## 5.1 Build Automation, Environment Bootstrapping, and Team Workflow

To efficiently develop and maintain this multi-component project (Android app + PC app), we will set up automation for building, smooth onboarding for developers, and a clear team workflow. The goals are to streamline common tasks (like building both apps together), ensure anyone can set up the dev environment easily, and establish collaboration practices for a monorepo with multiple developers. Key steps include:

* **Gradle Build Automation:** Create top-level Gradle tasks to streamline common actions. For example, define a Gradle task assembleAll that depends on the Android app’s assemble task (to build APKs) and also runs Python tests or packaging for the PC app. This can be done by using Gradle’s Exec tasks to call commands like pytest for the Python project. With this setup, a developer or CI can run ./gradlew assembleAll and build/test both subprojects at once. Ensure the Gradle Wrapper is included in the repo (choose a Gradle version (7.x or 8.x) compatible with all dev machines) and that everyone uses a consistent JDK version to avoid build issues.
* **Python Environment and Packaging:** Use a Conda-based workflow for the PC application’s environment. Provide an environment.yml file listing all Python dependencies (PyQt5, OpenCV, etc.) so developers can create the env with a single command (e.g. conda env create -f environment.yml). Using Conda ensures consistency across Windows machines (since the PC app is Windows-only). For now, running the PC app from source inside this environment is sufficient for our internal use. In the future, if distribution is needed, we plan to package the PC app (for example, using PyInstaller to create an .exe and perhaps an NSIS installer), but during development **running from source is enough**. The repository will include a simple requirements.txt or environment file for dependency management; no complex pip/Poetry setup, just Conda to keep things simple and consistent.
* **Environment Bootstrapping Script:** Provide a script (likely a PowerShell or Batch script for Windows) to automate initial setup on a new machine. This script can: install Miniconda (if not already present), create/update the Conda environment from the environment.yml, and ensure the Android build environment is ready (install the required Android SDK components via the command-line SDK manager, etc.). Listing prerequisites in the README (like “Install Android Studio 4.2+ and JDK 11, enable USB debugging on an Android device, etc.”) along with this script reduces friction for new developers. For example, a developer should be able to run setup\_dev\_env.ps1 and have everything needed installed and configured (Conda environment ready, Android SDK packages installed, Gradle wrapper configured).
* **IDE Configuration (Android Studio and Python IDE):** For Android development, use **Android Studio** to open the android-app project. Ensure that the correct SDK and NDK (if needed) are installed as per the project’s build.gradle (the project readme will list the required Android SDK version, build-tools, etc.). Android Studio will use the Gradle wrapper to import the project. Developers should configure Android Studio with a uniform code style (enable Ktlint or use Android Studio’s formatting rules defined by the project). For the Python desktop app, developers can use **PyCharm** or **VSCode**. In PyCharm, they should configure the project interpreter to use the Conda environment we set up (PyCharm can create from environment.yml or attach an existing environment). In VSCode, one should select the Conda environment as the Python interpreter in the bottom bar. We will include any necessary launch configurations (for example, a VSCode launch.json to run the PyQt main script, or PyCharm run configuration instructions) in the documentation. This ensures everyone can run and debug the PC app easily.
* **Team Branching Strategy:** We will adopt a clear Git workflow to manage changes in a monorepo. A recommended approach is using a main branch (or develop branch) where all completed features are merged, and separate **feature branches** for each task or feature. For example, one developer might work on feature/android-thermal-recording while another works on feature/pc-calibration-ui. Developers create Pull Requests to merge these into the main branch, and at least one other team member reviews the code (this helps share knowledge between Android and Python sides). We will tag significant milestones or releases (e.g., v0.1, v0.2) in Git so we have restore points. This branching strategy prevents developers from stepping on each other’s changes and keeps the main branch in a consistently working state.
* **Issue Tracking & Milestones:** We will use GitHub’s issue tracker (or an equivalent) to manage tasks and bugs. Each sub-task from the project plan can be an issue (e.g., “Implement 4K thermal video capture”, “Design calibration algorithm for sensor data”). We will group issues into Milestones corresponding to the project’s phases (Phase 1: Basic streaming, Phase 2: Calibration, etc.) so progress can be tracked easily. A GitHub Project board can further visualize task status (To Do, In Progress, Done). Regularly updating issues and linking commits/PRs to them will ensure traceability of changes. Everyone on the team will know who is working on what, and we can hold each other accountable for deadlines and deliverables.
* **Coding Standards and Pre-commit Hooks:** To maintain code quality across Android (Kotlin) and Python codebases, we will enforce consistent style and run static analysis. For Kotlin/Android, we integrate **Ktlint or Detekt** (possibly via Gradle plugins or a tool like Spotless) to automatically format code and catch common mistakes. For Python, we use **Black** (for formatting) and **Flake8/Pylint** for linting, plus **mypy** for type checking (as we add type hints). We can set up optional Git **pre-commit hooks** that run these linters/formatters before code is committed or pushed. This ensures the codebase stays clean and consistent (though these hooks can be bypassed in emergencies, we’ll encourage their use). We will document the process to install these hooks (possibly using the pre-commit framework or a simple script that symlinks a git hook). Consistency in code style makes it easier for everyone to read and review each other’s code.
* **Documentation and Knowledge Sharing:** Maintain up-to-date documentation in the repo. We’ll create a README.md for high-level usage instructions and separate docs (or a docs/ folder) for in-depth guides. For example, an **ARCHITECTURE.md** can outline the system design and each component’s role (this can include the class/module breakdowns below), and a **SETUP.md** can provide environment setup and troubleshooting tips. Additionally, each major module should have comments or even a short markdown explaining how to extend it (for instance, how to add a new sensor or change the calibration formula). New developers or collaborators should be able to read these and quickly understand how the project is structured and how to get it running. We’ll also encourage writing docstrings in the Python code and KDoc in Kotlin for important functions, which helps IDEs show hints and helps future maintenance. Good documentation ensures that the development process is not slowed down when team members rotate or new members join.
* **Collaboration and Testing Culture:** Foster a culture of open communication. We will have short regular meetings (e.g., weekly sync-ups or daily stand-ups if feasible) to discuss progress and blockers. Because this project spans mobile and desktop and deals with hardware (camera, sensor), integration testing is important. We plan to do end-to-end integration tests periodically: for example, once a week the team can assemble the hardware (thermal camera + Android device + PC) and run the full system to catch any integration issues early (like connectivity problems, calibration drifts, performance hiccups). This complements the automated tests. Team members should freely share interim results or problems on a common channel (Slack or Teams) so that knowledge is spread (e.g., Android dev can assist the PC dev if the issue actually originates from the app, and vice versa). By establishing these practices and tools, the development process will be smoother and more predictable. New developers can get the project running in minimal time, and consistent workflows will improve code quality and reduce “it works on my machine” problems.

## 5.2 Continuous Integration (CI) for Android and Python

We will set up Continuous Integration pipelines (using **GitHub Actions** exclusively) to automatically build and test the project on every push or pull request. This ensures code health and catches regressions early. Our CI will cover both the Android app and the Python desktop app, likely using a **matrix of runners** (e.g., Windows for the PC tests and Ubuntu or Mac for Android build, or just Windows for both if that suffices). The CI configuration will include:

* **Android CI Workflow:** A GitHub Actions job will be configured to build and test the Android application. Key steps in this job:
* **Set up JDK and Android SDK:** Use the GitHub Actions actions/setup-java action to install the correct JDK (matching our Android Gradle plugin requirements, e.g., JDK 11 or 17). Then install the Android SDK components. This can be done by using the Google provided command-line tools – for example, download and unarchive the SDK tools, then run sdkmanager to install the Android API level, build-tools, and emulator (if needed). There are also community actions that simplify this (like android-actions/setup-android which can install SDK packages). We will specify in the workflow which SDK level and build-tools version to install (matching the app’s build.gradle).
* **Build the APK:** Run the Gradle build. For example: ./gradlew assembleDebug (to compile the debug APK). We might also build the release variant (assembleRelease) if we want to ensure release builds succeed too. Gradle will output the APKs in android-app/app/build/outputs/apk/…. We can configure caching for Gradle and Android build outputs to speed up subsequent CI runs (GitHub Actions cache can store Gradle caches between runs).
* **Run Unit Tests and Lint (Android):** After building, execute ./gradlew testDebug to run any local unit tests for the Android code. These would include any JUnit tests for utility classes or data processing in the Android app. We will also run Android Lint via ./gradlew lintDebug. Android Lint can catch common mistakes or bad practices in the Android code. If we have configured Detekt or Ktlint, those can be run here as well (e.g., ./gradlew ktlintCheck). If any of these steps fail (tests failing or lint finding issues classified as errors), the CI job will fail, alerting us to fix the problems before merging.
* **(Optional) Instrumentation Tests:** If we write any Android instrumentation tests (UI tests or integration tests that require an emulator), we could set up an emulator on CI and run them (./gradlew connectedDebugAndroidTest). This is doable on GitHub Actions (there are workflows to start an AVD), but it adds significant time. Initially, we may skip this or run instrumentation tests locally, focusing CI on unit tests which are fast.
* **CI Artifacts:** The CI can be configured to upload build artifacts for convenience. For example, the debug APK (and release APK if built) can be attached as workflow artifacts. This way, anyone can download the built app from the CI run, and we have a record of the binary for that commit. This isn’t critical for internal development but can be handy for quick testing of a PR without building locally.
* **Python CI Workflow:** Another job (running on a Windows runner, since our GUI app is Windows-specific) will handle the Python side. Steps for the Python CI job:
* **Set up Python & Dependencies:** Use the Conda setup in CI. For instance, we can use a GitHub Action to install Miniconda on the runner, then run conda env create -f environment.yml to create the environment with all needed packages. This ensures the CI uses the same environment spec as developers. (Alternatively, for speed, we might maintain a requirements.txt and use pip install, but since we decided on Conda only, we’ll use the environment file approach. We can cache the Conda environment or the package downloads to speed up builds.)
* **Run Tests (PyTest):** Activate the conda environment (conda activate <env\_name> in the script), then run pytest to execute all unit tests for the PC application. We will write tests for things like the calibration math, image processing functions, and any protocol/communication code (using maybe dummy data to simulate the camera feed). The tests should be designed to not require a GUI display (for example, if testing GUI logic, use Qt’s QTest framework or simply test the underlying logic without actually showing windows). If some tests do require a display (e.g., if we instantiate a QApplication), we can use a virtual frame buffer or simply skip those in CI. The test suite should output a result and if any test fails, the CI job fails.
* **Code Quality Checks:** We will also run linting/type-checking in CI. For example, run flake8 or pylint on the Python code to ensure style guidelines are met. Also run mypy for static type checking (if we have type hints). This helps catch bugs like functions mis-typed or mismatched use of data structures early. Any errors from these tools should also fail the CI. Over time, these tools enforce a baseline of code quality (e.g., no unused imports, consistent naming, etc.).
* **Build Distribution (Optional):** If we decide to package the PC app (as mentioned, using PyInstaller to create an executable), we can add a CI step to build that package. For instance, run pyinstaller main.spec to produce an EXE. The CI can then archive the resulting EXE or installer as an artifact. This ensures that our packaging process is tested and reproducible. Even if we are mostly running from source during development, having a CI job that occasionally builds the standalone app is useful for catching any issues in that process (missing files, etc.) well before we actually need to deliver the software. Since the team indicated packaging **will be needed**, we will integrate this step once the app is stable, so we have ready-to-use packages for end users or demo sessions.
* **Integrating Workflows in GitHub Actions:** We will likely have a single YAML workflow file with **multiple jobs** (one for Android, one for Python, possibly more). They can run in parallel on different runners. We will set this workflow to trigger on every push and pull request to main or develop branches. We might also use path filters to skip certain jobs if files not related to that component changed (for example, if only Python files changed, skip the Android build job, and vice versa), to optimize CI time. Initially, running both for all commits is fine (given moderate project size). The CI will give quick feedback: if a developer pushes a change that breaks the Android build or makes a Python test fail, everyone will see a red X on the commit/PR and can address it before merging.
* **Continuous Delivery (Release Automation):** In addition to per-commit CI, we will set up workflows for releases. For example, when we create a Git tag like v1.0, a special GitHub Actions workflow can be triggered to build the Release APK and the packaged PC app, then attach those artifacts to a GitHub Release. This way, creating a new version for the stakeholders is straightforward – just tag the commit and let CI handle the rest (building, signing if applicable, and bundling outputs). This is not needed during early development, but planning for it will save time later when we need to deliver a usable system to the end users (the research team). We might not automate deployment to an app store (since this is a research tool), but having downloadable binaries from GitHub is useful.
* **CI Badges and Notifications:** We will add status badges in the repository README to show the build status (e.g., ![CI Status](data:text/plain; charset=utf-8;base64,Tm90IEZvdW5k)  
  ). This gives everyone quick insight into whether the main branch is currently passing all tests. Additionally, we’ll configure notification settings: for instance, if using GitHub, it can notify via email or integrate with Slack/Microsoft Teams when a build fails. The team should treat a failing main branch build as a high-priority issue – it means something is broken in integration. By keeping an eye on CI, we maintain a healthy codebase.
* **Quality Gates:** We will enforce that all tests and checks must pass before code is merged. In GitHub, this means making the CI checks required for merging pull requests. This encourages developers to run tests locally *before* pushing (to avoid broken builds) and ensures that nothing gets into the main branch that is failing. Over time, we might expand the CI with more quality gates – for example, set a code coverage threshold that must not drop. Initially, the focus is simply “no failing tests or linter errors”. This policy keeps technical debt low and prevents “works on my machine” issues. It effectively makes CI a team member that reviews every change for basics.

In summary, setting up CI with GitHub Actions for both the Android and Python components will greatly enhance our development workflow. Every commit will be verified by building the app and running tests in a fresh environment, giving confidence that the project remains buildable and that we catch integration issues early. It also simplifies onboarding (new devs can trust if main is green, they can pull and build without issues) and prepares us for smoother releases. CI is crucial especially in a cross-platform, multi-language project like this, acting as an automated guardian of code quality and functionality.

## 5.3 Developer Setup, Code Structure, and Testing Checkpoints

Finally, we provide a breakdown of the project’s structure and outline the steps for developers to set up and verify the system. This section serves as a technical guide for team members to understand the codebase organization, configure their development environment, and perform tests at key checkpoints to ensure each part of the system is working as expected.

* **Project Structure and Key Modules:** The repository is a monorepo containing two main subprojects – the Android app and the Python PC app – along with configuration files and docs. Below is an overview of the structure and important classes/modules in each part:

**Android Application (Kotlin + Android SDK)**  
- app/src/main/java/com/ourproject/thermal/MainActivity.kt: The main Android Activity that initializes the camera view and handles user interface on the phone. It sets up the preview from the thermal camera and possibly a normal camera feed if needed. It also starts the networking component to stream data to the PC.  
- app/src/main/java/com/ourproject/thermal/CameraManager.kt (or similar): A class responsible for interacting with the Camera2 API (or FLIR SDK if a FLIR One is used). It configures the camera resolution (4K thermal imaging, etc.), frame rate, and obtains frames in a background thread. This class might also handle any image preprocessing (like converting the camera frames to a suitable format for sending).  
- app/src/main/java/com/ourproject/thermal/NetworkSender.kt: This component manages the network communication from the Android device to the PC. For instance, it could open a WebSocket or UDP socket to the PC’s IP and send the thermal image frames (possibly as byte arrays or encoded images) along with sensor metadata (e.g., timestamp, and any reference temperature readings). It ensures efficient streaming (perhaps compressing frames or sending at a controlled rate to avoid flooding).  
- app/src/main/java/com/ourproject/thermal/CalibrationUtils.kt: (If calibration or some calculations are done on phone) A utility class that might package the calibration data (for example, reading device sensors or applying any correction to the raw thermal data before sending). In our design, most calibration is on the PC side, but the phone might still tag frames with some IDs or preliminary computations.  
- Other supporting classes: e.g., Settings.kt for app configuration (if users can adjust settings like IP address of PC, frame rate, etc.), and perhaps a DataModel.kt representing the data being sent (frame plus temperature info). If the app is complex, we might employ an MVVM architecture with ViewModel classes, but given the scope, a simpler structure with an Activity and a few helpers (as listed) is likely sufficient.

**Python Desktop Application (PyQt5 + supporting libraries)**  
- main.py: The entry point of the desktop application. This script likely creates a QApplication and instantiates the main window. It also sets up the network listener to start receiving data from the Android app. Developers run this to launch the GUI.  
- gui/MainWindow.py: A module defining the main GUI window (probably a subclass of QMainWindow). This class sets up the UI elements – e.g., a video display widget to show the thermal feed (could be a QLabel or a QGraphicsView that we update with incoming frames), and maybe labels or plots to show temperature readings or battery status. It may also include a matplotlib or PyQtGraph component for real-time graphing of temperature data if required. The MainWindow coordinates between the user interface and the backend logic.  
- network/Receiver.py: This module handles the network communication on the PC side. For instance, it opens a socket server (WebSocket server or UDP listener) that the Android app connects to. It continuously listens for incoming frames and data. Upon receiving a frame (which might be an image or byte array), it can convert it to an OpenCV image or QImage. This class likely runs in a separate thread or uses asynchronous callbacks so that it doesn’t freeze the GUI. It then passes the data to the GUI (using signals/slots in PyQt to safely update the UI).  
- calibration/CalibrationEngine.py: This contains the logic for applying calibration to the thermal data. For example, if we have a reference temperature sensor reading (say the Android sent over the current ambient or a blackbody temp), this module uses that to adjust the raw thermal image pixel values (perhaps converting raw sensor readings to actual temperatures, or correcting for drift). It might also handle alignment if the thermal image needs alignment with a visual image (if the phone also sent a normal camera frame for overlay). Functions here take in raw data and output corrected temperature values or corrected images. This module will be heavily tested with unit tests to ensure the math is correct.  
- analysis/DataLogger.py (optional): If the app logs data to disk (thermal frames, temperature readings, etc. for later analysis), this module would handle file I/O. For instance, saving incoming frames as images or recording temperature vs. time to a CSV. This is not core to functionality but useful for research, and would be invoked by the GUI when the user starts/stops a recording session.  
- utils/Helpers.py: A catch-all for small helper functions (like color map conversions for the thermal image, unit conversions, etc.).  
- **Note:** The Python project might be structured as a package (with an \_\_init\_\_.py), but given it’s an internal tool, we might keep it simple. However, we will organize by folders as hinted (gui/, network/, calibration/, etc.) for clarity. Each of these modules will have corresponding unit tests (e.g., tests/test\_calibration\_engine.py for CalibrationEngine, etc.). This modular breakup ensures different aspects (GUI vs. logic vs. networking) can be developed and tested somewhat independently.

* **Developer Setup Steps:** Setting up a new development environment is straightforward thanks to the automation:
* **Prerequisites:** Install Android Studio (with recommended version and SDK), and Miniconda (for Python). Also, ensure Git is installed and you have access to the repository. If using PyCharm or VSCode for Python, have those installed as well.
* **Clone the Repository:** git clone https://github.com/ourteam/thermal-project.git (use the actual repo URL). Navigate into the project directory.
* **Run Environment Setup Script:** Execute the provided setup script for your platform. For example, on Windows, open PowerShell and run .\setup\_dev\_env.ps1. This will:
  + Install Miniconda (or use existing) and create the Conda environment named (say) thermal-env using the environment.yml. (This may take a few minutes as it downloads PyQt, OpenCV, etc.)
  + Install required Android SDK components. The script might call Android’s sdkmanager to ensure you have the API (for example, API 33) and build-tools (for example, 33.0.2) that the project needs. You might be prompted to accept Android SDK licenses.
  + It could also assemble the Android project once to verify Gradle is working (or this can be done manually in Android Studio).
* **Open the Projects in IDEs:**
  + **Android Studio:** Choose “Open an existing project” and select the android-app folder. Android Studio will import the Gradle project. If prompted, allow it to download Gradle or Android SDK parts as needed. Once open, you should be able to run the app: connect an Android device via USB (ensure USB debugging is enabled on the device), click the Run button in Android Studio, and it will build and install the app on the device. A quick sanity check is to see the camera view on the phone and ensure no runtime errors.
  + **Python IDE/Editor:** If using PyCharm, go to Settings -> Python Interpreter, and add the Conda environment (thermal-env) that was created. Mark the python-app directory as a Sources Root if needed. You can then open main.py and run it (PyCharm will use the selected interpreter). If using VSCode, open the repository folder in VSCode. It should detect it’s a Python project; select the thermal-env interpreter. You might also install the **Python** and **PyQt5** VSCode extensions for better experience. To run the app, you can simply run python main.py in a VSCode terminal (after activating the environment) or use a launch configuration.
* **Build All Components (optional unified step):** You can also run the Gradle multi-build: from the root of the repo, execute ./gradlew assembleAll. This should trigger the Android build and run Python tests. Check that this finishes successfully. (The first run might take time to download Gradle dependencies and such.) This is a good smoke test that your environment is set up correctly.
* **Testing Checkpoints:** As development progresses, there are several points at which to test functionality to catch issues early:
* **Unit Tests:** We will write unit tests for critical modules (especially the Python logic like calibration). Run pytest frequently during development (or use your IDE’s test runner). If all tests pass, it gives confidence your recent changes didn’t break core logic. Android code can also have local unit tests (run with gradlew test or via Android Studio’s test runner). For example, if we add a math function for temperature conversion on Android, write a small JUnit test for it.
* **Integration Test – Connectivity:** Once the Android app and PC app can run, do a manual integration test. Launch the PC application (python main.py) on a PC and start the Android app on a phone. Ensure the phone and PC are on the same network. When the Android app starts streaming, the PC app should receive data. You might see the thermal video feed appearing in the PC app’s window. Check that the latency is reasonable and there are no connection errors in the logs. This tests the networking pipeline end-to-end. We should do this test whenever there are significant changes to the networking code or after any long gap in development to make sure nothing inadvertently broke the connection protocol.
* **Integration Test – Calibration:** Test the calibration feature by using a known temperature source. For instance, have a thermometer or known reference and the thermal camera both measure it. The PC app should display temperature readings that match the reference (within expected error). If we have a reference sensor integrated, ensure the data from that sensor is correctly influencing the thermal image (e.g., if we point the camera at an object of known 50°C, and the reference sensor also reads 50°C, after calibration the PC app should show around 50°C for that object). Perform this test after implementing the calibration logic, and whenever changes are made to it. It can be a manual procedure but is vital for verifying the scientific accuracy of our system.
* **Performance Test:** As a checkpoint, monitor the performance (FPS of the video, CPU usage on the PC and phone, etc.). For example, after implementing 4K video streaming, check on a typical dev machine that the PC app can handle the frame rate without lag, and the phone isn’t overheating. If issues are found, we may need to tweak settings (lower frame rate or resolution, or optimize code). It’s easier to catch performance bottlenecks early rather than later.
* **Continuous Integration Results:** Always check the CI status after pushing commits or merging PRs. If the CI flags a test failure or lint issue that was missed locally, fix it promptly. Treat CI as an additional set of tests – for example, CI might run on a fresh environment and catch a missing dependency or a platform-specific issue. Before major milestones or releases, ensure CI is fully green.

By following this developer setup guide and utilizing the testing checkpoints, each team member can confidently contribute to the project. The class/module breakdown gives an overview of where to find or place certain functionality, and the environment setup + CI guarantees that if something works on one machine, it will work on others. In summary, Milestone 5’s steps solidify the project’s foundation: automating builds, enabling easy environment replication, enforcing quality through CI, and establishing a robust team workflow. This allows us to focus on delivering the features (high-quality thermal imaging and calibration) without getting bogged down by integration issues or “it works on my machine” syndrome. Every part of the system, from code to collaboration practices, is now structured for efficiency, consistency, and reliability.