after stabilizing one device is logical because networking adds complexity and potential issues. By now tackling those, we can make sure that: - The **multi-phone synchronization** actually achieves the intended outcome: recording different modalities on separate phones but still aligned in time. This unlocks scenarios like one phone focusing on face thermal imaging while another tracks hand GSR, etc., broadening research possibilities. - The **PC integration** allows leveraging the phone as part of a larger sensor network. Many researchers use PCs to aggregate data; by streaming to PC (especially via LSL which many are familiar with), we improve adoption of the system. It also allows heavier processing (like real-time analysis or visualization) on the PC that a phone might not handle. - We address any **latency and accuracy** concerns by fine-tuning sync on these broader setups. Phase 3 might involve more iterative testing and potentially adjusting the synchronization algorithm, but this ensures the system is not only feature-rich but also scientifically reliable (so that data from different devices can be trusted to correlate).

There are trade-offs in Phase 3: focusing on networking might deprioritize some phone-only enhancements. It's assumed that by now, core performance is good enough that splitting focus to networking is acceptable. We must be careful not to re-introduce instability; thus, thorough testing here is needed. This phase likely yields a **milestone version 1.0** of the system that meets both demo needs and the full project requirements.

Phase 4: User Experience and Maintainability Improvements

Objective: After core functionality and networking are solid, Phase 4 invests in the overall quality of the application, making it easier to use and maintain in the long run. This includes refining the user interface (for better usability during experiments), improving code maintainability (documentation, tests), and addressing any technical debt or minor features postponed from earlier phases (like the hybrid view or advanced settings).

Key Tasks:

1. **Refine UI & Workflow:**
2. **Settings Integration:** Now implement the Settings screen properly. Wire up all user-selectable options:
    - Participant ID: ensure it's used in session metadata (the CsvDataWriter already accepts participantId for directory naming [129] , we just need to pass it in when calling `startNewSession`).
    - Recording mode (local vs network): if user chooses "Network only" vs "Local & Network", let that influence whether data is saved on phone or only streamed.
    - Resolution, sample rates: Make the RGBCaptureModule adjustable to 720p/1080p etc. The Camera2 API supports setting size; implement that logic based on settings. For GSR, allow selecting a lower rate if desired (Shimmer might allow it, but if not, we could downsample in software).
    - Preview mode default: if user selects "Thermal" as default, launch the app with thermal view on.
    - Network type and role: possibly incorporate from Phase 3 here so that going into settings you can pre-configure multi-device roles rather than fiddling in UI each time.
3. **Hybrid Thermal-RGB View:** If still desired, implement the hybrid preview: overlay the thermal image on the RGB view (maybe semi-transparent or using a small corner PiP view). This can be useful to correlate, for example, visible perspiration with thermal changes. Implementation might involve converting thermal frame to a Bitmap (as done in Phase 1) and drawing it on a Canvas that also has the RGB frame (which you can get via TextureView's bitmap or by using two TextureViews with blending). Ensure synchronization – both frames have timestamps; perhaps display the latest of each if within some small time window.

4. **In-app Tutorial or Guide:** For a more robust tool, consider adding some help text or a quick start guide in the app (maybe an "About" dialog explaining how to connect devices, or what each permission is for). While not crucial for a demo where you narrate it, it's helpful for maintainers or new users later.

5. **Code Maintainability:**

6. **Documentation:** Write a developer README or inline code comments for complex parts (e.g., explain how sync offset is computed in code, or how data format is structured). This helps future contributors or if this project is handed over to another student.
7. **Unit Tests:** If possible, add unit tests for critical components. For example, test the `SynchronizationManager.createTimestampedEvent()` logic to ensure it correctly stamps events, or test the `CsvDataWriter` formatting of a known DataEvent. There are already some test stubs in the repo (saw `RecordingControllerTest.kt`, etc.) – fill those out. Particularly test scenario logic like "if one module fails to start, does RecordingController stop all and report error gracefully?"
8. **Modularization for reuse:** Consider if any components can be made into libraries or easier to update. For instance, the Shimmer integration could be encapsulated so that if a new sensor (like Empatica E4 wristband) needs to be added, one could implement another CaptureModule easily. The architecture already supports that; just verify it by hypothetical – maybe add a dummy

"AccelerometerCaptureModule" that reads phone accelerometer, just to see that the pipeline accepts new modules without major changes. This is future-proofing.

9. **Optimize Resource Handling:** Revisit resource clean-up code. Ensure every `start()` has a matching `stop()` and `release()`. E.g., what if user exits the app without stopping recording? Use Android lifecycle (onPause/onDestroy) to handle that (which in MainActivity we saw they do call release on destroy [130]). Possibly use a Service for recording so it can survive UI rotation or background – but that's a larger design, maybe not needed if we assume app stays active. Document these assumptions.

10. **Known Issues Resolution:**

11. Address any minor bugs or user feedback collected from demo and testing phases. For example, if testers noted that "the app sometimes freezes at stop for a few seconds", investigate and fix (maybe writing large CSV buffers to disk on stop, solution could be to flush gradually during recording rather than all at stop).
12. If the app currently requires any workaround (like manually enabling GPS because BLE needs it), consider prompting the user properly ("Please enable Location for BLE scan"). The ConfigurationValidator might handle some of these checks – extend it if needed to cover more (e.g., notify if USB host not supported or if Wi-Fi Direct not supported).

13. Ensure compatibility with a range of devices: test on at least one other Android model. Sometimes camera or USB APIs behave differently (for example, Samsung devices might have different permission quirks). If issues found, handle via conditional code or update documentation that "tested on X, Y devices".

14. **Performance Tuning (revisit after enhancements):**

15. Now that new features are in, do another performance pass. Check CPU usage with multi-device and PC connected. If anything spikes, consider offloading (for instance, heavy JSON serialization could be moved to a background thread explicitly – though kotlinx is already async since we send via channel).
16. Investigate using more efficient data formats for internal pipeline if needed. For example, consider using **binary** for some internal exchange instead of JSON to reduce overhead (the DataStreamer currently JSON-encodes every event for network; using MessagePack or Protocol Buffers could cut size, but requires changes on PC too – maybe a future idea).
17. Profile energy consumption if the app will be used in long field studies. If battery life is an issue, a future step might be to allow pausing certain modalities (e.g., turn off RGB camera if only thermal is needed, to save power). Or using the phones in battery pack cases. These aren't immediate, but put on the radar.

Justification: Phase 4 is about refining the app into a user-friendly, reliable software that can be maintained and extended beyond the immediate project. At this point, presumably the demo or initial deployment is done, so we can invest in things that weren't critical for a one-time demo but are very important for continuous use: - A well-designed **UI and workflow** ensures that researchers using the app in experiments can do so without confusion or error. For example, having all settings working means they can customize to their needs (higher resolution if their phone can handle it, or lower if they need longer battery). - **Maintainability** is crucial if this project will be built upon by others or used for multiple studies. By cleaning up the code, adding documentation, and possibly tests, we reduce the risk of regressions and make future upgrades (like adding a new sensor or switching to a different thermal camera model) easier. - This phase also addresses any loose ends (like the hybrid view which was initially not done, or minor bug fixes). It's effectively a **quality assurance and polish** phase.

Trade-offs here involve spending time on polish versus new features. We assume that by Phase 4, major features are done, so focusing on polish is warranted. It might not directly add new capabilities (except maybe the hybrid view), but it ensures the system is robust and pleasant to use, which is important for adoption and longevity.

Phase 5: Extended Capabilities and Research Extensions

Objective: In the final proposed phase, we explore additions that push the app beyond its initial scope, potentially opening new research opportunities or product features. These might be lower priority or experimental, but are worth planning for if time/resources allow or for future developers to consider.

Key Possible Extensions:

1. **Real-Time Stress/Emotion Inference:** Incorporate an on-device algorithm that computes an indicator of stress or arousal from the sensor inputs. For instance, using the GSR and heart rate data to compute features like phasic driver or heart rate variability, and maybe thermal data (glabellar skin temperature changes) to estimate stress level. This could be shown on the UI (like a gauge or color that changes). It would demonstrate the ultimate goal (if the project is about estimating stress from RGBT data). Implementing this might involve a simple machine learning model or heuristic:
2. Perhaps train a model offline (using collected data) and then load it into the app to run on each new data point. TensorFlow Lite could be used if a ML model is involved.
3. Or a simpler threshold rule: e.g., if GSR increases by > X and temp increases by Y, classify as "High Stress".
4. This feature is clearly long-term and research-intensive, so it should only be attempted once the basics are rock solid. It can also be continuously improved as more data is collected.
5. The trade-off is CPU usage vs benefit; a small overhead model is fine, but anything heavy could hurt performance. So keeping it lightweight or only updating say every 5 seconds is prudent.
6. **Cloud Connectivity and Data Management:** Expand the system to not just connect to a local PC, but optionally upload data to a cloud server or database for remote monitoring. This could be valuable for experiments where researchers want to monitor participants in real-time from a different location. For example, integrate a module to send data to an MQTT broker or a Firebase service. Given security and complexity, this is beyond immediate needs, but it's a direction (this essentially generalizes PC streaming to internet streaming).
7. This would require careful handling of data rates (maybe only sending summary data or lower frame rate to cloud).
8. Also encryption if sensitive data is transmitted outside local networks.
9. **Support for Additional Sensors:** Perhaps add other sensing capabilities:
10. **Phone Accelerometer/Gyro** – to correlate physiological changes with motion artifacts (could be useful to filter out movement-caused GSR spikes or to know the orientation for thermal).
11. **Ambient sensors** – like temperature/humidity from phone or external modules, to calibrate thermal readings.
12. **Alternate cameras** – if a higher resolution or different spectrum camera becomes available, integrate that as a new module (the modular architecture makes this feasible).

13. This extension is about making the platform more versatile. Each added sensor will need corresponding DataEvent type and UI indication. It's a moderate effort each but straightforward given the patterns established.

14. **Enhanced Data Visualization and Export:**

15. Build in the ability for the app to generate a quick report or visualization after a session. For instance, after stopping recording, the app could plot GSR vs time or show a timeline of synced data (maybe using MPAndroidChart library to draw graphs of the CSV data on the phone). While researchers often do analysis offline on PC, an immediate visual on phone can validate data quality on-site.
16. Provide more export options: maybe zip all the CSVs and make it easy to share via email or upload to cloud from the phone. This makes it user-friendly to get the data off the device if not using PC in real-time.
17. Implement a **replay mode** in the app: allow loading a recorded session and playing back synchronized data (like a mini viewer). This is a complex but interesting feature for demonstration and debugging.

Justification: Phase 5 is largely about future-proofing and exploring how to leverage the solid foundation built in earlier phases. By adding intelligence (stress inference) and broader connectivity (cloud), the system could move from a data collection tool to a real monitoring solution. These are not required for baseline functionality, hence they come last, but they align with possible future research questions or product development (e.g., creating a mobile app that not only records but also *interprets* physiological signals on the fly).

Implementing these would likely require collaboration with domain experts (for stress algorithms) and careful testing to ensure they don't conflict with real-time performance. They should be approached once the core recording and syncing system has been validated thoroughly, as they introduce new complexities.

Each phase above is numbered and prioritized to first **get the basics working (Phase 1)**, then **make the app polished for demonstration (Phase 2)**, then **enable advanced use cases like multi-device and PC integration (Phase 3)**, followed by **improving usability and code quality (Phase 4)**, and finally **stretch goals (Phase 5)**. This phased approach ensures that at any point, the project can deliver value (after Phase 1 it's already functional for single-device recording; after Phase 2 it's demo-ready; after Phase 3 it's full-featured for network use).

Throughout these upgrades, it will be important to maintain the synchronization accuracy and reliability of data – those should not be sacrificed for new features. The implementation guidance given for each phase should help in making design choices that align with that principle (e.g., ensuring background threads, minimal overhead for real-time parts, etc.). By the end of Phase 4, the application should be stable and maintainable enough that new contributors can easily extend it in Phase 5 and beyond, and the system could potentially be published for other researchers to use confidently.

3. Comparative Analysis with Existing Platforms

It's important to contextualize the FYP-GSR-Android application in the landscape of multimodal sensor capture systems, both academic and commercial. This comparative analysis will highlight how our app differs from or improves upon existing solutions, as well as acknowledge areas where others might have advantages. We consider a few categories of existing platforms:

1. Academic Research Tools (PC-based e.g. LabRecorder, OpenSignals):

- *LabStreamingLayer (LSL) with LabRecorder:* This is a popular open-source framework for synchronizing data streams from multiple sensors on a PC. LabRecorder on a computer can collect streams from various devices (EEG, GSR, eye trackers, etc.) with sub-millisecond sync, but it requires those devices to be connected to the PC (often wired or via separate wireless receivers). Compared to FYP-GSR-Android, LSL+LabRecorder is **less portable** – you usually need a laptop and all sensors around it. Our app, in contrast, brings the data collection onto a **single mobile device**, allowing field experiments without tethering to a PC [131] [132]. However, LSL does have very strong sync accuracy since everything funnels into one process. Our software-based sync on a phone is within ~50 ms, which is quite good for mobile, but not as tight as hardware sync or a single-clock PC system. The **trade-off** is acceptable given the mobility we gain. In fact, we incorporate LSL support to get the best of both worlds: we can stream to LabRecorder if needed, effectively turning our phone into a wireless LSL node. This hybrid approach isn't something pre-existing tools offered out-of-the-box; we added it because we recognized the value of LSL's ecosystem. - *OpenSignals (Plux) or Shimmer's Consensys software:* These are PC or tablet-based tools from sensor manufacturers which record data from their specific sensors (GSR, ECG, etc.) and sometimes video. For example, Shimmer's own software can record GSR and perhaps you can manually sync it with video by clapping etc. Compared to these, our app's **uniqueness** is integrating *third-party hardware (thermal camera)* with *physiological sensor* and phone's own sensors seamlessly. Manufacturer tools often handle only their device's data, requiring the researcher to later combine video from another source. We eliminate that manual sync step by doing it in software. Moreover, our app is **tailored for research-grade synchronization** (common timestamp for all modalities) which is not typically present in simple recording apps.

2. Commercial Physiological Monitoring Devices (mobile-focused):

- *Empatica E4 wristband & App:* Empatica offers a wristband that measures GSR, temperature, motion, etc., and has a mobile app to stream or record data. However, it does not integrate any camera data. If one wanted thermal or facial video, they'd need a separate camera and again sync externally. Our solution covers that gap by including cameras. In terms of GSR data quality, Empatica is comparable to Shimmer's, but Empatica's advantage is a very polished app and cloud system for its specific device. Our advantage is **flexibility** – we can work with different cameras and sensors (not tied to one vendor) and we store data locally in an open format (CSV + potential JSON), making it easier to do custom analyses. Empatica's app is mostly a viewer/logger; it doesn't allow user extensions or adding new sensors. - *FLIR One / FLIR Thermal SDK:* FLIR (now Teledyne FLIR) provides a mobile thermal camera (FLIR One) and an SDK, plus their own app for viewing thermal images. The FLIR One app can capture thermal video or images and even has some features like spot temperature measurement. However, it again doesn't support **physiological signals** or GSR. If a researcher tried to use a FLIR camera and a separate GSR device with a phone, they'd have to run two apps and attempt to align later. Our app's **unique selling point** is that it **combines** these streams in real time with a single button press. Also, FLIR's app might not provide the raw data access or synchronization we need – it's more a consumer tool. In contrast, we expose raw data (like actual temperature values in ThermalFrameEvent) and time-align it with other modalities, which is valuable for quantitative analysis rather than just visualization. - *Biometric smartwatches / Wearables:* Devices like Garmin watches, Apple Watch, etc., measure heart rate and sometimes stress (via HRV), but their GSR capability is either non-existent or not open. And they definitely do not incorporate thermal imaging. They are highly portable and user-friendly, but black-box in terms of algorithms and with limited sensor range. Our platform, being open-source and customizable, allows researchers to use high fidelity sensors (Shimmer GSR is research-grade, thermal camera provides detailed thermal maps) and get data that's not pre-filtered or locked. The trade-off is our solution is bulkier (phone + attachments vs a sleek watch) and requires more setup, but for research that's acceptable if it yields better data.

3. Prior Academic Projects on Multimodal Stress Monitoring:
We also compare to any literature examples: - Some research setups use a stationary thermal camera (e.g., FLIR A65) pointed at a subject and a separate GSR sensor on hand, all synced via a trigger or just by timestamping on a PC. Our system was inspired to miniaturize that into a mobile form [133]. By doing so, we trade some **sensor quality** for **mobility** [24]. For instance, the FLIR A65 has much higher resolution and sensitivity than the Topdon, and a benchtop GSR can be more finely calibrated than a wearable. But those require a controlled lab environment. Our app allows data collection in the field (e.g., outdoors, or in a car, etc.), enabling more ecologically valid studies. This is a clear advantage where context matters. It's a trade-off: if a study needs very fine thermal details, they'd stick to the lab setup; if the study needs real-world data (like monitoring a person during public speaking on stage), our solution shines. - Another related system is the combination of regular video and physiological signals for affect recognition (e.g., some works use RGB video to estimate heart rate or breathing). There are smartphone apps that can get HR from camera (using PPG at fingertip or face color changes), but integrating that with GSR and thermal is novel. We effectively cover PPG via Shimmer and could also derive HR from either PPG or possibly thermal (face blood flow can sometimes be measured with thermal too). No existing app in literature was found that does **RGB, thermal, GSR, audio together on mobile** – this integration is a key **novelty**. We built on known techniques (like software timestamp sync and multi-threading) but applied them in a new combination of modalities.

Trade-offs and Uniqueness:

- **Synchronization vs. Sensor Quality:** Some existing solutions achieve better sync by hardware means (like using a common trigger or sampling clock). We do it in software; as long as we stay within our ~50 ms target, for many applications (like stress which has slower dynamics) this is sufficient. Our tests with a finger tap across streams confirm sub-50ms alignment [108]. So we believe we've balanced complexity and performance well for mobile. Achieving this on Android is non-trivial (due to unpredictable scheduling), and that is an innovation of our project – careful use of a single clock reference and start triggers. Competing mobile apps often ignore sync (they just assume phone timestamp is fine per sensor but don't align streams).

- **Modularity and Extensibility:** Unlike proprietary apps, our architecture is modular. If tomorrow a new sensor (say a galvanic skin imaging camera or a different wearable) needs to be added, it can be done by implementing the CaptureModule interface. This is a strength not found in many closed systems. In academic terms, this app can serve as a **platform** for various experiments, not just one specific study. Many existing tools are one-offs or specific to a study and hard to repurpose – we tried to make ours general.

- **Portability and Cost:** Using off-the-shelf components (an Android phone, a ~$300 thermal camera, a Shimmer sensor) is relatively low-cost compared to some lab setups that involve medical equipment or high-end cameras. There are commercial systems for stress monitoring (some vehicles have driver monitoring systems with thermal cameras, etc.), but those are expensive and not flexible. Our system provides similar capabilities at a fraction of cost and with source code availability. This could democratize this research toolset to other labs.

- **Data Ownership and Privacy:** One aspect to compare is that commercial apps often upload data to cloud or lock it in their format. Our app stores data locally in human-readable form, putting the researcher in control. This is a plus for academic users concerned about data privacy (especially with sensitive data like physiological responses).

In summary, the FYP-GSR-Android app appears to be **unique in combining** a thermal camera, RGB camera, GSR sensor, and microphone in a synchronized fashion on a mobile device. It fills a niche between high-end lab systems (which are accurate but not portable) and wearable gadgets (which are portable but not multi-modal or open). Its multi-device sync capability also sets it apart: even if one tried to use several wearables and phones, syncing them would be hard without our networking approach. The closest alternatives might be custom rigs reported in research papers, but those often require custom hardware or aren't publicly available tools. Thus, our platform stands to be quite valuable for the research community.

One must acknowledge, however, that with this breadth of features come challenges: - Complexity: Setting up all these components correctly means our learning curve might be steeper than a single-purpose app (like Empatica's, which you just wear and press start). We mitigate this by improving UI/UX in Phase 4 as discussed. - Potential Points of Failure: More sensors means more things can go wrong (battery dying, connections dropping). Competing solutions that are simpler have fewer failure modes. We have tried to address this with robust error handling (auto-reconnect, fallback to simulation), which not all systems have.

Overall, in the comparison, our system is **most similar to a PC-based multimodal lab, but miniaturized**, and **most dissimilar to single-sensor mobile apps** that don't attempt synchronization. We consider that combination an innovation, providing researchers a new level of flexibility.

4. Performance and Metrics Expectations

Ensuring the system meets performance requirements is crucial for its usability in both demo scenarios and real experiments. We outline the expected performance metrics and how the system measures up, as well as recommendations for profiling and improving performance.

Target Frame Rates and Data Rates: - **Thermal Camera:** Target ~25 frames per second (fps). The Topdon TC001 hardware supports 25-30 Hz [1], so the app should aim to capture every frame it provides. In practice, slight variations may occur; we expect a consistent ~25 fps with <5% frames dropped over a session. At 25 fps with a 256x192 frame (~49k pixels) of 16-bit data, that's about 1.2 MB/s if raw. However, we currently don't transmit all that raw data off-device in real-time; locally, it's handled in memory and optionally written as metadata (the raw frames aren't stored, only stats or perhaps if saving to a binary file). So 25 fps is achievable given the resolution – the phone's CPU can handle generating a bitmap for each and timestamping. - **RGB Camera:** Target 30 fps at 1080p. Modern phones can capture 1080p30 easily, especially using the hardware camera pipeline (which we do). Our module likely captures from a camera preview stream; if we don't do heavy processing on each frame (which we currently do not, aside from logging timestamp and perhaps format), it should maintain full frame rate. Data-wise, 1080p frames at 30 fps uncompressed is huge (~190 MB/s if raw YUV), but again we do not move that off device fully. If writing to disk, one would normally compress to MP4. We are just logging metadata or streaming possibly compressed frames over network (maybe not implemented yet). The system as is should comfortably handle camera preview and some encoding if needed (phones can encode 1080p video to MP4 in hardware with negligible CPU). - **GSR Sensor:** Target 128 Hz sampling. That is 128 samples per second, which is trivial in terms of data volume (each sample a few bytes). The challenge is latency – BLE packets might come in bursts. But 128 Hz is a requirement to capture quick skin conductance responses. We expect the app to capture essentially 100% of GSR samples (no loss) because the throughput is low. Tolerance for data loss here is near-zero for analysis integrity. Our BLE integration (once implemented) should buffer and send each sample to SyncManager. We must ensure no bottleneck (like writing to disk every sample individually could slow it, but we buffer in memory and CSV flushes in batches, so it should be fine). - **Audio:** If recording at 44.1 kHz stereo, that's quite high data rate (about 176 KB/s PCM). Our default is 16 kHz mono (16-bit), which is ~32 KB/s. The AudioRecord is capable of continuous capture; saving to WAV or writing segments to CSV is okay. The app's design captures in 100 ms segments (so 1600 samples per chunk at 16 kHz). As long as the device can write 10 segments a second (which is trivial I/O), audio won't drop. We expect zero audio dropouts in ideal conditions. Latency between audio and video is expected to be <±50 ms given same clock and immediate timestamping.

Synchronization Latency Goals: - The software sync method targets sub-50 ms alignment [108]. In practice, we'd like to measure the actual offset distribution. For demo and general use, 50 ms is fine (that's around 1-2 video frames). For example, if a sudden event (like a clap) happens, it might appear in thermal and RGB within the same frame or one frame apart, and GSR typically has a slower response (physiologically, GSR may lag a stimulus by 1-3 seconds anyway). The important point is that all data share a timeline, so when analyzing, one can align them easily. - **Inter-device sync latency:** If using multi-device (Phase 3), our aim is to get devices within ~10–20 ms of each other after sync calibration. There will be some network jitter, but since Wi-Fi Direct is P2P and likely <10 ms latency, and we can measure offsets, 20 ms offset is achievable. Even if offset were 50 ms, that's usually acceptable. For reference, professional motion capture allows maybe 10 ms; physiological signals often have inherent delays (heart rate from PPG vs instantaneous ECG differs by ~200 ms due to pulse transit time, etc.), so small offsets may be negligible compared to those factors. - We might incorporate a method to measure achieved sync: e.g., log a sync marker on master and simultaneously make slave create one on its timeline, then compare. This can be done to verify after development. Our expectation is within one video frame period difference.

Data Loss Tolerance: - **Thermal/RGB Frames:** Ideally zero frame drops for short recordings. If recording is long (minutes) and minor drops occur due to GC or something, it's not catastrophic as long as it's rare (and maybe indicated). We flush or drop frames if needed to not backlog (as discussed in Phase 2). For analysis, missing one frame out of 300 is not a big issue if it's documented. But missing many would break sync trends (like if thermal is missing a second of data, you can't correlate with that second of GSR properly). So our tolerance is very low (<1% frames dropped). The current design of separate threads and minimal per-frame work suggests we can meet that. The README proudly said "No frame drops in typical 15-minute sessions" [107] – that is an ambitious claim, but we strive for it. Achieving that likely assumes plenty of device resources and maybe saving video via efficient means. We will test and if minor frame loss appears, we might note it, but try to optimize to approach zero. - **GSR Samples:** Should aim for 0% drop until maybe out-of-range (if BLE disconnects, that's different). BLE might buffer if app doesn't read fast enough, but since we handle events quickly in a callback, we should get them all. Data loss in GSR would directly translate to gaps in a time-series which could affect signal processing (like missing a peak). - **Audio:** Minor audio dropouts are more audible than visible in data; we want to avoid them. If writing to WAV, usually AudioRecord will block if system is overloaded, which can cause stutter. Our approach of chunking plus ensuring flush in background should avoid blocking the recording thread. Tolerance for audio drop is low if audio is important (e.g., analyzing spoken stress markers). If audio is just supplemental, a tiny gap won't ruin analysis. Ideally, none is lost though.

CPU and Memory Usage: - We expect high CPU usage given multiple threads: - Thermal frame processing (some math each frame), - RGB camera (which on modern devices offloads JPEG/encoding to hardware, but if we don't encode it's just feeding frames – not too heavy), - BLE handling (negligible CPU), - Audio encoding (some CPU but not heavy for PCM). - Additionally, if streaming or encoding video to file, that could use hardware codec which is fine. In tests, likely CPU usage might be around 50-70% on a mid-range phone when all sensors active. This is okay but we want to avoid hitting 100% to prevent thermal throttling. - Memory usage: Each thermal frame ~100KB, each RGB frame maybe ~2-3MB (YUV 1080p). If we don't store them, only buffers of that size exist and reused. The main memory usage is buffering network messages (channel size * avg message size) and any cached data. The CsvDataWriter buffers are small (8KB flush buffers). So baseline memory might be <100 MB usage. That's safe for most phones. We must avoid leaks – which testing with Android Profiler can check (e.g., ensure after recording and stopping, no large objects linger). - **Battery consumption:** Running all these sensors and CPU means heavy battery drain. ~2 hours was estimated for continuous full usage [62]. That was likely with a fairly modern phone with a large battery. In practice, the thermal camera draws power from the phone as well, and BLE sensor slightly. So 2 hours is plausible but one should have a power bank for longer experiments. It's an acceptable trade given the complexity. If needed, one could disable the phone's screen (like dim it) to save some power if not needed for preview, or close any unnecessary apps.

Benchmarks and Profiling: - We should conduct profiling on a test session of say 10 minutes: - Use Android Studio Profiler or simple log-based stats from PerformanceMonitor. The app has a `PerformanceMonitor` which likely tracks frame rates and sample counts and could log them or update UI. According to code, it probably measures things like CPU load. Using that, we can gather metrics: average FPS achieved, total events captured, etc. The README states some performance stats which presumably came from such monitoring (like frame rates sustained, memory bounded by queues) [134]. - Specifically measure the **time-stamping accuracy**: We can do a test where a known hardware event triggers multiple sensors and compare timestamps. For instance, shine a laser that is visible in both RGB and Thermal and hits a photodiode whose signal triggers a GSR channel (if MacGyvering a bit). But simpler: record the system clock difference from start between modalities. The SynchronizationManager might be logging that "frames processed, FPS, etc.". If we suspect drift or offset, we can adjust approach (like if we found one thread systematically lags, we might take timestamp earlier in pipeline). - **Profiling Tools:** We recommend using: - **Systrace** (to capture system-level timeline of threads) – can see if frame callbacks are being scheduled regularly. - **Android Profiler (CPU)** – to ensure no single thread is pegging unexpectedly, identify any function that is taking too long (e.g., if analyzeThermalFrame (min/max computation) is too slow, it could be optimized by using native code or doing less often). - **Memory Profiler** – track if memory usage grows over time. If it does, find what allocations are rising; possibly a sign of a leak (like not clearing a list of events or not closing file descriptors). - After profiling, set **performance budgets**: - E.g., each thermal frame processing must complete in <40ms to maintain 25 fps (which is a 40ms period). If it's borderline, consider offloading heavy parts to C++ or using RenderScript (deprecated though) or simpler: reduce tasks per frame. - GSR processing is trivial so no issues, audio runs on its own buffer which is small. - The UI thread should remain responsive (<16ms per frame for 60Hz UI). We do minimal work on UI thread, mainly updating text, so that should be fine. Just ensure not to do big writes or heavy calculations on UI thread.

Expected Outcomes: - With the improvements outlined in earlier phases, we expect: - **Reliable operation for sessions of at least 15-30 minutes** without crash or severe slowdown. (Even if battery might not last much beyond 2 hours, the app should handle long runs in principle, in case connected to power.) - **Time-sync accuracy** on the order of a few tens of milliseconds within one device, and maybe ~<100ms across devices if networked (which is likely acceptable for physiological correlation). - **Resource usage**: CPU might be high but should not thermal throttle within typical session length. Monitoring device temperature is wise; we could suggest to do intensive tasks (like possibly not turning screen brightness high or kill extraneous services while recording). - **Storage usage**: around 1-2 GB per hour as said [62], likely dominated by video if we record it. Currently, CSV files alone for an hour might be maybe 100 MB (because if no raw frames saved, just text logs of indices and such). If we add video recording, that would produce MP4 maybe 0.5-1 GB/hr (depending on resolution and compression). So that aligns with their estimate. Ensure the external storage has enough space for intended capture lengths, or allow writing to SD card if available. - **Network performance**: If streaming, we expect to handle it on a standard Wi-Fi direct link or Wi-Fi network. Thermal+RGB video uncompressed would overwhelm most networks; we are likely not sending raw frames, maybe only for preview or low-quality images. If needed, we could compress frames (JPEG them) before sending to PC. But that is heavy; better to do a more efficient approach (like send only needed data or let PC reconstruct images from some calibration). This is an area where performance trade-off is tricky – either saturate network or do compute to compress. For now, in local mode, we avoid network issues; in PC mode, possibly constrain what is sent (maybe do not send every single RGB frame if not needed, or send at lower rate).

Profiling Recommendations: - Use the built-in logging ("GSRRGBT" tag logs) to monitor frame rates and sample counts during test runs [135]. The app is configured for verbose logs which can be filtered. This provides a quick way to see performance metrics (like "Processed frame 300, FPS: 25.0" log every few seconds [136]). - If encountering dropped frames or buffer issues, try profiling in smaller pieces: e.g., run just thermal and GSR without RGB to see if performance is fine, then add RGB to isolate the bottleneck. - Test on different phones if possible: some phones have better thermal throttling management. This can inform whether we need adaptive features (like if device gets too hot, maybe automatically reduce frame rate of thermal to 15 fps to cool down). - For multi-device, measure sync offset by logging timestamps of a shared event (like both phones log when they receive a sync command from master). The difference in those logs on PC can quantify offset.

In conclusion, the app is expected to meet or exceed the performance needed for accurate multimodal recording: - **Frame rates** at sensor maximums (25–30 fps video), - **Sampling rates** at sensor max (128 Hz GSR, 44.1 kHz audio) with minimal losses, - **Sync accuracy** well within a human perceptual threshold (no noticeable AV offset, and good enough for physiological analysis), - **Resource usage** high but manageable on a modern smartphone (somewhere below the threshold of causing system instability).

We will continuously test and tune these as development proceeds through phases. Having clear metrics and monitoring in place (through logs or a performance overlay) is important to ensure we catch regressions – for example, if adding a new feature in Phase 5 inadvertently slows frame processing, we'd see the FPS drop in logs and can address it. By maintaining this vigilance, we aim to keep the app's performance optimized for both real-time user feedback and the fidelity of recorded data.

5. Recommendations and Enhancements

In light of the above analysis, we summarize the key recommendations to address immediate blockers, as well as medium- and long-term enhancements to improve maintainability and robustness. This section serves as a checklist of things to do (some reiterating phase tasks in brief) and additional suggestions drawn from the audit.

5.1 Immediate Blockers to Address (Short-Term)

These issues should be fixed as soon as possible (ideally before the next demo or testing session) to ensure the app functions as intended:

- **Implement Real GSR Support:** Replace the simulated GSR module with actual Shimmer integration [5]. Without this, the GSR data is not real, defeating the purpose. This involves using the Shimmer BLE API, handling connection in `DevicePairingDialog` (the TODO there [33]), and feeding data to the SyncManager.
- **Fix Missing Microphone Permission:** Add `RECORD_AUDIO` to permission requests and manifest so that audio recording works out of the box [53]. Verify audio capture after this change.

- **Enable Thermal Camera Usage:** Ensure the Topdon SDK is properly included and loaded. Add runtime USB permission request to avoid silent failure. Also implement the frame rendering so the thermal feed is visible (even if just grayscale) – a demo critical fix so we can actually show thermal output.
- **Stabilize Start/Stop Recording:** Test the start/stop lifecycle thoroughly and fix any crashes or deadlocks. For instance, ensure RecordingController's latch mechanism always releases even if one module fails to start (maybe add a timeout or error callback to avoid hanging). The stop sequence should handle each module's exceptions.
- **User Feedback on Errors:** Implement basic notifications for critical errors. E.g., if Shimmer fails to connect, show a Toast "GSR sensor not connected – using simulation." This way the operator knows what's happening and can act (or at least explain to viewers).
- **Update PC Software for Compatibility:** Coordinate with the FYP-GSR-Windows side to fix small compatibility issues (like the `"gsr_data"` vs `"gsr"` type mismatch). This ensures end-to-end data flow. Also test the PC receiving data from the updated phone app once the above changes are made – any issues should be resolved early (like adjusting buffer sizes or message parsing).
- **Prepare Demo Configuration:** If a demo is imminent, create a specific config or mode for it: e.g., if the demo will only use one phone and local recording, maybe disable multi-device code to avoid accidental interference. Pre-load any required pairings (bond the Shimmer to the phone via Bluetooth settings beforehand, etc.). Essentially, remove as much unpredictability as possible.

(Most of these align with Phase 1 tasks in the plan, underlining their importance.)

5.2 Medium-Term Improvements (Next Steps after Core Stability)

Once the immediate blockers are resolved and the system can reliably record data, focus can shift to improving user experience, reliability, and maintainability:

- **Complete the Settings UI and Persistence:** Make sure that user preferences (like chosen resolution, participant ID, default mode) actually affect the app behavior and are saved between sessions. Use `SettingsRepository` to load and apply settings on app start [137]. This prevents issues where a user thinks they set something but it's ignored.
- **UI Polishing:** Refine the main UI layout – for instance, ensure it scales well on different screen sizes, label all indicators clearly (maybe "Thermal: Connected" text, "GSR: 5.2 μS" display, etc.). Add units to values displayed (μS for GSR, BPM for heart rate) to avoid confusion.
- **Implement Heart Rate Display:** If PPG is coming from Shimmer, calculate and show heart rate in real-time. If Shimmer doesn't provide it directly, use a simple peak detection algorithm on PPG data in the GSRCaptureModule. This was touted as a feature in README [116], so delivering it adds value for users interested in cardiovascular responses.
- **Robust Reconnection Logic:** Build on initial connection work to add auto-reconnect:
- For BLE GSR: if connection drops, app should try to reconnect a few times quietly (and update UI status as "Reconnecting…").
- For Thermal camera: if it's unplugged mid-run, maybe pause the thermal module and pop-up a message like "Thermal camera disconnected". If plugged back in, perhaps allow resuming (this might be complex, but at least handle the disconnection gracefully instead of crashing on null).
- For Wi-Fi Direct: if using, auto re-initiate group formation if it fails once, etc. We have some of this logic but test and refine.
- **Logging and Debugging Aids:** Include more detailed logging (possibly to a file) for debug builds. For example, log timestamps and major events to a "debug.log" so if a user reports an issue, developers can inspect what happened internally. This will help maintainers troubleshoot sync issues or sensor errors in the field.

- **Battery Optimization Options:** Provide options or automatic adjustments to extend battery if needed. E.g., an option “Limit camera frame rate to 15 fps to save power” or “Disable audio recording” if audio isn’t needed. This way the app can adapt to different scenarios (some studies might not need audio, etc.). Not enabling these by default, but giving advanced users control is useful.

- **Security and Permissions:** As best practice, ensure the app only requests permissions it absolutely needs and explain why (the system prompts can be supplemented with rationale dialogs if necessary). For multi-device (Bluetooth/Wi-Fi), maybe implement a pairing confirmation if unknown device attempts to connect to avoid any chance of unwanted data injection (though that’s low risk in our usage).

- **Known Issue Mitigations:** Summarize any known issues (some were identified: e.g., “must enable location for BLE” – ensure user is reminded to turn on GPS if off, as scanning will yield nothing otherwise). Document these in an in-app help or the README so users are aware. For instance, in an Appendix of the user guide, list known issues and how to solve (like “If thermal camera not detected, check USB OTG support and permissions”).

These medium-term improvements aim to make the system **robust for regular use**, not just a controlled demo. By implementing reconnection and thorough UI feedback, the app becomes resilient to common problems (wireless hiccups, user mistakes), reducing data loss and user frustration.

5.3 Long-Term Enhancements (Maintainability & Expansion)

Looking further ahead, once the system is stable and in use, the following enhancements are recommended for maintainability and to keep the project modern and extensible:

- **Refactor and Modularize if Necessary:** As features have been bolted on, periodically revisit the code structure to see if any classes have grown too large or if certain responsibilities can be better separated. For example, the RecordingController is quite lengthy (1000+ lines [138] [11]); perhaps factor out networking coordination into a separate class, etc. Use design patterns to keep code clean (e.g., Strategy pattern for different network types, Observer pattern is already used with flows).
- **Upgrade Dependencies and SDK:** Keep the project updated with the latest Android SDK and third-party libraries:
- As of now, target SDK probably around 30; eventually move to 33 or whatever is current, ensuring compatibility changes (like if they deprecate some BLE API, adapt accordingly).
- Topdon SDK: watch for any updated version that might improve performance or compatibility (and update our integration accordingly).
- If using any external libraries like for JSON (we use Kotlinx which is fine) or LSL, update those as needed for bug fixes.
- **Automated Testing:** Develop a suite of instrumentation tests that can simulate sensor input (maybe use the simulation modes to feed known data) and then verify the synchronization logic and data output. For instance, simulate a GSR waveform and a thermal event at known intervals and check if the CSV output aligns times correctly. This can guard against future regressions when modifying code.
- **Platform Independence and Scalability:** Consider how to port or scale the system:
- Perhaps create a version for different Android form factors (tablets, or maybe even an embedded board with Android Things if that was still around, for stationary uses).
- Investigate iOS support if ever needed (though integrating external hardware on iOS would be a new project entirely, so maybe not).

- Or more practically, ensure that if a new sensor or camera comes (like a higher-res thermal camera with a different SDK), it's straightforward to integrate. Our modular approach should allow it, but it may require writing a new CaptureModule variant. Documenting how to do that will help others extend the platform.
- **User Interface Enhancements:** Long term, if the app is used by others frequently, a more sophisticated UI could be warranted:
- Multi-page UI where one screen is live capture, another could review past sessions.
- The ability to label events or insert markers during recording (e.g., press a button to mark "stimulus shown now") which gets logged as SyncMarkerEvent with description. This can greatly aid later analysis. Currently, we have SyncMarkerEvent structure [86] but we might not expose an easy way for the user to generate them except start/stop. Adding a "Mark Event" button or voice command could be an enhancement.
- Accessibility improvements (if any users require them).

- **Extend to Additional Use Cases:** The architecture can allow capturing other modalities (for example, PPG via camera or integrating a respiration belt sensor). As a long-term plan, exploring integration with other wearable sensors or camera-based algorithms can expand the usefulness. Essentially, the app could become a general framework for multimodal data collection on mobile, beyond just GSR-RGB-Thermal. If that's a goal, rename and generalize accordingly, and possibly attract contributions from other developers who have different sensors.

- **Community and Open Source:** If releasing this as an open-source tool, set up a repository with clear contribution guidelines, issue tracker, etc. Over the long term, community feedback could drive enhancements (e.g., support for another brand of thermal camera if someone contributes it). This requires ensuring code is clean, well-documented, and modular – which loops back to above points on maintainability.

In sum, the long-term enhancements revolve around keeping the system **adaptable and high-quality** as technology and user needs evolve. After investing so much into this platform, we want it to remain useful for years, which means accommodating new sensors, new Android versions, and new research demands with relative ease. By continuously refactoring for clarity, testing thoroughly, and documenting, we ensure the app can be handed over to future researchers or developers without losing understanding of its inner workings.

Finally, one more recommendation: maintain a **change log** and thorough **documentation** (possibly an Appendix in documentation listing all classes and their roles, as started below). This will make onboarding new contributors or students much faster and reduce the chance of duplicate work or misusing the system.

By following these recommendations in a timely manner (short-term immediately, medium-term over the next few development cycles, long-term as ongoing practice), the FYP-GSR-Android application will not only meet its current objectives but also stand as a reliable and extensible platform for multimodal data capture in affective computing research.

6. Integration with FYP-GSR-Windows

A crucial aspect of this project is the integration between the Android mobile app and the Windows desktop application (`fyp-gsr-windows`). The two are designed to work in tandem for cases where live data monitoring, larger storage, or additional processing on PC is needed (for example, a researcher viewing the data in real time on a PC while the subject is wearing the devices). This section details how the integration is achieved, covering data formats, transmission protocols, synchronization of time and markers, and testing strategies for the combined system.

6.1 Data Streaming Format Compatibility

To integrate seamlessly, the Android app and Windows app must **speak the same language** in terms of data format: - As described earlier, the Android app serializes each data event as a JSON string with fields including a `type` identifier, a timestamp (in Unix time or another agreed reference), and a data payload (which may be a simple value, or a structured object, depending on event type) [65]. For example:

```
{
  "type": "thermal_frame",
  "timestamp": 1625140800.123,
  "data": {
      "width": 320,
      "height": 240,
      "frame_number": 150,
      "file_path": "frame_150.png"
  }
}
```

(Note: In our current design, we aren't actually sending image bytes over JSON for frames due to size; instead might send a file path or a small representation). - On the Windows side, the `AndroidStreamReceiver` expects exactly this structure. It reads the length-prefixed JSON messages from the socket [66], decodes JSON to a dictionary, and then examines `message['type']` to route it [65] [139]. The code defines handlers for 'gsr', 'rgb_frame', 'thermal_frame', 'sensor_data', and 'event' [140]. Therefore, the Android must send those exact type strings: - We will adjust the Android code to ensure it uses `'gsr'` as the type for GSRDataEvent (instead of 'gsr_data') when streaming to PC. E.g., we might map our internal `GSRDataEvent` to an outgoing message with `"type": "gsr"`. This tiny fix will make the PC's `_handle_gsr_data` get called properly [68]. - Similarly, ensure `'event'` type is used for session start/stop or error events so that PC's `_handle_event_data` triggers [72]. - The other types 'rgb_frame' and 'thermal_frame' we already use matching names [141] [142], so those align. - **Data fields compatibility:** We should check that the PC handler expects fields that we provide. For example, the PC's `_handle_rgb_frame` looks for 'width', 'height', 'format', 'file_path', 'frame_number' in the JSON [69]. Our Android `RGBFrameEvent` includes all of those: width, height, format, frameIndex (which will be serialized as frame_index), and we can include a file path if we save images or video frames on the phone. If currently we are not saving frames, perhaps for streaming we might send one of: - Option 1: `file_path` pointing to an image stored on phone – but PC can't access the phone file system directly. So that's not immediately useful unless we implement a secondary channel to actually send the image bytes or a URL. - Option 2: we could extend the protocol to allow sending frame bytes via UDP (since PC's AndroidReceiver opens a UDP socket on port+1). For example, PC could be receiving large frames via UDP `_process_received_data` for 'thermal_frame' already calls the handler [143], but actually PC's `_handle_thermal_frame` currently just gets meta like width, height, etc. and no actual pixel data [70].

Given the complexity, for now, we likely will **not transmit full image data** to PC except maybe low-rate preview images. The integration will focus on numeric data and metadata. The PC can record those and possibly merge with video recorded separately (like if phone saves an MP4 and then transfers it, the timestamps in CSV vs video can be aligned by markers). - **Timestamp format:** We must ensure both sides interpret timestamps the same way. If Android sends `timestamp` as a floating-point seconds (like Unix epoch or relative), the PC should use it directly. It looks like PC's code, when receiving JSON, does:

```
timestamp = message.get('timestamp', time.time())
```

So if we provide one, it uses it, otherwise it defaults to PC's current time. That means if clock sync is done, we should send our timestamps in a unified reference. A good approach is to send as Unix epoch in seconds (which elapsedRealtimeNanos can be converted to by offsetting with the boot-time reference). Or, simpler, since PC is mostly just logging, we might actually want PC to use its own receipt time for some logs and separately use a sync offset approach for analysis. However, better is to align on one timeline: - Possibly we make both phone and PC use the phone's start-of-recording as time zero. If phone sends a session_start event with its timestamp, the PC's AndroidSyncManager could compute offset = PC_time_at_arrival - phone_start_time and use that to convert all incoming phone timestamps to PC clock domain or to a common epoch. - Because LSL typically uses time-since-start or NTP-synced time, we could emulate similar. For now, a pragmatic solution: use Unix epoch (System.currentTimeMillis) from phone for each event's timestamp. The PC will log those. Even if they differ from PC's local time, they are consistent among themselves. Later analysis can subtract and align if needed. The important part is: **the format is standard (seconds or ms)** and that PC does not confuse it. The PC code treats timestamp as float (likely seconds), and later code might handle differently. We ensure to send a float seconds (since JSON encode of a Kotlin Long nanosec might become scientific notation large number, better to send double in seconds). - **Data completeness:** We should decide which events we stream. Possibly not every single sensor sample needs to be streamed if the PC is also saving (to avoid saturating link). But the PC architecture suggests it's meant to handle all. Given our testing, Wi-Fi Direct can handle the data rates but if not, one might choose to not send raw audio (since that's heavy) or compress it. At least at first, stream all events except maybe audio if not needed in PC live view (the PC doesn't appear to have an audio handler currently). - **JSON overhead vs Binary:** JSON is human-readable but verbose. For high-rate events like GSR at 128 Hz, that's 128 JSON messages per second, which is okay but not super efficient. Possibly combining samples (pack multiple GSR readings into one message) could be more efficient. The PC's code has `_handle_sensor_data` that seems intended for accelerometer/gyro arrays [144]. We could use that concept to batch, but that complicates sync slightly. For now, one message per sample is fine until proven problematic (128 msg/s is fine on Wi-Fi; on Bluetooth classic maybe it's borderline, but if needed we can batch).

Overall, establishing a clear specification: - We will update documentation to clearly define the message format that Android sends and PC expects (like a mini protocol spec). For example: **Message types and payloads:** - `"gsr"`: `{ "value": <float> }` representing microsiemens, - `"rgb_frame"`: `{ "width": <int>, "height": <int>, "format": <string>, "frame_number": <int>, "file_path": <string> (optional) }`, - `"thermal_frame"`: similar to rgb_frame but maybe with `temperature_range` or stats instead of format, - `"audio_data"`: maybe not used yet on PC (we could add later), - `"event"`: `{ "event_type": "session_start/stop/error", "event_data": <details> }` for control events, - etc. Ensuring both sides stick to this spec will avoid integration bugs.

6.2 Socket and LSL Transmission, Synchronization Markers

There are two parallel transmission mechanisms possible: **sockets (TCP/UDP)** and **LSL streams**. The integration plan should clarify how to use them and how synchronization is handled in each:

- **Socket Transmission (TCP for reliability, UDP for high-frequency frames):**
- The PC's AndroidReceiver sets up both a TCP server (for control and general data) and a UDP server for possibly receiving larger data bursts (like video frames) [145] [146]. In our integration:
    - Use **TCP for critical, low-frequency data**: session start/stop events, GSR samples, PPG, maybe even audio segments, and possibly thermal frame *metadata*.
    - Use **UDP for high-bandwidth data**: if we decide to send image frames (like actual pixel arrays or encoded images), send them via UDP to port 8081 (if TCP is 8080). UDP is connectionless, so the phone can just fire packets. The PC's `udp_server_loop` will catch them and feed into `_process_received_data` [147]. Because UDP has no inherent ordering or guaranteed delivery, we should only use it for data that can tolerate a drop (like video frames, where losing one frame just results in a slight glitch, not a showstopper).
    - We need to implement on Android side something like: after capturing an RGB or thermal frame, if networking is active, encode it (maybe compress to JPEG or PNG) and send via UDP socket to the PC's IP and port. Then send the metadata via TCP with the `file_path` or a frame ID that PC can match with a later actual image (this is complex to implement fully for now – likely we skip sending actual images in initial integration).
    - Another approach: since currently PC `_handle_rgb_frame` and `_handle_thermal_frame` just log metadata, we may simply not send the actual pixel data at all. The PC may then only be used for real-time plotting of numeric data (GSR, HR, maybe a framerate count, etc.) and rely on phone to store video. That's acceptable for many research workflows (they often manually sync video and sensor data by timestamps anyway). But it reduces the PC's use as a live monitor of video feed. If a live video on PC is desired, we might consider streaming only one of the two video feeds (say thermal) at lower res to PC as a preview. This could be done by sending JPEGs occasionally (like 1-2 fps thermal images) to PC to display for context, rather than full 25 fps which might be too heavy. This is a design decision to be made based on demo requirements.
    - **Synchronization of clocks via socket:** The `AndroidSyncManager` on PC likely is intended to handle this. Possibly the PC can send a sync request to phone and get a timestamp response to compute offsets. In practice:
    - When phone connects, PC could immediately emit a "time_sync" command. The phone would respond with its current `SystemClock.elapsedRealtime` or a set of sample timestamps.
    - The PC calculates offset = phone_time - PC_time (or vice versa). Then, PC's AndroidSyncManager might adjust how it logs or might simply report the offset for analysis.
    - Because both devices likely have some drift, doing this at start should suffice for a short session (drift of quartz clocks in 15 minutes is negligible, maybe a few ms).
    - If using LSL as well, LSL has built-in clock sync – but if we skip LSL and do direct, we implement our own lightweight sync as above.
    - We saw in PC code `AndroidSyncManager.syncCompleted` signal and storing offsets [88] [148]. That suggests the PC might already have a routine. Possibly when PC receives a `SyncCommand` event from phone (the phone can send a special message with its notion of time), PC sets that as reference. If not implemented, we can easily add: on

session_start, include in event_data the phone's start elapsedRealtime in nanos. PC receives that in _handle_event_data and then can compute difference to its own now. That difference can be our offset.
  - For analysis, as long as we log both phone and PC times for the same event, we can retroactively align.
  - **Markers:** The integration uses synchronization markers as a means to verify timing alignment:
  - The phone's SynchronizationManager can generate a periodic marker (or on specific triggers like user tapping the screen or a known event like "clap"). If these SyncMarkerEvents are transmitted to PC (they likely are via normal data flow with type perhaps "sync_marker" or as part of "event"), the PC can log when it received them. If clock offset is known, it can compare.
  - Markers like session_start and session_stop are explicitly handled on PC to log and maybe start/stop its own recording [149]. We should ensure the phone sends those. For example, when user hits Start on phone, phone should send an 'event': 'session_start'. PC then knows to maybe open its files or at least mark the log. Similarly, on stop.
  - Another marker could be calibration events (say user triggers a sync flash).
  - The system could even use an external event: e.g., an auditory tone or visual flash captured by both phone sensors and PC microphone (if PC has one) – but that's overkill since we can rely on network sync here.
  - If the PC also has sensors (imagine PC also recording e.g. a high-quality camera or a separate GSR device via USB), then markers become extremely important to align phone data with PC's own data. The summary of Windows side suggests PC supports FLIR A65 and serial GSR too [150] [151]. If a use-case involves combining PC's own sensor data with phone's, we definitely need strong sync:
    - Possibly they use LSL on both so streams are auto-synced. If not, sending periodic sync pings is necessary. In that scenario, we should incorporate LSL, because LSL can have both phone and PC data in one timeline if used.

- **LSL Transmission:**

- If LSL is enabled, the phone creates LSL outlets for GSR, PPG, RGB frames (metadata), Thermal frames (metadata), etc., as described [18] [80]. The PC, if running LabRecorder or the PC app's LSL integration (which from Implementation Summary it does have after installing pylsl [83]), will subscribe to those streams.
- LSL's benefit is it handles time sync by itself (each stream has its own clock but LSL calibrates offsets). The PC's LSLReceiver would then get all data with globally aligned timestamps.
- We must ensure the naming matches: The Android LSLStreamingService defines stream names like "GSRRGBT_GSR", "GSRRGBT_RGB", etc. [152]. The PC config listed stream names in a JSON snippet [153] (GSR_Data, RGB_Video, Thermal_Video, PPG_Data). We might adjust the names to match or adjust PC config. The summary suggests PC expecting these exact names. We should set LSL stream names on Android to those in the config or vice versa.
- If LSL is fully working, it arguably simplifies integration: just turn it on on phone and PC and let them find each other. The downside is requiring both to be on same network and sending a lot of data possibly. It's fine in lab environment usually.
- One idea: use LSL for time sync and maybe small data (GSR, HR) and still record video locally for high quality. LSL would still get frame timestamps so data can be merged later.
- There's also an LSL event stream concept which we use for sync markers perhaps.
- In any case, if going LSL route, ensure that:
  - The Android LSL service gets properly initialized (the current stub will need actual LSL library included).

- The PC has pylsl installed (which they did).
- Test that PC can see the streams (there are tools like LabRecorder UI that can show active streams).
- Validate timing: Use LabRecorder to capture, say, the GSR from phone and a known PC clock stream to ensure offsets remain small (LSL does in-software NTP-like sync between clocks each few seconds to keep them aligned).

- In usage, we might pick either Sockets or LSL at a time to avoid double effort. The user might decide via settings: e.g., "Use LabStreamingLayer (ON/OFF)". If ON, maybe skip the socket approach to avoid duplicate overhead.

6.3 Testing Protocol for Coordinated Streaming

To verify the integration thoroughly, a testing protocol is needed:

Test 1: Single-phone to PC connectivity – With one Android phone and the Windows app: 1. Start the PC application (ensuring the Android receiver server is running on port 8080 and optionally LSL receiving if needed). 2. Connect the phone to the same Wi-Fi network as PC (if using normal TCP/IP) or initiate Wi-Fi Direct between them (if using that). 3. On the phone, enable network streaming in settings: e.g., choose "PC Sync: On" and enter PC's IP if needed. 4. Launch recording on phone. Observe on PC: - The PC console or UI should indicate a client connected (AndroidStreamReceiver might log "Server started..." then "Client connected from [IP]" [154] [155]). - DataReceived signals should fire – if PC has a UI to show values or just log to console, verify GSR values are printing, etc. If PC UI has plots (the description in the summary implies PyQt plots for GSR/PPG [156]), check that they move in real-time. - Check that timestamps on PC side are reasonable (not all zeros or unchanging). Also, PC's status signals like `connectionEstablished` should have triggered (maybe updating some GUI element). - If possible, produce a known pattern: For example, on phone create a manual sync marker by tapping a button that we program to send an 'event' – verify PC logs that event. - Stop recording on phone. PC should detect "session_stop" event and perhaps close out files or show a message. Ensure PC received all data up until stop (no significant data backlog that continues after stop). 5. Now examine the logged data: PC likely saves CSV or in-memory data. Compare it to phone's data (the phone also saved CSV locally). They should have the same count of events and aligned times (taking into account any offset). - For example, take a specific GSR sample index n – on phone's CSV, find timestamp, on PC's log find the timestamp for that sample. The difference ideally equals the sync offset we expect (if we used PC's local time for one and phone's epoch for other, you might need to align them manually). - If mismatches found, trace why (maybe some messages not received due to network – that indicates tuning needed e.g., increase socket buffer or identify if it was dropped over UDP). 6. Repeat by toggling different methods: try with LSL instead of raw sockets: - Turn off socket streaming and turn on LSL on phone, run LabRecorder on PC capturing streams. Record for a bit, then examine the XDF file or any output – ensure streams align, and that sample counts match expectation. - LabRecorder typically can show a graph of GSR as it records; see if it updates smoothly. - One can also check LSL timing: LabRecorder might output time stamps – see if difference between successive thermal and RGB frame timestamps is about the expected ~ a few ms (should be, if both share phone clock indirectly through LSL sync). 7. Try stressed conditions: e.g., run for 10 minutes streaming – see if any drift appears or PC complains about missing data. Also try temporarily disconnect network (turn off Wi-Fi for 5 sec then on) to see if reconnection works gracefully or if the system hangs. The phone's NetworkingManager has auto-reconnect settings we may tune [157], but testing will confirm if it works.

Test 2: Multi-phone coordinated with PC – If using multiple phones: 1. Set up two phones: designate one as master (with PC connection), another as slave. 2. Connect master to PC as above, and also connect slave to master via Bluetooth or Wi-Fi Direct. 3. Start recording via master (which should command slave to start almost simultaneously via network sync). 4. Observe PC: it should either get data from master only (which includes forwarded slave data if we implement that) or separate connections from both: - If separate, PC log would show two clients. It might then differentiate by client address. The PC's design doesn't explicitly combine multiple phone data; it treats each connection separately (it can handle multiple clients). That means it would keep track e.g., for each connected client, it logs accordingly. We need to ensure it doesn't mix them up. - Possibly better to test one at a time first. But if doing multi, see if PC prints data from both sources. 5. Verify synchronization between phones via PC data: e.g., if both phones had GSR sensors on two people and they clap together at an agreed time, do the GSR spikes align in PC logs after we align their time using offset? Or if one phone records an event (like a sync marker, which ideally triggers on all via network), does PC see markers from both or a combined one? 6. Multi-device is tricky to validate on PC; might have to rely on analyzing recorded logs post-hoc. We can cross-reference phone logs: ensure the offset we applied yields consistent ordering (e.g., a thermal frame from phone B that happened at same real time as one from phone A should have nearly same timestamp after offset). 7. Also test failure scenarios: slave disconnect mid-run (PC should still get master data, master maybe logs "device lost" event; when slave reconnects, how does PC handle data resuming?). - If the system can recover from a dropped slave and continue, that's excellent (likely a rare use-case, but important for field robustness).

Test 3: End-to-End Data Consistency – After integrated recording: - Collect a dataset with phone and PC together, then do a full comparison analysis: * Compare phone's local saved data vs PC saved data: + GSR series correlation (they should overlay if time axes aligned properly). + Frame timestamps sequence difference (maybe plot time difference between successive RGB frames as recorded by phone vs by PC – should be identical). + Check any lost segments (is PC missing some events that phone had? If yes, debug why – likely network drop or PC overhead. Then adjust design to fix that, e.g., by buffering or by acknowledging and resending if using TCP). * If differences found, implement fixes or at least measure how large differences are (e.g., PC got 98% of frames – maybe acceptable for monitoring, but if not, then consider compressing frames and sending via TCP to not lose them). - Evaluate system under performance limits: e.g., what if phone's CPU is near 100% and PC is receiving late? Possibly measure the end-to-end latency: phone event time vs when PC processes it (PC's AndroidReceiver could timestamp on arrival). The difference is network + any PC queueing. Ideally it's small (<0.1s on Wi-Fi, maybe ~1s on Bluetooth if slower). If latency is too high, consider dropping frames or not sending some data in real-time and only send summary (tunable based on need). - Use synchronization markers to validate combined timeline: For example, on phone generate a sync marker at a known time (like exactly 5.000s after start). See what timestamp PC assigns to it and confirm it matches after offset. If we implement an explicit offset, then PC might adjust it. The test here is whether our offset calibration is working. We might refine offset by comparing a few markers (like average difference).

Test 4: UI/UX Integration Testing: - Ensure the user controls are intuitive: e.g., if user checks "Stream to PC" on phone, the app automatically attempts connection and shows status "Connected to PC" or error if fails (maybe via a toast). If possible, on PC side, some acknowledgment or visual indicator that phone is connected (the PC UI likely has something). - Test the scenario of forgetting to turn on PC or PC app – phone should not hang or crash; maybe it will just log error "Connection refused" and continue local recording. That's acceptable; perhaps after that it could retry periodically. - Conversely, if PC is on but phone doesn't connect (like wrong IP), need a way to troubleshoot – but at least ensure it times out and doesn't freeze the phone UI. - If using Wi-Fi Direct, test the workflow: user taps "Connect devices" on phone, phone shows a list of peers, connect, etc. That part might not have an interface in our app currently – if not, user must do from phone's Wi-Fi Direct settings or we quickly implement a simple connect logic (maybe assume a particular device name for now). This is a usability gap to fill if multi-phone is heavily used.

By following this testing protocol, we can gain confidence in the integrated system: - That data is consistent across phone and PC, - That time synchronization is functioning (within acceptable error), -

That the system is robust to typical network issues, - And that the user interface supports using the integration without confusion.

Finally, it's advisable to keep a **synchronization test log**: a document where each integration test run is recorded with date, devices used, any issues observed, and whether it passed. This will help track improvements and ensure that a fix in one part doesn't break another (regression testing). Also, running a full integration test before any major demo or deployment is highly recommended to catch last-minute issues (like a forgotten permission or a firewall blocking PC port, etc.).

In summary, the integration between FYP-GSR-Android and FYP-GSR-Windows is designed to be flexible: supporting a direct socket connection method as well as the Lab Streaming Layer protocol. We have to ensure that whichever method is used, the **data formats align** and **timestamps are synchronized** so that the combined dataset is coherent. Through careful adherence to a data protocol and thorough testing (as outlined), we can achieve a smooth coordination where, for example, a researcher can watch on their PC as a participant's GSR rises in sync with a thermal image of their face flushing, all in real-time. This capability demonstrates the full power of the integrated system for multimodal physiological monitoring.

7. Appendices

(The appendices provide supplementary information that may be useful for reference, troubleshooting, or deeper understanding of the system. This includes a glossary of important classes/modules in the code, tables of required permissions and configuration settings, and a summary of known issues with their mitigation strategies.)

7.1 Glossary of Key Classes and Modules

This glossary briefly describes the main components of the FYP-GSR-Android app (and related PC app where relevant):

- **CaptureModule (interface)** – Found in `capture/CaptureModule.kt`. Defines the basic lifecycle for sensor capture components (`init()`, `start()`, `stop()`, `release()`, etc.) and status reporting. All specific sensor modules implement this.
- **ThermalCaptureModule** – Handles the Topdon thermal camera. Initializes SDK, sets up frame callbacks, processes frames (timestamping and sending to sync manager) [158] [37]. Contains simulation mode fallback if camera not present [40]. Key constants: resolution (256x192 native, interpolated to 320x240) [159], target FPS 25. Located in `thermal/`.
- **RGBCaptureModule** – Manages the device's RGB camera via Camera2 API. Likely opens a camera session with specified resolution and frame rate. It captures frames (perhaps as ByteBuffer or Image and then calls SyncManager with each frame's data or metadata). Not deeply shown in snippet, but analogous structure to Thermal. Located in `rgb/`.
- **GSRCaptureModuleSimulation / (Real GSRCaptureModule)** – The simulation version generates synthetic GSR and PPG values periodically [160] [161]. The real version (to be implemented) will connect to Shimmer BLE and stream actual readings. These reside in `gsr/`.
- **AudioCaptureModule** – Uses Android's AudioRecord to capture audio segments. Key parameters: sampleRate (default 16kHz), channelConfig (mono), segmentDuration (100ms) [47] [162]. It pushes AudioDataEvents with PCM bytes and timestamps to SyncManager. Supports simulation mode if mic not available [51]. Located in `audio/`.

- **SynchronizationManager** – Central coordinator that timestamps incoming data from all CaptureModules with a common clock [6]. Likely holds a queue or directly dispatches events to DataRecorder and/or Network. It may also generate SyncMarkerEvents at start/stop and handle incoming sync commands for multi-device. In `sync/SynchronizationManager.kt`.
- **DataEvent (and subclasses)** – Data model classes for events [163]:
- ThermalFrameEvent (timestamp, frameData byte array or maybe not serialized fully, width, height, frameIndex) [142].
- RGBFrameEvent (timestamp, frameData or not, width, height, frameIndex, format) [164].
- GSRDataEvent (timestamp, gsrValue, sampleIndex, sensorTimestamp, ppgValue) [8].
- SyncMarkerEvent (timestamp, markerType like "start"/"stop", description) [165].
- AudioDataEvent (timestamp, audioData byte array, sampleRate, channels, sampleIndex, durationMs) [166]. All marked `@Serializable` for JSON encoding. Located in `data/DataEvent.kt`.
- **RecordingController** – High-level orchestrator of recording sessions [11]. It creates instances of each CaptureModule (thermal, rgb, gsr (simulated or real), audio) [12], and manages starting them in sync using a CountDownLatch [13]. It also ties in the NetworkingManager and LocalDataRecorder. It monitors status of modules and can stop all on error. Essentially, the “brain” that Start/Stop button calls into. Found in `controller/RecordingController.kt`.
- **LocalDataRecorder** – Not extensively shown above, but likely part of the RecordingController which handles writing events to storage. It probably pulls events from SynchronizationManager or gets called by SyncManager for each event. It then uses DataWriter (CSV or potentially other formats) to log. Could involve a background coroutine.
- **CsvDataWriter** – Implementation of DataWriter that writes to CSV files [167]. It opens files for each type (audio.csv, gsr.csv, etc.) and appends lines with event data [60] [58] [59]. Uses a Mutex for thread safety and flushes periodically. Located in `data/CsvDataWriter.kt`.
- **NetworkingManager** – Manages network sync between devices [168]. It holds instances of:
- BluetoothNetworkingService,
- WiFiNetworkingService,
- LSLStreamingService, but only one active at a time. Provides methods to start as master/slave, discover devices, connect, disconnect, send events over network [169] [170]. Essentially an abstraction so RecordingController can use networking without caring about the details.
- **BluetoothNetworkingService** – Implementation to sync devices via Bluetooth. Likely uses BluetoothSocket (for classic) or BLE GATT for communication. Should implement all interface methods (startAsMaster might open a server socket, etc.). Not shown above but listed in project. Would be in `networking/BluetoothNetworkingService.kt`.
- **WiFiNetworkingService** – Implementation using Wi-Fi Direct (P2P) [171]. It uses WifiP2pManager to create groups and connect peers. Transfers data presumably via standard sockets once connected (the code opens a ServerSocket on port 8888 for master and uses Socket for client to connect) [172] [119] [173]. It serializes DataEvents to JSON (using kotlinx Json as DataStreamer does) and sends them through I/O streams. Also listens for incoming data from clients and processes it.
- **LSLStreamingService** – Implementation for Lab Streaming Layer [174]. It doesn’t connect devices per se, but opens LSL outlets. It treats LSL as a special “network type”. It doesn’t have separate master/slave – each device just streams out. The PC or others pick up the streams by name. It simulates operation if lib not found (the stub LSLOutlet logs if no native lib) [175] [176]. If properly loaded, it uses native calls to create streams and push samples [177] [178].
- **DataStreamer** – A helper class in `remote/DataStreamer.kt` that manages streaming DataEvents to multiple clients over sockets [179] [180]. NetworkingServices might use it; e.g., WiFiNetworkingService possibly sets a clientSender callback which actually writes to socket. DataStreamer has a channel buffer and a thread sending out messages to all clients in its list [181]

[10]. It JSON-encodes DataEvent using kotlinx serialization [125] and adds a prefix like "DATA:" (if needed) to the message. It also can broadcast commands like "STREAMING_STOPPED" [182].
- **MainActivity (Android)** – The main UI controller for Android app [183]. Manages permissions on startup [184], sets up UI elements (TextureView for preview, Buttons) [185], and interacts with RecordingController. It starts the RemoteControlService (which might be for PC remote commands via network) [186]. Also handles menu actions like opening Settings or DevicePairing dialog [187]. It updates UI status text by collecting flows from RecordingController's status or directly from modules (likely in a repeatOnLifecycle coroutine).
- **DevicePairingDialog** – UI fragment for scanning and selecting BLE devices (intended for picking the Shimmer) [31]. It uses BluetoothAdapter's LE scanner, handles the scan results and lists devices. On select, it passes the BluetoothDevice to a listener (MainActivity implements listener and in onDeviceSelected, currently just logs and TODO to connect) [188] [33].
- **PerformanceMonitor** – Likely a class that measures performance metrics (like CPU usage, memory) in background, possibly via Android's Debug APIs or reading /proc stats. It might log or even update UI with these metrics (maybe that "statusOverlay" text could show FPS or memory usage if configured).
- **RemoteControlService** – Possibly an Android Service that handles commands from the PC to control the app remotely (like start/stop recording triggered from PC UI). The code in MainActivity starts it and registers a BroadcastReceiver to handle commands like `ACTION_COMMAND_RECEIVED` [189]. If PC sends a JSON command like "START" to phone (the PC has a `send_command` method [190]), this service might broadcast it within the app, and RecordingController or MainActivity reacts by starting/stopping. This is how multi-device sync might be triggered too. It basically allows the PC or a master phone to instruct a slave phone.
- **ConfigurationValidator** – (Mentioned in MainActivity) likely checks that all required hardware features are present (e.g., does device have BLE, USB host, enough camera support, etc.) and prints warnings. Good to have to ensure environment is ready.
- **Exceptions (custom)** – Possibly classes like `CameraNotAvailableException` etc., grouped in `exceptions/` which RecordingController might catch to show user-friendly messages.

(On the Windows side for completeness:)

- **AndroidStreamReceiver (PC)** – Module that listens for incoming socket connections from phone(s) [191] [123]. It spawns threads for each client and processes incoming JSON messages, emitting Qt signals for data or status changes [192] [193]. As described, it has `_handle_gsr_data`, `_handle_rgb_frame`, etc., to integrate with the GUI or logging on PC [68] [69].
- **AndroidSyncManager (PC)** – Handles time sync between PC and phone(s) [88]. It might listen for a special sync command or event and compute offset. Possibly also instructs the PC DataLogger to align times. It interacts with AndroidStreamReceiver by connecting to its signals (like on dataReceived to possibly log them, and maybe do something on sync events).
- **LSLManager / LSLStreamer (PC)** – From the summary, PC has classes for managing LSL output and logging (like log_ppg_data, etc.) [157] [194]. The PC can itself push data into LSL or record from LSL, depending on config.
- **UI (PC)** – The PC uses PyQt5, so classes like `MainWindow`, `PlotWidget` likely exist to display live plots of GSR/PPG, video frames (maybe using a QLabel or OpenCV integration for frames), and indicators of connections. The UI likely has settings dialog for IP, port, etc. The code references `settings_dialog.py`, `dual_phone_manager.py` which implies PC can also coordinate two phones (the name suggests managing two phone inputs).

This glossary should help developers and users quickly identify what part of the system is responsible for what functionality and where to look in the code for certain behaviors.

7.2 Permissions and Configuration Table

The following table summarizes the key permissions required by the Android app and their purpose, as well as important configuration settings and their meaning:

Permission (Android)	Purpose	Required in
CAMERA	Access camera for RGB video	MainActivity (runtime) [53]
RECORD_AUDIO	Access microphone for audio recording	*Should be in MainActivity (Add)*
BLUETOOTH_CONNECT (API ≥31)	Connect to paired Bluetooth devices (Shimmer)	MainActivity (runtime) [53]
BLUETOOTH_SCAN (API ≥31)	Discover BLE devices (for pairing Shimmer)	MainActivity (runtime) [53]
ACCESS_FINE_LOCATION	Required for BLE scanning on Android (and Wi-Fi Direct discovery)	MainActivity (runtime) [195]
WRITE_EXTERNAL_STORAGE	(For Android < Q) Save recordings to external storage (files directory)	MainActivity (runtime) [195]
READ_EXTERNAL_STORAGE	(Not strictly needed for app itself unless user wants to browse files)	MainActivity (runtime) [195]
INTERNET (in Manifest)	Allow network socket connections (for Wi-Fi/LSL)	Manifest (no runtime needed)
ACCESS_WIFI_STATE / CHANGE_WIFI_STATE	(Possibly needed for Wi-Fi Direct) Manage Wi-Fi connections for P2P	Manifest (for Wi-Fi Direct)
ACCESS_COARSE_LOCATION	(Older Android) For BLE if fine not available	Manifest (though fine covers it)

Note: On Android 10+, WRITE_EXTERNAL_STORAGE is not needed for writing to app's own external files dir, but it's included likely for backward compatibility or if user might want to share files. Microphone permission (RECORD_AUDIO) must be added to both manifest and request logic.

Configuration Settings (App Settings):

These are typical configurable parameters in the app (through Settings UI or constants):

- **Participant ID** – A user-entered identifier for the subject or session. Used for naming session folder and recorded files [129] and possibly stored in metadata.txt for reference.
- **Recording Mode** – e.g., "Local" (store on phone only) vs "Network" (stream to PC/other device) or "Both". This toggles NetworkingManager usage. If "none" network, the app won't start networking.
- **Network Type** – If network mode on, choose between Bluetooth or Wi-Fi Direct or LSL for multi-device sync. (Perhaps an enum in settings).
- **Device Role** – Master or Slave when multi-device. Master will initiate sync and possibly connect to PC; Slave will wait for master commands.

- **Auto Start** – (If implemented) whether recording should auto-begin when app launches or devices connect. Useful for headless operation, but likely default off.
- **RGB Resolution** – Options: 720p, 1080p, 4K as seen in code [44]. Defaults to 1080p. A higher resolution will increase storage and maybe slight processing cost; lower might be used to save space or if performance needed.
- **Thermal Interpolation** – Possibly a toggle if user wants to store original 256x192 frames instead of interpolated 320x240. Not clearly exposed, but mention in code comment suggests something about enabling/disabling interpolation.
- **GSR Sampling Rate** – Options like 64, 128, 256, 512 Hz in settings [196]. Shimmer's max is 128 Hz for GSR I believe, but maybe PPG can do more? Or simulation can. The default is 128 Hz. If set higher, and hardware can't, it might fall back or oversample with duplicates – need to ensure realistic. We likely keep at 128 but option exists.
- **Preview Mode** – default camera view (RGB or Thermal or Hybrid) when app starts [197]. This is a user preference whether they want to see thermal imagery or normal camera by default.
- **Save Raw Data** – Possibly a setting `includeRawData` in DataWriter config [198]. If true, maybe store raw frame bytes to files (though currently we do not implement separate saving of raw images except via sending to PC or via video encoding).
- **Buffer Sizes** – Not user facing, but configuration constants like DataWriter buffer size (8KB default) [199], channel capacities (DataStreamer buffer 1000) [95]. These could be tweaked if needed (likely not exposed to end user).
- **Retry Attempts** – Networking: e.g., Bluetooth reconnect max retries = 3 (default) [157], with delays 5s, 10s, etc. Possibly configurable in a config file or constants.
- **Port Numbers** – If needed to change (default 8080 for PC server, 8888 for Wi-Fi Direct group owner). Usually fixed in code, but mention for completeness.
- **LSL Stream Names** – Configured either in code or a config file on PC. Ensuring names match is more a developer config than user, but if making it adjustable, one could allow custom stream name prefix.

Wi-Fi Direct configuration: The Wi-Fi Direct connection doesn't require user to join network, it auto-groups. But ensure both devices have Wi-Fi on. Permissions needed: Fine Location (which we have). Also on Android 29+, need to ask for location on each discovery, but code seems to handle with permission check.

Shimmer BLE configuration: The Shimmer GSR+ device might need to be paired via Bluetooth settings or at least ensure phone's BLE is on. Usually no PIN or pairing needed for BLE (pairing dialog appears only if using classic). The app's pairing dialog triggers scan and connect directly.

Thermal camera usage: The Topdon TC001 may have an **OTG mode** toggle in some phones (some require enabling USB host). Also on some Androids, if multiple USB devices, one might specify which to use. Our app simply finds the first matching USB device. If multiple Topdon or similar present (rare), user might have to ensure only one plugged. Also, in Developer Options, some devices have settings for USB – ensure not in "charging only" but "USB device" mode. Usually granting permission handles that.

In conclusion, this table and notes can help a developer or power user ensure all needed permissions are granted and settings configured to get the app running properly. It can also be part of a user manual's quick setup section.

7.3 Known Issues and Mitigation Plans

Below is a list of known issues (as of this audit) in the system and how we plan to address or work around them:

- **Issue: Shimmer GSR not actually connected (simulation used)** – *Status:* Under development. *Mitigation:* Implement the BLE connectivity as highest priority (Phase 1). Until then, if a real sensor must be used immediately, one workaround could be to use Shimmer's Android app separately to collect GSR in parallel – but synchronization would be manual. Not ideal, hence implementing our support is the proper fix.
- **Issue: Thermal camera preview blank or not intuitive** – Because rendering wasn't implemented, user might think thermal isn't working when it's actually capturing data in background. *Mitigation:* Implement at least grayscale visual. Meanwhile, as a temporary measure, the app's status text does show "Capturing (Sim)" vs "Capturing" for thermal to indicate if it's simulation or real [200]. We can also log frame arrival count in UI to reassure that it's working (e.g., update a counter on screen).
- **Issue: Microphone permission missing causing silent audio** – *Mitigation:* Add permission request. In the interim (if one cannot update the app easily), the app will run in simulation mode for audio. That means silent audio in output (or some dummy tone perhaps). This is not critical for some studies but if needed, the user would have to manually grant mic permission via settings. We note this in documentation so if audio is unexpectedly not recording, the user can check permissions.
- **Issue: Bluetooth LE scanning requires Location On** – Many users forget to enable GPS when using BLE apps. This will result in no devices found in pairing dialog. *Mitigation:* Detect if Location is off and prompt user. For example, when user opens DevicePairingDialog, check `locationManager.isProviderEnabled()` – if false, show alert "Please enable location services for BLE scanning." Also could incorporate a way to open location settings. This is documented in user guide as a troubleshooting tip.
- **Issue: Wi-Fi Direct connection sometimes fails or is finicky** – Wi-Fi Direct can be affected by device compatibility. *Mitigation:* Provide alternative (Bluetooth) if Wi-Fi Direct fails. Also instruct users to ensure both devices not connected to any Wi-Fi network (some devices disable P2P when actively on Wi-Fi, though usually they coexist). If group creation fails, our app emits an error event (we handle that in logs [201]). The user may need to retry. We plan exponential backoff (maybe auto retry after a few seconds up to 3 attempts). Document that if one method fails, try toggling Wi-Fi or use BT as backup.
- **Issue: Data alignment between devices needs calibration** – If after initial tests, we find a constant offset (say all of device B's timestamps are 40ms behind device A's), it's likely clock offset. *Mitigation:* Implement an automatic offset calibration at start (as described in integration section). Also, as a quick fix, in analysis one can subtract that offset. But we prefer to handle in app to present unified times. We will thoroughly test and adjust algorithm. Meanwhile, note in docs: "If using multi-device, you may notice a small constant lag; this can be corrected by calibrating with a sync event like a clap at the beginning and aligning in data processing."
- **Issue: High battery consumption / thermal throttling** – During prolonged use, phone might overheat or battery drain quickly. *Mitigation:* Instruct users to do things like turn off unnecessary apps, reduce screen brightness, maybe use airplane mode (except Bluetooth) to reduce network usage overhead, etc. Possibly use a back cover cooling fan (as used in gaming phones) if needed in extreme cases. For now, we also allow toggling off camera preview if not needed (looking at just text values, can free some GPU work). For battery, recommend an external power pack for long sessions. Overheating can cause camera frame drops; if we detect device is too hot (some devices expose a thermal status), we might reduce frame rate or warn user.

- **Issue: App not handling background/interruptions** – If the app is sent to background (home button) during recording, Camera and Audio might pause or stop due to lifecycle. We haven't explicitly handled onPause beyond releasing controller onDestroy. *Mitigation:* For now, advise users to keep app in foreground while recording (lock screen from sleeping via a WakeLock if necessary). In future, implement a foreground service for recording to continue if screen off. But currently, treat backgrounding as end of session. We will document: "Do not switch away from app or turn off screen during recording, as this may interrupt sensor capture." This is a limitation we plan to fix in later versions with a persistent service.
- **Issue: File access on PC for live images** – Not really an app bug, but the integration: PC gets a file path from phone but cannot read it because it's on phone storage. *Mitigation:* Our design will avoid sending file paths unless we implement a fetch mechanism. If needed, we could share a network drive or just not rely on file paths. For now, ignoring `file_path` on PC side and focusing on data content. So if one sees `file_path` logged on PC and wonders what to do with it – the answer is nothing unless we set up a later transfer of actual files.
- **Issue: Minor UI glitches** – e.g., Hybrid mode not implemented: if user toggles to "Hybrid", it might just show the last selected or nothing. *Mitigation:* Until implemented, we could remove or disable the Hybrid toggle. If left in, mention in help that Hybrid preview is experimental. Also, UI rotation (if not locked) could restart camera – we might lock orientation to portrait for stability.

- **Issue: Known device-specific quirks** – e.g., Some Samsung phones in power saving mode throttle background threads heavily – could cause frame drops or Bluetooth lag. *Mitigation:* Recommend users disable battery optimizations for the app (via Android settings) so system won't put it to sleep. Possibly detect if on battery saver and warn. Another example: Huawei phones often kill apps in background aggressively – again, instruct to allow the app to run in background or keep screen on.

- **Issue: Large log files if verbose logging on** – The app logs a lot (tag GSRRGBT at debug) which could, over a long experiment, slow things or fill log buffer. *Mitigation:* Possibly turn down log level in production build or allow enabling/disabling verbose logs in settings. For now, this is not user-facing but developer note to avoid leaving super-verbose logging on in release builds.

- **Issue: Unimplemented license information** – Just a note, the README has "License info to be added" [202] . If releasing widely, ensure we add appropriate license text (especially if using any vendor SDK with restrictions).
- **Issue: Multi-device naming** – If multiple phones connect to PC, their data might mix. PC currently identifies by address string. Perhaps not a big issue if we know which is which, but we might label events with device name or ID. If not done, the user must differentiate manually. *Mitigation:* Could prepend an ID in data on PC log (like device1_gsr vs device2_gsr streams). This is an enhancement to implement if multi-device logging to single file.

All the above issues have either immediate fixes in progress or at least workarounds and are documented so users and developers are aware. By systematically knocking these issues out (particularly those in short-term list) and updating documentation accordingly, we aim to reach a state where the system is robust and any limitations are clearly communicated.

Document Revision: This technical audit and upgrade plan reflect the state of the system as of *June 2025*. As development continues (especially through the phased plan detailed), this document should be updated to mark issues resolved and note new improvements. The comparative analysis and performance expectations should also be revisited after significant updates (for instance, after Phase 3 multi-device implementation, measure actual sync error achieved and update Section 4).

By following this plan and recommendations, the FYP-GSR-Android application and its integration with FYP-GSR-Windows will evolve into a stable, high-performance platform that stands well against both commercial and research alternatives, and provides a solid foundation for future research and development in multimodal physiological sensing.

---

1 2 3 4 6 13 30 35 46 52 62 100 101 102 106 107 108 114 116 134 135 202 GitHub
https://github.com/buccancs/fyp-gsr-android/blob/bd9828a3a2899254d1e2a1e042cdd70175637d52/README.md

5 28 29 111 160 161 GitHub
https://github.com/buccancs/fyp-gsr-android/blob/bd9828a3a2899254d1e2a1e042cdd70175637d52/app/src/main/java/com/gsrrgbt/android/gsr/GSRCaptureModuleSimulation.kt

7 8 45 71 86 87 141 142 163 164 165 166 GitHub
https://github.com/buccancs/fyp-gsr-android/blob/bd9828a3a2899254d1e2a1e042cdd70175637d52/app/src/main/java/com/gsrrgbt/android/data/DataEvent.kt

9 10 95 96 97 124 125 179 180 181 182 GitHub
https://github.com/buccancs/fyp-gsr-android/blob/bd9828a3a2899254d1e2a1e042cdd70175637d52/app/src/main/java/com/gsrrgbt/android/remote/DataStreamer.kt

11 12 44 94 99 137 138 189 196 197 GitHub
https://github.com/buccancs/fyp-gsr-android/blob/bd9828a3a2899254d1e2a1e042cdd70175637d52/app/src/main/java/com/gsrrgbt/android/controller/RecordingController.kt

14 15 16 17 76 121 122 168 169 170 GitHub
https://github.com/buccancs/fyp-gsr-android/blob/bd9828a3a2899254d1e2a1e042cdd70175637d52/app/src/main/java/com/gsrrgbt/android/networking/NetworkingManager.kt

18 19 77 78 79 80 81 82 84 85 152 174 175 176 177 178 GitHub
https://github.com/buccancs/fyp-gsr-android/blob/bd9828a3a2899254d1e2a1e042cdd70175637d52/app/src/main/java/com/gsrrgbt/android/networking/LSLStreamingService.kt

20 21 73 74 75 GitHub
https://github.com/buccancs/fyp-gsr-android/blob/bd9828a3a2899254d1e2a1e042cdd70175637d52/app/src/main/java/com/gsrrgbt/android/networking/NetworkingService.kt

22 23 24 110 131 132 133 GitHub
https://github.com/buccancs/fyp-gsr-android/blob/bd9828a3a2899254d1e2a1e042cdd70175637d52/docs/android_research.tex

25 26 27 36 37 38 39 40 41 42 43 98 112 113 115 136 158 159 200 GitHub
https://github.com/buccancs/fyp-gsr-android/blob/bd9828a3a2899254d1e2a1e042cdd70175637d52/app/src/main/java/com/gsrrgbt/android/thermal/ThermalCaptureModule.kt

31 32 103 104 105 GitHub
https://github.com/buccancs/fyp-gsr-android/blob/bd9828a3a2899254d1e2a1e042cdd70175637d52/app/src/main/java/com/gsrrgbt/android/ui/DevicePairingDialog.kt

33 53 93 109 130 183 184 185 186 187 188 195 GitHub
https://github.com/buccancs/fyp-gsr-android/blob/bd9828a3a2899254d1e2a1e042cdd70175637d52/app/src/main/java/com/gsrrgbt/android/ui/MainActivity.kt

34 GitHub
https://github.com/buccancs/fyp-gsr-android/blob/bd9828a3a2899254d1e2a1e042cdd70175637d52/app/src/main/java/com/topdon/thermal/sdk/ThermalSDK.kt

47 48 49 50 51 162 GitHub
https://github.com/buccancs/fyp-gsr-android/blob/bd9828a3a2899254d1e2a1e042cdd70175637d52/app/src/main/java/com/gsrrgbt/android/audio/AudioCaptureModule.kt

54 55 56 57 58 59 60 61 92 118 129 167 GitHub
https://github.com/buccancs/fyp-gsr-android/blob/bd9828a3a2899254d1e2a1e042cdd70175637d52/app/src/main/java/com/gsrrgbt/android/data/CsvDataWriter.kt

63 64 65 66 67 68 69 70 72 88 89 90 91 123 126 127 139 140 143 144 145 146 147 148 149 154 155 190 191 192 193 GitHub
https://github.com/buccancs/fyp-gsr-windows/blob/4cd1d1074afcabad6987ac5bec39dcc8df22a644/src/network/android_receiver.py

83 128 150 151 153 156 157 194 GitHub
https://github.com/buccancs/fyp-gsr-windows/blob/4cd1d1074afcabad6987ac5bec39dcc8df22a644/docs/summaries/IMPLEMENTATION_COMPLETION_SUMMARY.md

117 198 199 GitHub
https://github.com/buccancs/fyp-gsr-android/blob/bd9828a3a2899254d1e2a1e042cdd70175637d52/app/src/main/java/com/gsrrgbt/android/data/DataWriter.kt

119 120 171 172 173 201 GitHub
https://github.com/buccancs/fyp-gsr-android/blob/bd9828a3a2899254d1e2a1e042cdd70175637d52/app/src/main/java/com/gsrrgbt/android/networking/WiFiNetworkingService.kt