# Milestone 3.4: Calibration Engine (OpenCV) – Technical Guide

## Overview and Goals

Calibrating the cameras is crucial to align the RGB (visible light) camera and the thermal camera on each phone. The calibration process will determine the internal parameters of each camera (intrinsics) and the precise 3D relationship between the RGB and thermal camera (extrinsics). With these parameters, the PC application can **spatially correlate thermal and visual data**, for example mapping a hot spot in the thermal image to the correct location in the RGB image. This milestone will implement a complete calibration workflow using OpenCV on the PC, with coordination between the PC and Android devices via socket communication (network sockets). Key goals include:

* **Calibration Data Capture:** Provide a guided procedure in the PC app for capturing synchronized images from both the RGB and thermal cameras (on one or both phones). The user will capture multiple images of a known calibration pattern (e.g. a chessboard or ArUco marker board) from different angles.
* **Compute Calibration Parameters:** Use OpenCV (cv2 in Python) to detect the calibration pattern in the images and compute camera intrinsics and the transformation between RGB and thermal cameras. This will involve functions like cv2.findChessboardCorners, cv2.calibrateCamera, and cv2.stereoCalibrate.
* **Store Calibration Results:** Save the calculated parameters (camera matrices, distortion coefficients, rotation & translation between cameras, etc.) to a file (e.g. YAML/JSON) so they persist for future sessions. This allows re-use of calibration without repeating the process each time.
* **Real-time Overlay (Optional):** Utilize the calibration to enable an **overlay of thermal imagery onto the RGB video feed** in the PC app. When enabled (via a toggle in the UI), the system will warp and blend the thermal image on top of the RGB image in real-time, providing the operator a fused view. This will be implemented carefully to ensure it can be toggled on/off for performance or preference.
* **User Interface Integration:** Create a user-friendly interface (e.g. a calibration dialog or wizard in the PC app) to guide the user through calibration. The UI should include instructions, a button to capture frames, a counter of how many frames have been captured, a “Compute Calibration” button (enabled after enough frames), and feedback on the calibration result (e.g. error metrics). The UI may also show thumbnails of captured images or overlay previews to help the user verify the pattern was captured correctly.

By the end of this milestone, the PC application will have a robust calibration engine that ensures **each phone’s RGB and thermal cameras are precisely aligned**. This lays the groundwork for accurate multi-modal data analysis in later stages.

## System Architecture and Components

To implement the calibration feature, several components and classes will work together across the PC application and the Android devices. Below is a breakdown of the main modules and their roles:

* **PC Application (Calibration Module):**
* **CalibrationManager (or CalibrationEngine) Class:** This class orchestrates the entire calibration workflow on the PC. It provides methods to start a calibration session, trigger image capture on devices, collect images, run the OpenCV calibration computations, and store the results.
  + *Methods:* start\_calibration(device\_ids), capture\_frame(), compute\_calibration(), save\_results(file\_path), load\_results(file\_path), apply\_overlay(frame\_rgb, frame\_thermal) etc.
  + *Attributes:* Lists or buffers for storing captured image pairs (for each device, e.g. calib\_images[device\_id]['rgb'] and calib\_images[device\_id]['thermal']), calibration pattern settings (like chessboard dimensions), and results (camera matrices, distortion coeffs, R/T matrices for each device).
* **CalibrationProcessor Class/Module:** This contains the OpenCV-related functions that perform the heavy-lifting. It can be a utility module used by CalibrationManager. Functions include:
  + find\_calibration\_corners(image): detect chessboard or ArUco corners in a given image (returns 2D points if found).
  + calibrate\_intrinsics(objpoints, imgpoints, image\_size): wrapper around cv2.calibrateCamera to compute a camera’s intrinsic parameters[[1]](https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html#:~:text=Now%20that%20we%20have%20our,rotation%20and%20translation%20vectors%20etc).
  + calibrate\_extrinsics(objpoints, imgpoints1, imgpoints2, cameraMatrix1, dist1, cameraMatrix2, dist2, image\_size): uses cv2.stereoCalibrate to find rotation and translation between two cameras[[2]](https://temugeb.github.io/opencv/python/2021/02/02/stereo-camera-calibration-and-triangulation.html#:~:text=stereocalibration_flags%20%3D%20cv,criteria%2C%20flags%20%3D%20stereocalibration_flags).
  + (Optional) compute\_homography(points1, points2): compute a homography between two image planes given corresponding points (could be used for overlay if assuming a planar scene).
* **CalibrationResult Class/Struct:** A simple data container for the results of calibration. For each device (or each camera pair) it holds: cameraMatrix\_rgb, distCoeffs\_rgb, cameraMatrix\_thermal, distCoeffs\_thermal, R, T, and perhaps derived matrices like homography for overlay or rectification maps. It might also store a reprojection error value to indicate quality. This class can have a method to serialize to disk (e.g. to YAML/JSON) and to deserialize (for loading saved calibration).
* **PC Application (UI Integration):**
* **CalibrationDialog/Panel:** A GUI component (could be a new window or a panel in the main app) that provides the user interface for calibration. It likely includes:
  + Instructions for the user (e.g. “Place the calibration pattern in view of the cameras”).
  + A **“Capture Calibration Frame”** button to capture a new set of images.
  + A counter or list displaying number of frames captured (e.g. “Frames captured: 3/10”).
  + A **“Compute Calibration”** button, which remains disabled until a minimum number of frames (e.g. 5 or more) have been captured.
  + Possibly thumbnails or indicators for each captured frame (to review if the pattern was detected).
  + After computation, display the results: could be text like “Calibration successful. RMS error = X pixels” and maybe the intrinsics/extrinsics values or a simplified summary.
  + If possible, a small preview where the user can see an overlay of images for verification (for example, showing one of the captured RGB images with the thermal image projected onto it to illustrate the alignment).
  + A **“Save Calibration”** action (if not saved automatically) to export the calibration parameters, and maybe an **“Apply Overlay”** checkbox to toggle the real-time overlay on the main view.
* **Main Application Integration:** The main app (which likely already displays live video from the phones) will have hooks to start the calibration dialog and to use the calibration results. For example, a menu item “Calibrate Cameras” opens the CalibrationDialog. Also, the main preview panel will check if an overlay is enabled and if so, use the calibration data to blend thermal imagery onto the RGB feed.
* **Android Phone App (Capture side):**  
  *Note: Milestone 2.8 on the Android side is about supporting calibration capture.* Each phone’s app needs to respond to calibration commands from the PC and provide images from both cameras:
* **Socket Command Handler:** The Android app (running on each phone) should have a listener for a command (e.g. a JSON message or simple string) like "CAPTURE\_CALIBRATION". This likely is implemented in the same socket communication system used for recording data. When the PC sends this command, the app knows the user is requesting a calibration image.
* **Dual Camera Capture:** Upon receiving the command, the Android app should **capture a frame from the RGB camera and the thermal camera in synchronization**. Ideally, both images should be taken at as close to the same time as possible (to ensure the calibration pattern hasn’t moved between captures). If true simultaneous capture is not possible due to hardware/API limitations, capturing them sequentially within a second is acceptable (the calibration object is typically static during capture). Each phone should then transmit the captured images back to the PC via the socket connection. For example, the app can send a message containing the RGB image (perhaps JPEG or PNG encoded) followed by the thermal image. The data can include identifiers or headers so the PC knows which is which (e.g. a simple protocol: send a short header like “RGB\_IMAGE” then the image bytes, then “THERMAL\_IMAGE” and its bytes, or use separate socket channels if available).
* **Image Format Considerations:** The phone’s RGB image will be in color (likely JPEG). The thermal image may be grayscale (and possibly lower resolution, e.g. 160x120 or 256x192). The thermal camera image should be sent in a format the PC can easily read – possibly a grayscale PNG or even as raw data that the PC can interpret. If needed, the thermal data could be normalized to 8-bit before sending (if the thermal camera provides 14-bit raw values, for example). The PC’s calibration code will handle grayscale images for corner detection.
* **Each Phone Calibrated Separately:** Since the user specified **each phone is calibrated individually**, the app will treat calibration commands per device. If two phones are connected, the PC might either calibrate one at a time or request both to capture concurrently. We will design so that the PC *can* send the capture command to both phones at once (to speed up the process), but the calibration calculations will be performed separately for each phone’s image set. This means the Android app doesn’t need to coordinate with the other phone – each just sends its images back. The PC groups images by device ID internally.

## Setting Up the Development Environment

Before diving into coding the calibration engine, ensure the development environment is prepared for OpenCV and the project structure is updated:

* **Install OpenCV for Python:** The calibration code will use the OpenCV Python library (cv2). If not already installed, add it to your project. In a Python environment, you can install via pip:
* pip install opencv-python opencv-contrib-python
* We include opencv-contrib-python to have access to extra modules like ArUco (in case we use marker boards). This installation will also include NumPy, which OpenCV uses extensively.
* **IDE Configuration:** If using an IDE like PyCharm or VS Code, ensure that the interpreter/environment for your project has the OpenCV packages installed. In PyCharm, for example, go to **Settings > Python Interpreter** and add the packages if needed. In VS Code, confirm the selected interpreter (shown in the status bar) is the one where OpenCV is installed.
* **Project Structure:** Add a new module or package for calibration in your PC app’s source code. For example, you might create calibration/ directory or a calibration.py file. Inside, implement the classes discussed (CalibrationManager, CalibrationProcessor, etc.). If your project is organized with a main GUI module, you may integrate the CalibrationDialog class in the UI package and keep the calibration logic in a backend module.
* **Dependencies:** Aside from OpenCV, ensure you have any GUI toolkit dependencies resolved (e.g., PyQt5/PySide if using Qt for the interface). The new UI elements (buttons, dialogs) will use the existing GUI framework. Set up signals/slots or callbacks for the button events (e.g., when “Capture Frame” is clicked, call a method in CalibrationManager).
* **Android App Readiness:** On the Android side, confirm that Milestone 2.8 (Calibration capture capability) is implemented: the socket communication should be running, and the app should be ready to capture images from both cameras. You might want to test the Android app separately to ensure it can capture an image from the thermal camera and RGB camera on demand. If not yet done, implement camera capture using the Camera2 API or the appropriate FLIR SDK for the thermal camera, and ensure the images can be retrieved in a useable format.

With the environment set up and all libraries available, you’re ready to implement and test the calibration engine step by step.

## Calibration Data Capture Process

The first phase of the calibration is capturing a dataset of corresponding images from the RGB and thermal cameras. This dataset will be used to calculate the calibration parameters. The process is interactive, involving the user, the PC app, and both phones. Below is the step-by-step workflow:

### 1. Initiate the Calibration Session

* **User Action:** The user clicks the “Calibrate Cameras” button in the PC app (or chooses the calibration option in a menu).
* **PC App:** The app opens the Calibration dialog/panel. In this dialog, explain the process to the user: for example, “To calibrate, place the calibration pattern (chessboard or marker board) in view of both the RGB and thermal cameras on each device. Capture at least 5-10 images from different angles. Press ‘Capture Frame’ each time you reposition the pattern.”
* **Device Selection:** If multiple devices (phones) are connected, the dialog might ask whether to calibrate both simultaneously or one at a time. Given that each phone is calibrated individually but the user said “both please” to capturing, the application can allow capturing from both devices at once for convenience. One approach is to calibrate one device at a time (making the user do two separate rounds), but a more efficient approach is:
* Capture frames from **both phones in each round**, since the calibration board can be visible to both devices simultaneously if arranged properly.
* The PC will accumulate two separate sets of images – one per device. Later, it will run calibration twice (once per phone).
* This way, the user only has to move the pattern around and capture, say, 10 times, instead of 10 times for phone A and another 10 for phone B (if both can see the pattern together, it halves the effort).
* **Preparing the Pattern:** Ensure the user has a calibration target. Typically, a checkerboard pattern (with known square size) printed on paper or cardboard is used. For thermal cameras, a plain printed chessboard might not show up because everything is at room temperature. One trick is to create a temperature contrast: for example, stick pieces of self-adhesive reflective tape for the black squares (so they have different thermal emissivity), or heat the board slightly so the pattern is visible in thermal. Alternatively, use an **ArUco marker board** (which has bold black squares that could be heated under light or have different thermal properties) or a ChArUco board (hybrid chessboard with ArUco markers)[[3]](https://www.sciencedirect.com/science/article/pii/S1350449524001038#:~:text=A%20geometric%20calibration%20method%20for,to%20be%20performed%20with). The PC software can support either pattern, but the default we'll assume is a chessboard unless configured otherwise. The pattern configuration (number of internal corners, square size) should be known to the software. For example, a common chessboard has 7x6 or 9x6 internal corners. We will use that in the OpenCV functions.

### 2. Capture Calibration Frames

This step is repeated multiple times to gather a variety of views of the pattern.

* **User Action:** The user positions the calibration board in a certain orientation (e.g., centered in view, or tilted), then clicks **“Capture Calibration Frame”** in the PC app.
* **PC App -> Phones (Socket Communication):** The CalibrationManager on the PC sends a capture command via the open sockets to each connected phone. For instance, it might send a JSON like {"cmd": "capture\_calibration"} to both devices. This communication is done over the network (Wi-Fi or USB network) using the existing socket connections.
* **Android App (Device) Behavior:** When each phone receives the capture command:
* It triggers its **RGB camera** to take a picture (if the camera is already streaming or previewing, it might grab the latest frame; otherwise, it may need to open the camera briefly to snap a photo).
* It also triggers the **thermal camera** to capture an image in quick succession. If possible, the app can capture both nearly simultaneously (e.g., by having both cameras open and grabbing frames). In some implementations, the thermal feed might always be running (some thermal cameras continuously deliver frames), so grabbing one frame from it at the same moment as the RGB capture is ideal.
* Once images are captured, the device prepares them for sending. The RGB image could be compressed to JPEG or PNG to reduce size (since it might be high resolution). The thermal image is smaller; it can be sent as PNG (which will preserve the grayscale values without loss or as little loss as possible).
* The device sends the image data back over the socket. This could be done by first sending a small header (indicating which camera and image size), followed by the binary image bytes. For example, phone A might send: MSG:RGB\_IMAGE; SIZE:12345 then 12345 bytes of JPEG data, then MSG:THERMAL\_IMAGE; SIZE:23456 then that many bytes of thermal image data. The PC side needs to reconstruct the images from these bytes. The communication protocol should ensure the PC knows when one image ends and the next begins (size fields or distinct messages help).
* **PC App Receiving Data:** The PC’s socket listener for each phone will receive the image bytes. The CalibrationManager should collect these and once it has both images from a device, pair them as one “calibration frame” for that device. For example:
* For device 1 (phone A), get frame1\_rgb.png and frame1\_thermal.png.
* For device 2 (phone B), get frame1\_rgb.png and frame1\_thermal.png. If capturing both devices at once, you effectively get two pairs of images for each capture click. The manager can store them in lists: e.g., calib\_images[device1] += [(rgb\_image, thermal\_image)] and calib\_images[device2] += [(rgb\_image, thermal\_image)]. It is important to keep images from the same moment together because they correspond to the same board position (this correspondence is what allows stereo calibration for that device’s cameras).
* **Feedback to User:** Once the images arrive, the PC app can update the UI:
* Increment the frame counter (e.g., “Captured 1 frame” for each device, or a combined count if doing both together).
* Optionally, show a small thumbnail of the captured images (especially the RGB image) so the user can verify the pattern was in view. We could even run a quick corner detection immediately and highlight if the pattern was found or not (for user feedback). If a pattern wasn’t detected on one of the images, we might alert the user to retake that frame (or simply not count that frame for calibration, see next step).
* The UI might display something like a list:
  + Frame 1: Device A ✔️ Pattern found, Device B ❌ Pattern not found (if one failed).  
    In case one device’s image didn’t see the pattern well (maybe it was out of frame on that device), the user could choose to discard that sub-frame or recapture. Since we are calibrating each device separately, it’s okay if one device misses a frame – we just won’t use that frame’s data for that device’s calibration. However, to maintain simplicity for the user, it might be better to ensure both devices see the pattern each time or calibrate separately to avoid confusion.
* **Repeat Captures:** The user should capture multiple frames. **Recommendation:** Aim for **at least 5 to 10 good frames per device** (OpenCV recommends a minimum of ~10 for reliable calibration[[4]](https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html#:~:text=the%20image%2C%20so%20we%20can,at%20least%2010%20test%20patterns)). More frames can improve accuracy, but diminishing returns after a point. The pattern should be moved around: sometimes at different corners of the camera’s view, at different distances, and orientations. For example, one frame with the board centered, one with it towards the left edge, one rotated 45 degrees, one closer, one farther, etc. Ensuring the pattern appears in different parts of the image helps the algorithm solve for lens distortion and focal parameters more robustly[[5]](https://answers.opencv.org/question/193623/calibration-between-thermal-and-visible-camera/#:~:text=10%20images%20,idea%20to%20change%20grid%20orientation). Also vary orientation to avoid all frames being planar in the same direction[[5]](https://answers.opencv.org/question/193623/calibration-between-thermal-and-visible-camera/#:~:text=10%20images%20,idea%20to%20change%20grid%20orientation).
* **Monitoring Progress:** The Calibration dialog should reflect how many frames have been captured. If calibrating both devices at once, ensure each device has sufficient frames. It might show “Device A: 6 frames, Device B: 5 frames captured.” The “Compute Calibration” button can become enabled only when each device has at least the minimum number of frames with detected patterns. (If one device has fewer, maybe ask for more captures visible to that device.) If doing one device at a time, then it’s simpler (just one count to track).

### 3. Complete Data Collection

* After the user has captured the required number of frames (say 10 frames for each phone’s cameras), they click the **“Compute Calibration”** button. At this point, the data capture phase ends and the processing phase begins. The UI should prevent further captures or disable capture buttons while computation is in progress (to avoid new images coming in mid-calculation).
* The collected images (for each device) are now ready to be processed by OpenCV routines to extract calibration parameters.

## Calibration Computation with OpenCV

With the calibration images in hand, the next step is to use OpenCV on the PC to calculate the camera parameters. This involves detecting the calibration pattern features in each image, assembling corresponding 3D-2D point sets, and running calibration algorithms. We will break this down into sub-steps:

### 4. Detect Calibration Pattern in Images

For each device, and for each captured image pair, we need to locate the calibration target in both the RGB and thermal images. This yields the image coordinates of known points on the pattern.

* **Convert to Grayscale:** If the RGB images are in color, convert them to grayscale using cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY) because the corner detection works on single-channel images. The thermal images might already be single-channel (if sent as grayscale). If the thermal image was pseudo-colored, we should convert it to grayscale as well (but ideally, we send it as raw grayscale to avoid losing the actual intensity differences).
* **Choose Detection Method:** Assuming a chessboard pattern is used, we apply OpenCV’s chessboard corner finder:
* Use ret, corners = cv2.findChessboardCorners(gray, pattern\_size) where pattern\_size is (columns, rows) of the **interior** corners (for example, a 7x6 chessboard has 7 columns and 6 rows of internal corners)[[6]](https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html#:~:text=Find%20the%20chess%20board%20corners). This function will return ret=True if it found the full pattern, and the corners array with the (x,y) pixel coordinates of each corner in order[[7]](https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html#:~:text=ret%2C%20corners%20%3D%20cv,None). If ret=False, the pattern wasn’t found in that image.
* If the chessboard has high contrast in the thermal image (e.g., heated black squares on cooler white background), findChessboardCorners can work on thermal images as well. If it struggles (due to low contrast or noise), an alternative is using **circles grid** or **ArUco markers**:
  + For a circles grid (e.g., array of black dots), cv2.findCirclesGrid could be used (requires a different pattern board)[[8]](https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html#:~:text=given%20are%20good,calibration%20using%20a%20circular%20grid).
  + For ArUco markers, OpenCV’s cv2.aruco.detectMarkers can find markers, and if using a ChArUco board, cv2.aruco.detectCharucoBoard can refine corner positions of a chessboard that has ArUco markers. This can be more reliable in thermal if the markers are warm or have distinct IR signatures. Implementing ArUco detection would require generating a board layout and using aruco.Dictionary and aruco.Board definitions. This is optional and can be added if needed for better detection (as the user indicated “add if needed”). In our initial implementation, we will try with the standard chessboard approach and see if it suffices.
* If a pattern is not found in one image (especially likely in some thermal images), you have a few options:
  + Drop that image from the calibration set (don’t use it in calculations).
  + Or, if one camera found it and the other didn’t, you can also drop the corresponding image from the other camera’s list to keep pairs consistent (especially important for stereo calibration, we need both views for each pattern pose).
  + The UI could notify the user and perhaps allow them to capture an additional frame to replace it. For simplicity, our calibration code will just skip any frame where detection failed on either image for that device. We want the same number of valid points sets for RGB and thermal.
* **Refine Corner Positions:** When the pattern is found, OpenCV typically gives corner coordinates with some pixel accuracy. We can refine these to sub-pixel accuracy for better results using cv2.cornerSubPix()[[9]](https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html#:~:text=if%20ret%20%3D%3D%20True%3A). This function takes the grayscale image and initial corner guesses, and iteratively fine-tunes the corner locations. We should do this for both RGB and thermal corner sets. (Thermal images might be lower resolution, but sub-pixel refining can still help if resolution allows.)
* **Store Image Points:** Save the 2D points for each successful detection:
* For each device’s RGB image list, build imgpoints\_rgb (a list of arrays of corner points).
* For each device’s thermal images, build imgpoints\_thermal. The order of points in each array corresponds to the pattern’s point ordering (usually left-to-right, top-to-bottom order as given by findChessboardCorners).
* Also prepare the matching 3D object points for each image:
  + Create objpoints list: each entry corresponds to one image (pattern pose), containing the 3D coordinates of the chessboard corners in the calibration pattern’s coordinate space. Typically, one assumes the chessboard lies on the z=0 plane (all points have coordinates (x, y, 0) in some unit). We can define the origin as one corner of the board. For example, if square size is 20mm, the corner points could be (0,0,0), (20,0,0), (40,0,0), ... etc for each intersection. If we don’t know the actual square size or don’t need the calibration in real-world units, we can set the square size to 1.0 (unit grid) – the calibration won’t know the difference, it will just yield focal length in “units per square” which is fine.
  + The same objpoints can be used for both cameras of a device because the pattern’s real coordinates for a given pose are the same; just viewed from two cameras. So for each successful frame (where both images had corners found), append the objp (the template of 3D points) to objpoints list (ensuring alignment of indexes with imgpoints lists).

After this step, for each device we have: - objpoints\_dev – a list of N arrays (N = number of frames used) of 3D points (all identical sets, just duplicated N times with possibly different orientations implicitly). - imgpoints\_rgb\_dev – a list of N arrays of 2D points from RGB images. - imgpoints\_thermal\_dev – a list of N arrays of 2D points from thermal images. - image\_size\_rgb and image\_size\_thermal – the image resolution for each camera (needed for calibration functions). If resolutions differ, note that OpenCV calibration still works; for stereo we might use one image size (usually the first camera’s size). Usually, you calibrate each camera with its own size; stereoCalibrate will expect both sets to have same image size if images were truly simultaneous of same scene. If RGB and thermal have different resolutions, OpenCV’s stereoCalibrate can still handle it by just specifying one (it will internally use the image coords relative to their own images). We will likely use the RGB image size for stereoCalibrate function call, since it asks for a Size parameter (assuming both images are same size – if not, might need to scale points or pad images; another approach is to resize one set of points but that distorts calibration, better to feed correct). Actually, cv2.stereoCalibrate expects points in their respective coordinate systems along with the camera matrices for those systems, so it *should* handle different resolutions as long as camera matrices correspond to those resolutions. To be safe, we could only use stereoCalibrate after undistorting or scaling, but that’s advanced – likely it’s fine as is if we pass full data correctly.

### 5. Calibrate Individual Camera Intrinsics

Before finding the relation between the two cameras, we will calibrate each camera on its own. This gives us the intrinsic parameters (focal lengths, principal point, distortion coefficients) for each RGB and thermal camera. Intrinsic calibration uses the 3D points and the 2D image points for each camera.

* **RGB Camera Intrinsic Calibration:** Using the objpoints\_dev and imgpoints\_rgb\_dev for a particular device, call cv2.calibrateCamera. For example:
* ret, cameraMatrix\_rgb, distCoeffs\_rgb, rvecs, tvecs = cv2.calibrateCamera(  
   objpoints\_dev, imgpoints\_rgb\_dev, image\_size\_rgb, None, None)
* This function returns the camera matrix (3x3) and distortion coefficients for that camera, along with rotation and translation vectors for each image (those rvecs/tvecs are the extrinsic pose of the pattern in each frame, not needed for now except maybe to compute error). The ret is the RMS reprojection error (in pixels) – a measure of calibration quality[[10]](https://temugeb.github.io/opencv/python/2021/02/02/stereo-camera-calibration-and-triangulation.html#:~:text=Then%20camera%20calibration%20can%20be,2). A lower number means the found parameters explain the observed image points well. Typically, an RMS error < 1.0 pixel is excellent[[10]](https://temugeb.github.io/opencv/python/2021/02/02/stereo-camera-calibration-and-triangulation.html#:~:text=Then%20camera%20calibration%20can%20be,2), up to a couple of pixels might be acceptable if resolution isn’t high or pattern detection had some noise. We will report this error to the user as an indicator of how good the calibration is.  
  *Note:* If the phone’s RGB camera specifications are known (sometimes we can get focal length from EXIF or Camera2 API), we could compare or even feed them as initial guess. But using OpenCV’s calibration from images ensures we have precise alignment relative to the thermal, so we’ll trust our computed values.
* **Thermal Camera Intrinsic Calibration:** Similarly, call cv2.calibrateCamera(objpoints\_dev, imgpoints\_thermal\_dev, image\_size\_thermal, ...) to get cameraMatrix\_thermal and distCoeffs\_thermal. The thermal camera likely has more distortion (if it’s a wide-angle lens on the thermal module) and a much lower resolution. Calibration here might be less accurate due to fewer pixels, but it’s still important. If OpenCV has trouble because the image is very low-res or the pattern points are very few, one strategy is to upsample the thermal images for detection – but since detection is done, calibration uses the pixel coordinates as is. The results will be in the thermal image’s pixel coordinate system.
* **Handle Calibration Failure:** If either calibration call fails or produces very high error, it means something went wrong (perhaps not enough valid points, or all points coplanar in a degenerate way). In such a case, the user might need to capture more frames or ensure the pattern was properly detected. We should catch exceptions or check the ret value. OpenCV calibrateCamera returns ret even if high; it rarely outright fails unless inputs are invalid.
* **Review Intrinsic Results (Optional):** For debugging or user interest, we can log or display the intrinsics:
* Camera matrix (for RGB might look like [[fx, 0, cx], [0, fy, cy], [0,0,1]]). If we used unit square size, fx,fy will be in “pixels” effectively. cx,cy should be roughly half the image width/height if the optical center is near image center.
* Distortion coefficients (array of 5 or 8 numbers depending on model used). Typically k1,k2 (radial), p1,p2 (tangential), k3, etc.  
  If the distortion coefficients are small (close to zero), the lens is near rectilinear; if larger, there is significant fisheye or wide-angle distortion. We can also compute and show the **reprojection error** for each camera. The formula involves projecting the object points back with the found parameters and comparing to detected points[[11]](https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html#:~:text=Re), but OpenCV’s ret is already an overall RMS error. A printout like “RGB cam reprojection error: 0.5 px, Thermal cam error: 0.8 px” is informative.
* **Use or Save Intrinsics:** Keep these intrinsic parameters ready. We will feed them into the stereo calibration next. It’s often recommended to **fix intrinsics during stereo calibration**, because optimizing everything at once can be unstable with limited data[[12]](https://temugeb.github.io/opencv/python/2021/02/02/stereo-camera-calibration-and-triangulation.html#:~:text=The%20cameras%20are%20first%20calibrated,for%20the%20stereo%20calibration%20case)[[13]](https://temugeb.github.io/opencv/python/2021/02/02/stereo-camera-calibration-and-triangulation.html#:~:text=Once%20we%20have%20the%20pixel,keep%20the%20camera%20matrices%20constant). So having reliable intrinsics now is good. We will not send these back to the phone unless needed (the user wasn’t sure if that’s needed; typically it’s not necessary for the phone to know its intrinsics in our application, since the PC does the analysis. We can always send them later if we want the phone to do something like annotate images, but for now, we’ll use intrinsics on PC side only).

### 6. Stereo Extrinsic Calibration (RGB-Thermal Alignment)

Now comes the core goal: finding the spatial relationship (extrinsics) between the RGB and thermal cameras of each device. This essentially tells us how to map coordinates from one camera to the other. OpenCV provides cv2.stereoCalibrate for this purpose.

* **Prepare Data for stereoCalibrate:** We need the same object points and the corresponding image points from both cameras for each frame. We have objpoints\_dev (list of points for each frame), imgpoints\_rgb\_dev, and imgpoints\_thermal\_dev. Ensure that these lists are all the same length N (we only include frames where both cameras had detections). Each index i in these lists corresponds to the same physical chessboard position for both camera views.
* **Call stereoCalibrate:** We already have cameraMatrix\_rgb, distCoeffs\_rgb, cameraMatrix\_thermal, distCoeffs\_thermal from the previous step. We use:
* flags = cv2.CALIB\_FIX\_INTRINSIC # we keep the intrinsics fixed  
  ret, camMatrix1, dist1, camMatrix2, dist2, R, T, E, F = cv2.stereoCalibrate(  
   objpoints\_dev, imgpoints\_rgb\_dev, imgpoints\_thermal\_dev,  
   cameraMatrix\_rgb, distCoeffs\_rgb,  
   cameraMatrix\_thermal, distCoeffs\_thermal,  
   image\_size\_rgb, criteria=criteria, flags=flags)
* Important details:
* We pass CALIB\_FIX\_INTRINSIC flag to not recompute the cameraMatrix values (just trust what we got)[[2]](https://temugeb.github.io/opencv/python/2021/02/02/stereo-camera-calibration-and-triangulation.html#:~:text=stereocalibration_flags%20%3D%20cv,criteria%2C%20flags%20%3D%20stereocalibration_flags). The algorithm will then only adjust extrinsics (R, T) to minimize reprojection error across both cameras.
* image\_size\_rgb is provided – OpenCV uses it for scaling maybe, but since each camera has its own matrix and points, it’s fine. If the thermal image size differs, OpenCV knows from cameraMatrix if needed. (There is a nuance: stereoCalibrate might assume both image sizes same; if not, one might scale points. In practice, one might resize the thermal image points by a factor if the thermal image was upsampled to match sizes. However, if each camera’s points are in its own pixel coordinate, and camera matrices correspond, it should work. If issues arise, an alternative is to run stereoCalibrate with normalized coordinates or rectify differently. But let’s assume OpenCV handles it or the thermal was maybe resized before detection – it could be we choose to upscale thermal images to a size similar to RGB for detection, which effectively scales the camera matrix too. For now, we’ll keep it simple and proceed.)
* The output gives us:
  + R (3x3 rotation matrix) and T (3x1 translation vector) that transform points from the RGB camera coordinate system to the thermal camera coordinate system[[14]](https://temugeb.github.io/opencv/python/2021/02/02/stereo-camera-calibration-and-triangulation.html#:~:text=The%20return%20values%20of%20the,rotation%20and%20translation%2C%20calculate%20as). In other words, if we take a point in 3D as seen by the RGB camera, we can rotate and translate it by R,T to get the coordinates as the thermal camera would see it. This is exactly the extrinsic calibration between the two cameras (basically the orientation and relative position of the thermal camera with respect to the RGB camera).
  + E and F are the essential and fundamental matrices (they’re by-products of stereo calibration, not directly needed for our usage, except perhaps for debugging or if we wanted to do stereo triangulation).
  + The function also returns refined camera matrices and dists (camMatrix1, camMatrix2, etc.), but since we fixed intrinsics, those should remain basically identical to what we input (the algorithm might still output them, but they shouldn’t change when fix flag is used).
* The ret from stereoCalibrate is another RMS error (this time for the stereo re-projection). We will check this error too. It might be a bit higher than the individual calibrations because it’s a combined fit. If it’s on the order of a few pixels, that could be okay given likely lower resolution of thermal; if it’s very high (like >5-10 pixels), then our calibration is poor (perhaps frames were not all good or pattern not flat, etc.).
* **Stereo Calibration Result Use:** The R and T are the main results. They can be interpreted as: the thermal camera’s position relative to the RGB camera. For example, T might be something like [3.0, 0.5, 0.1] in some unit (which would mean the thermal camera is 3 units to the right, 0.5 up, and 0.1 forward from the RGB camera if units = chessboard square size, which could be scaled to real-world if square size was in mm). R is a 3x3 rotation matrix; we could convert it to Euler angles just for understanding (e.g., a slight yaw/pitch if the thermal camera is angled relative to RGB).
* **Compute Homography for Overlay (Optional):** While R and T are the full 3D relationship, sometimes for overlaying one camera’s image onto another, a **homography** (planar perspective transform) is useful. A homography can map points from the thermal image to the RGB image *assuming those points lie on a particular plane in the scene*. If our subjects are relatively distant or we just want an approximate overlay, we can derive a homography that maps the thermal image pixels to RGB image pixels using the calibration data:
* One way: Use one of the calibration images where the pattern was detected. We know the correspondence of each chessboard corner in the thermal image (pixel coordinates) and in the RGB image (pixel coordinates). Using a set of corresponding points, we can call H, mask = cv2.findHomography(thermal\_points, rgb\_points) to compute a 3x3 homography matrix. This homography **only perfectly maps points on the plane of the calibration board**, but if the scene we later observe is roughly at a similar distance or flat, it can work as an overlay mapping.
* Another way: Since we have R, T and camera intrinsics, we can simulate a homography for a plane at a certain distance. For instance, if we assume an average scene depth or use the chessboard’s plane (z=0 in pattern coordinates), we can compute the projection matrices P1 and P2 for the two cameras and then derive a homography that maps image1 to image2 for that plane. This is more complex math, but effectively H = K\_rgb \* [R - (1/d)\*T\*n^T] \* inv(K\_thermal) where n is the plane normal and d is distance (for planar calibration object, we could solve it from a particular frame). This might be overkill; using findHomography empirically from points is simpler.
* We will implement the simpler empirical approach: take all the corner correspondences from all frames (a large set of 2D-2D matches) and run a single findHomography. Because the pattern moved, those points won’t lie on one plane in the *world* simultaneously, but each pair individually has its own plane. Using them all might give a least-squares homography that kind of averages the geometry. Alternatively, just choose one good frame’s data to compute H. It may not be perfect for all depths, but it’s a start for overlay.
* The overlay feature will use this homography to warp the thermal image onto the RGB image. Keep in mind, a homography means we assume a planar scene or fronto-parallel overlay; any parallax due to true 3D differences won’t be corrected. However, if the cameras are very close or the observed scene is far (relative to baseline), this is a reasonable approximation. (In real fusion systems, one might use depth data or project each pixel via R,T if a distance is known, but we lack depth here).
* **Calibration Data for Multiple Devices:** We repeat the above calibration (intrinsics + stereo) for each device independently. If we captured both devices concurrently, by now we would have run two separate sets of calibration calculations. Each yields its own intrinsics and extrinsics. The end result could be stored in something like calibration\_deviceA.yaml and calibration\_deviceB.yaml.

### 7. Storing Calibration Results

After successful computation, we need to save the calibration parameters for future use. This can be done automatically once the computation finishes, and/or on user command (e.g., clicking “Save”).

* **Data to Store:** For each device, store at least:
* cameraMatrix\_rgb (3x3 matrix) and distCoeffs\_rgb (vector) for the RGB camera.
* cameraMatrix\_thermal and distCoeffs\_thermal for the thermal camera.
* R and T (extrinsic transform from RGB to thermal coordinate frame, or vice versa – define clearly which it is; typically we got R,T taking RGB to thermal, but we should verify and perhaps store both directions or at least note it).
* If computed, the homography matrix H\_thermal\_to\_rgb.
* Perhaps the RMS errors: error\_rgb, error\_thermal, error\_stereo for record.
* Optionally, the resolution of images used (so we know the context of the intrinsics – though the cameraMatrix implicitly encodes that via focal length ~ pixels). If the app might run cameras at different resolution later, we might need to scale intrinsics accordingly, or simply always calibrate and use at a fixed resolution. It’s best to calibrate at the same resolution you’ll use for overlay, to avoid needing to adjust the calibration matrices.
* **File Format:** Use a convenient format like YAML or JSON. OpenCV’s cv2.FileStorage can write a YAML file easily with cv2 matrices. Alternatively, since we’re in Python, we could do:
* data = {  
   "cameraMatrix\_rgb": cameraMatrix\_rgb.tolist(),  
   "distCoeffs\_rgb": distCoeffs\_rgb.tolist(),  
   ...  
  }  
  import json  
  json.dump(data, open(filename, "w"))
* YAML (using FileStorage or even ruamel.yaml) is also human-readable. OpenCV can read its own YAML back if needed. We can name the file distinctly for each device (maybe using the device identifier or name). For example: calibration\_phoneA.yaml, calibration\_phoneB.yaml. If there’s only one device, a generic calibration.yaml is fine.
* **Internal Storage:** The CalibrationManager should also keep the results in memory (e.g., as a CalibrationResult object) so that the rest of the app can use it immediately (for overlay or other computations) without re-reading from disk.
* **Inform the User:** The UI can now display a summary:
* e.g., “Calibration completed for Device A and Device B. Reprojection error: RGB=0.5 px, Thermal=0.8 px, Stereo=1.2 px. Calibration data has been saved to calibration\_phoneA.yaml and calibration\_phoneB.yaml.”  
  If only one device, just one set of numbers. If the error is high or pattern detection was suboptimal, we might warn like “Calibration error is somewhat high; you may want to capture more frames or ensure the pattern covers the frame well.”  
  Also, let the user know that the calibration will be applied to future analysis and the overlay feature.
* **(Optional) Sending Calibration to Phones:** The user wasn’t sure if needed, but we can consider: If the phones themselves need to know the calibration (for example, to possibly do their own processing or just to store meta-data), we could send some information back. However, since the PC is doing the heavy work, it’s not strictly necessary. One scenario: if the phone records video, it might tag each frame with thermal-to-RGB alignment info so that if someone only had the phone data they could align it. This is an edge case. For now, we will **not send** anything to the phones; the PC will handle alignment when needed. We simply store it on the PC side.

At this stage, the system has completed the calibration calculations and stored the parameters. The final part is utilizing these parameters for enhanced functionality, like image undistortion and overlay, and integrating into the normal operation of the app.

## Utilizing Calibration Results (Undistortion & Overlay)

With calibrated parameters, there are a few immediate uses in our application: we can undistort images for a cleaner view, and we can overlay thermal imagery on RGB using the known alignment. These features improve the quality of data and the user’s ability to interpret it.

### 8. Image Undistortion (if needed)

If the cameras exhibit significant lens distortion (common in wide-angle or cheap lenses, including some thermal cameras), we may want to **undistort** images before further processing or display. OpenCV’s calibration output includes distortion coefficients which can be used to correct images: - We can pre-compute rectification maps using cv2.initUndistortRectifyMap for each camera[[15]](https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html#:~:text=This%20way%20is%20a%20little,Then%20use%20the%20remap%20function), or simply call cv2.undistort(image, cameraMatrix, distCoeffs) each time on the frames[[16]](https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html#:~:text=1). For real-time video, computing a map and using cv2.remap is more efficient.  
- If our main aim is overlay, it might be easier to undistort both images to a common rectified space. However, given the low distortion likely in the FLIR and phone camera (assuming not super fisheye), we might skip real-time undistortion for performance unless distortion is obvious. Alternatively, we can undistort the thermal image when overlaying onto RGB so that things line up better at the edges.  
- This is an optional improvement. The calibration dialog could have a checkbox “Apply undistortion” for the preview.

### 9. Thermal-RGB Overlay Feature

One of the exciting outcomes of calibration is the ability to **overlay** the thermal image on top of the RGB image in the correct alignment. We implement this as a real-time feature in the PC app’s main view, toggled by the user.

* **UI Toggle:** In the main UI (outside the calibration dialog, perhaps in the live view panel), add a checkbox or button “Overlay Thermal”. When checked, the app will combine the thermal feed with the RGB feed for display. When unchecked, the feeds are shown separately or only the RGB is shown normally.
* **Fetching Frames:** The app already likely receives video frames from the phones (either via streaming sockets or by grabbing periodic snapshots). We need to have access to the latest RGB frame and the latest thermal frame from a given device at each display refresh. Assuming we have those (perhaps the PC is already showing both feeds side by side), for overlay we will merge them.
* **Coordinate Mapping:** Using the calibration results, map the thermal image onto the RGB image coordinate system. There are a couple ways:
* **Using Homography:** If we computed a homography H\_thermal\_to\_rgb, we can apply cv2.warpPerspective on the thermal image. This will produce an output image that is the thermal image reprojected into the RGB camera’s view. For example:
* thermal\_aligned = cv2.warpPerspective(thermal\_image, H\_thermal\_to\_rgb, (w\_rgb, h\_rgb))
* where (w\_rgb, h\_rgb) is the size of the RGB image. The thermal image will be stretched/translated according to H. Note that areas of the RGB image that the thermal doesn’t cover will be black (we can later ignore those or only overlay where valid).
* **Using R, T (projection):** For a more accurate (but potentially slower) method, we could map each pixel by projecting rays. Essentially, for each pixel in the thermal image, we can convert it to a ray in the thermal camera’s coordinate system (in 3D, that ray goes out into space). Without depth, we don’t know where along that ray a given real scene point is. But if we make an assumption (like all objects of interest lie on a plane at some average distance, or simply use the calibration board’s plane for visualization), we could project that to the RGB. This quickly becomes complex without a depth map. So we will prefer the homography or simpler approximations.
* **Rectification approach:** We could also rectify both images to a common plane using stereoRectify and then overlay (commonly done for stereo camera alignment). cv2.stereoRectify given intrinsics and extrinsics can compute rectification transforms for each camera to a common perspective. If we rectified the thermal image to the RGB camera's perspective, that essentially does the same as warping it. However, stereoRectify assumes we might want parallel cameras for depth, which is not exactly needed for overlay (we actually want the RGB’s original perspective). We could set the RGB rectification to identity and only transform the thermal. This is effectively deriving a homography for an arbitrary fronto-parallel plane. It might be more than needed.
* **Blending:** Once we have the thermal image aligned in the RGB frame coordinates, we need to overlay it visually. Some options:
* Create a color map for the thermal image (for example, convert grayscale thermal intensities to a colormap like “JET” or a heatmap gradient) to make it easier to see differences. OpenCV has cv2.applyColorMap(thermal\_aligned\_gray, cv2.COLORMAP\_JET) to get a false-color thermal image. This colorized thermal can then be blended with the RGB image.
* Overlay by alpha blending: e.g., blended = cv2.addWeighted(rgb\_image, 0.7, thermal\_color, 0.3, 0), which would make the thermal semi-transparent over RGB. The UI can allow adjusting this alpha (like a slider from 0 to 100% thermal). The example ratio 0.7/0.3 is just a starting point.
* Alternatively, show thermal contours or outlines on RGB. But a simple blend is usually effective to highlight hot spots. The Luxonis example in the OAK documentation uses trackbars to adjust blending[[17]](https://docs.luxonis.com/software/depthai/examples/thermal_align/#:~:text=This%20example%20demonstrates%20how%20to,to%20adjust%20the%20blending%20ratio). We can implement a fixed or adjustable blend factor.
* **Performance Considerations:** Thermal images are small (often 160x120 or 320x240). Warping and blending at, say, 30 FPS is not heavy for a modern CPU. We should be fine doing this in real-time. If using Python, just ensure to avoid excessive data copying. Use NumPy arrays efficiently. We can do the warp and blend in the same thread that handles the UI frame update. If the UI library allows drawing overlays, we could even skip conversion to Qt images by drawing directly, but that’s specific to implementation. Possibly easier: get the RGB frame as a NumPy array, do blending to produce a composite image, then display that in the GUI widget (just as if it were a normal frame).
* **Validation of Overlay:** A good test of the overlay after calibration is to point the device at the calibration board again with overlay on – the thermal hot/cold pattern on the board should line up with the visible pattern. For example, if you heated the black squares, the thermal overlay should cover those black squares exactly on the RGB image. We can perform this test informally to verify calibration success.
* **Toggle Off:** When the overlay checkbox is off, we simply show the RGB image (and optionally the thermal image in a separate window or not at all). We should not stop receiving thermal frames just because overlay is off (we might still record them), but we don’t process them for display.

### 10. Real-Time Use of Calibration in Analysis

Beyond visualization, the calibration parameters can be used in any analysis algorithms we add: - For instance, if later we develop an algorithm to detect something in thermal and want to locate it in the RGB image (for object identification or tagging), we can use the calibration to transform coordinates. We could take the pixel coordinates of a hot spot in thermal, use the homography or full projection to find the corresponding RGB pixel, then perhaps draw a marker there in the RGB view.  
- If doing any 3D estimation or combining data from two phones, calibration would also be prerequisite (e.g., triangulating an object seen by both devices, though we have not planned cross-device calibration yet).

In summary, the calibration data is now actively improving the system: by correcting lens distortions, aligning two different sensor modalities, and enabling richer visual outputs.

## Testing and Verification Plan

Given the complexity of calibration, thorough testing at each step is important. Below are the recommended test checkpoints and methods:

* **Unit Test: Corner Detection** – Write a test function that takes a sample chessboard image (you can use one of OpenCV’s sample images or a synthetic one) and runs findChessboardCorners and cornerSubPix. Verify that the number of corners detected matches the expected pattern size and that the corners are in the correct order. You can draw the corners using cv2.drawChessboardCorners and display or save the image to visually confirm it’s detecting correctly. This ensures our corner-finding code works before integrating with the full pipeline.
* **Simulated Data Test** – If possible, use an existing dataset of images for calibration to test the algorithm. For example, OpenCV’s samples/data folder has stereo images of a chessboard (like left01.jpg, right01.jpg, etc.). Those are for two visible cameras, but for testing, treat one as “RGB” and the other as “thermal” camera images. Run the calibrateCamera and stereoCalibrate on those to see if the code yields sensible results (we know those images should produce a certain intrinsics and extrinsics). This helps validate our implementation of using the lists of points and calling the OpenCV functions properly.
* **Android-PC Integration Test (Single capture)** – Run the system with one phone connected. Click “Capture Calibration Frame” once. Check that:
* The phone receives the command (maybe logcat or internal logs can confirm).
* The phone sends back an RGB and a thermal image.
* The PC receives the images and correctly decodes them (check the arrays or save them to disk to ensure they look right).
* The UI updates the frame count and does not crash. If possible, integrate a debug option in the UI to display the captured image (like when you click on the thumbnail it opens a larger view). This helps verify that the image is good and the pattern is visible.
* **Multiple Frames Capture Test** – Continue the above by capturing, say, 5 frames. Make sure the count increases, and the app can handle multiple back-to-back captures. Test scenarios where the user presses capture quickly vs slowly, ensuring the system can handle the incoming data (it might be wise to disable the capture button until the previous images have been received and processed, to avoid overload). Also test the case where one of the captures might fail to find the pattern: simulate by capturing with the board out of view for one frame, and see if the software either warns or just skips it. Ideally, it should not crash; it should either not count that frame or mark it invalid.
* **Compute Calibration Test** – After capturing enough frames, press “Compute Calibration”. This will run the heavy OpenCV part. Observe the console or logs for any exceptions. It’s useful to print intermediate results for debugging, like the number of valid corners found per image, the calibration errors, etc. Check that the output values make sense (for example, focal length values in cameraMatrix should be roughly in the ballpark of the image size in pixels – e.g., for a 1920x1080 RGB image, fx might be ~1000-1500 if a moderate FOV lens). If any value is extremely off or any matrix is singular, something’s wrong.
* Specifically verify R and T: If the cameras are physically a few centimeters apart, T in units of the chessboard square (say 1 square = 30mm) might be something like (roughly 2-6 squares apart, i.e., 60-180mm, depending on how mounted). If T comes out huge or tiny, reconsider unit or input consistency.
* The reprojection error (ret) from calibrations should be checked. If you have stored all imgpoints and objpoints, you can independently compute the average reprojection error as well[[11]](https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html#:~:text=Re) to cross-verify OpenCV’s ret. This isn’t necessary for the final product but useful during testing to be sure we interpret results correctly.
* **Overlay Mapping Test (Offline)** – Without relying on real-time feed, test the overlay mapping using one of the captured calibration frames:
* Take one RGB image and the corresponding thermal image from the calibration set (which we know contain the chessboard). Use the computed calibration to warp the thermal image onto the RGB. Then draw the chessboard corners from thermal (after warp) onto the RGB image and see if they line up with the RGB corners. They should overlap closely if calibration is good. You can do this by:
  + Undistort both images with their intrinsics (optional).
  + Use R, T and camera matrices to project the chessboard’s 3D points into both images, see if they coincide with detected corners.
  + Or simpler: take the pixel corners from thermal image, and map them through the homography to RGB and compare to the RGB corners. If using homography from a different frame, this is approximate, but if you computed H from that frame’s points, then mapping thermal corners by H should exactly match RGB corners (since H was computed from them). This at least tests that correspondences and homography logic are correct.
* Also, test blending: take a known thermal image (could be just a grayscale heat pattern) and an RGB image, run your overlay code to ensure it produces an output image. Check for size mismatches or crashes (e.g., if thermal image is smaller, ensure warpPerspective doesn’t crash and output is correct size).
* **Live Overlay Test (Real device)** – After calibration, enable the overlay toggle while the system is running live. Point the device(s) at something with thermal contrast (your hand, a cup of hot water, an ice pack) and verify that the thermal overlay appears in the correct location on the RGB video. Adjust distances to see how parallax affects it – if you move very close to an object, you might see the thermal image shift slightly off (because our alignment is correct at one depth but can deviate at others). That’s expected. At moderate distance, it should be reasonably aligned. Ensure the blending looks good and the performance is acceptable (no major lag).
* Test turning the overlay off and on quickly, to ensure the app can switch modes without issues.
* If possible, test on both devices to ensure calibration for each is being applied independently. E.g., if you calibrated device A and B, then when viewing device A’s feed, it uses A’s calibration data, and same for B. If the app shows both feeds simultaneously, each with its overlay, ensure it doesn’t mix up the parameters.
* **Error Handling:** Try some edge cases: What if the user clicks “Compute Calibration” with zero frames captured? The app should handle that (likely the button is disabled until frames exist, but just in case). What if only 1 or 2 frames captured? OpenCV might throw an error or return nonsense – we should guard against running calibration with too few frames (set a minimum, like require >= 3 or 5).  
  Also, if the socket disconnects mid-calibration (e.g., phone goes offline), the app should handle it gracefully (though calibration can still proceed with already captured images – just notify that you can’t capture more).
* **User Guidance:** Finally, have a user (or yourself as tester) run through the entire calibration process following the UI prompts, without peeking into logs, to see if the instructions are clear and the flow makes sense. This may reveal if any step is confusing or if additional info is needed on screen (for example, the user might not know how far to place the board, etc., so maybe add tips like “Try to fill about half the frame with the board for some shots, and more distant for others”).
* **Documentation:** It’s useful to document the calibration process for end-users in a user manual. Summarize how to perform it, how long it takes, and how to tell if it succeeded. Also, instruct that re-calibration is needed if the cameras’ relative position changes (e.g., if the thermal camera module is remounted or the phone hardware changes).

By following these testing steps, we can be confident the calibration engine works reliably. Calibration is one of the most math-heavy parts of the system, so verifying each piece helps avoid frustration later on.

## Additional Tips and Future Considerations

* **Multiple Calibration Patterns:** We assumed a single chessboard target. In practice, some setups use a combination of a visible pattern and an IR pattern (for example, an **infrared-visible dual pattern**: one side printed with an IR absorbing material). If the current pattern proves hard for the thermal camera, consider alternatives: a pattern of black squares on aluminum (aluminum stays cooler under IR when illuminated), or simply a heated chessboard (e.g., print the pattern on a sheet and warm it with a hairdryer briefly so the black vs white have different temps). Ensuring the thermal contrast is key.
* **Using ChArUco Board:** A ChArUco board (chessboard with ArUco markers in the squares) can be beneficial. It allows detection even if not all corners are visible (markers help identify pattern) and can work with fewer images. OpenCV’s aruco module can calibrate using charuco with functions like cv2.aruco.calibrateCameraCharuco. This could be an improvement if our initial method has trouble. It does require printing a specific charuco pattern and possibly heating it too.
* **Cross-Device Calibration:** We focused on calibrating each phone’s cameras. The original milestone notes mentioned “potentially aligning multiple devices’ coordinate systems.” If in the future we need to know how Device A’s view aligns with Device B’s (for example, to triangulate an object seen from two different angles by the two phones), we would need a **cross-device calibration**. That would involve both devices seeing a common calibration object at the same time and performing a stereo calibration between, say, PhoneA’s RGB and PhoneB’s RGB. This is more complex (baselines are larger, synchronization is needed). The user here specified *not* to do that now (“Each phone individually”), which is fine. But keep in mind if that becomes necessary, a similar procedure can be done: place both phones so they see the same chessboard, capture simultaneous images on both, and run stereoCalibrate across devices. For now, we assume devices operate in their own coordinate space, and if needed the PC can relate them if an external reference is given later.
* **Automating Save/Load:** It’s a good idea to automatically load a saved calibration when the app starts (or when a device reconnects) so that if calibration was already done in a previous session, the user doesn’t have to redo it every time. The app can check for a calibration file corresponding to the device (maybe keyed by device serial or name). Perhaps provide a way in UI to manage multiple calibrations if using different devices. In this project’s context, if it’s mostly fixed hardware, one calibration per phone is fine.
* **Integrating with Recording**: When the system records data (video streams, etc.), consider logging the calibration info (or a reference to it) with the recordings. That way, if someone later analyzes the recorded data offline, they know what calibration to apply to align frames.
* **IDE Tips:** Use the IDE’s debugging features to step through the calibration code if needed. For example, after capturing images, you can set a breakpoint before the OpenCV processing to inspect the collected data (maybe even visualize points). Also leverage plotting libraries or OpenCV’s GUI functions during development (e.g., imshow) to debug alignment – but remember to remove or disable these in the final UI to avoid blocking calls.

With the Calibration Engine implemented, tested, and integrated, the system can now ensure all sensor data is spatially aligned. This significantly enhances the capability of the multi-sensor platform, enabling more accurate analysis, easier interpretation of thermal vs visual data, and a polished feature (thermal overlay) that provides at-a-glance insight to the operator. The groundwork is laid for any advanced features that rely on this alignment in subsequent milestones.

**Sources:** OpenCV documentation and community resources were referenced for best practices on camera calibration. OpenCV’s calibrateCamera provides intrinsics and distortion coefficients[[1]](https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html#:~:text=Now%20that%20we%20have%20our,rotation%20and%20translation%20vectors%20etc), and stereoCalibrate computes the rotation/translation between cameras when intrinsics are known[[2]](https://temugeb.github.io/opencv/python/2021/02/02/stereo-camera-calibration-and-triangulation.html#:~:text=stereocalibration_flags%20%3D%20cv,criteria%2C%20flags%20%3D%20stereocalibration_flags). It is generally advised to use at least 10 calibration images and vary the pattern placement for accuracy[[4]](https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html#:~:text=the%20image%2C%20so%20we%20can,at%20least%2010%20test%20patterns)[[5]](https://answers.opencv.org/question/193623/calibration-between-thermal-and-visible-camera/#:~:text=10%20images%20,idea%20to%20change%20grid%20orientation). The overlay approach is informed by common methods of warping one image to another using the homography or rectification, similar to examples in the Luxonis DepthAI documentation for RGB-thermal alignment (they blend images with adjustable alpha)[[17]](https://docs.luxonis.com/software/depthai/examples/thermal_align/#:~:text=This%20example%20demonstrates%20how%20to,to%20adjust%20the%20blending%20ratio). These references guided the implementation to ensure a robust and validated calibration process.

[[1]](https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html#:~:text=Now%20that%20we%20have%20our,rotation%20and%20translation%20vectors%20etc) [[4]](https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html#:~:text=the%20image%2C%20so%20we%20can,at%20least%2010%20test%20patterns) [[6]](https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html#:~:text=Find%20the%20chess%20board%20corners) [[7]](https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html#:~:text=ret%2C%20corners%20%3D%20cv,None) [[8]](https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html#:~:text=given%20are%20good,calibration%20using%20a%20circular%20grid) [[9]](https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html#:~:text=if%20ret%20%3D%3D%20True%3A) [[11]](https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html#:~:text=Re) [[15]](https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html#:~:text=This%20way%20is%20a%20little,Then%20use%20the%20remap%20function) [[16]](https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html#:~:text=1) OpenCV: Camera Calibration

<https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html>

[[2]](https://temugeb.github.io/opencv/python/2021/02/02/stereo-camera-calibration-and-triangulation.html#:~:text=stereocalibration_flags%20%3D%20cv,criteria%2C%20flags%20%3D%20stereocalibration_flags) [[10]](https://temugeb.github.io/opencv/python/2021/02/02/stereo-camera-calibration-and-triangulation.html#:~:text=Then%20camera%20calibration%20can%20be,2) [[12]](https://temugeb.github.io/opencv/python/2021/02/02/stereo-camera-calibration-and-triangulation.html#:~:text=The%20cameras%20are%20first%20calibrated,for%20the%20stereo%20calibration%20case) [[13]](https://temugeb.github.io/opencv/python/2021/02/02/stereo-camera-calibration-and-triangulation.html#:~:text=Once%20we%20have%20the%20pixel,keep%20the%20camera%20matrices%20constant) [[14]](https://temugeb.github.io/opencv/python/2021/02/02/stereo-camera-calibration-and-triangulation.html#:~:text=The%20return%20values%20of%20the,rotation%20and%20translation%2C%20calculate%20as) Stereo Camera Calibration and Triangulation with OpenCV and Python

<https://temugeb.github.io/opencv/python/2021/02/02/stereo-camera-calibration-and-triangulation.html>

[[3]](https://www.sciencedirect.com/science/article/pii/S1350449524001038#:~:text=A%20geometric%20calibration%20method%20for,to%20be%20performed%20with) A geometric calibration method for thermal cameras using a ...

<https://www.sciencedirect.com/science/article/pii/S1350449524001038>

[[5]](https://answers.opencv.org/question/193623/calibration-between-thermal-and-visible-camera/#:~:text=10%20images%20,idea%20to%20change%20grid%20orientation) Calibration between thermal and visible camera - OpenCV Q&A Forum

<https://answers.opencv.org/question/193623/calibration-between-thermal-and-visible-camera/>

[[17]](https://docs.luxonis.com/software/depthai/examples/thermal_align/#:~:text=This%20example%20demonstrates%20how%20to,to%20adjust%20the%20blending%20ratio) RGB-Thermal Align

<https://docs.luxonis.com/software/depthai/examples/thermal_align/>