# Monorepo Setup for Android (Kotlin) + Python (PyQt5) in Android Studio (Windows)

## 1. Project Goals and Monorepo Rationale

This project combines an **Android mobile app** (Kotlin, Camera2 API, Shimmer sensors, USB thermal camera SDK) with a **Python desktop controller app** (PyQt5 UI, OpenCV for calibration, and socket networking). Using a single **monorepo** ensures both components stay in sync – any changes to the network protocol or data formats can be updated in one commit and tested together. It simplifies collaboration: developers pull one repository and have everything needed. The goal is to manage both apps from **Android Studio on Windows** for a unified development environment. Android Studio (based on IntelliJ IDEA) will handle editing, building, and running *both* the Android and Python code. Using a Gradle multi-project setup, we leverage Gradle’s flexibility to handle multiple modules (it’s primarily a JVM build tool but **supports many other languages as well**[[1]](https://earthly.dev/blog/gradle-monorepo/#:~:text=Gradle%20is%20a%20powerful%20and,many%20other%20languages%20as%20well)) and to automate tasks for the Python side.

Key objectives:  
- **Single IDE (Android Studio)** for Android + Python to streamline development on Windows.  
- **Gradle multi-module project** to organize Android and Python codebases under one roof.  
- **Consistent tooling**: Gradle for builds and tasks, plus Python virtual environments (venv) for the Python app’s dependencies.  
- **GitHub integration**: easy CI setup and a single repo for issues and version control, improving team collaboration (especially for Windows developers using Android Studio).

## 2. Repository Layout and Folder Structure

We will use a **Gradle multi-project** (monorepo) structure with one root project and two sub-modules: one for the Android app and one for the Python app. The layout is designed so Android Studio can open the whole repository as one project. Below is the proposed folder structure (monorepo root with two main subfolders):

project-root/   
├── settings.gradle # Gradle settings: includes both modules   
├── build.gradle # Root Gradle build (configuration for all modules)   
├── gradle/ wrapper/ # Gradle Wrapper files (gradle-wrapper.properties, jar)   
├── gradlew & gradlew.bat # Gradle wrapper scripts for Unix/Windows   
├── AndroidApp/ # Android app module (Kotlin + Camera2, Shimmer, etc.)   
│ ├── build.gradle # Gradle build for Android module (com.android.application)   
│ ├── src/ # Source set for Android code (Kotlin files, resources, etc.)   
│ │ └── main/   
│ │ ├── AndroidManifest.xml   
│ │ ├── java/... # Kotlin source packages   
│ │ └── res/... # Android resources (layouts, values, etc.)   
│ └── ... (other Android config files like proguard, etc.)   
├── PythonApp/ # Python desktop app module (PyQt5, OpenCV)   
│ ├── build.gradle # Gradle build for Python module (uses Python plugin)   
│ ├── src/ # (Suggested) Python source files   
│ │ ├── main.py # Entry-point script for the PyQt5 app   
│ │ └── ... (other .py modules, e.g., calibration utils, network code)   
│ └── requirements.txt (optional) # (If not listing deps in build.gradle, else omit)   
└── .gitignore # Git ignore file for Android & Python artifacts

**Layout rationale:** The Android module and Python module reside side by side in the repository. Gradle’s **settings.gradle** will include both modules so they load together in Android Studio. Each module has its own build.gradle for module-specific build logic. The Android module follows the standard Android Studio structure. The Python module contains the Python source; here we use a simple src/ folder, but you can organize it as needed (the Gradle plugin will allow running any script path).

This structure allows Android Studio to treat AndroidApp as a typical Android application module and PythonApp as another module (with custom tasks). Gradle sees one project with two subprojects. As the Earthly Gradle monorepo guide notes, *the settings.gradle file tells Gradle which subprojects are part of the monorepo and where they are located*[[2]](https://earthly.dev/blog/gradle-monorepo/#:~:text=contains%20all%20of%20the%20subprojects,and%20where%20they%20are%20located). Each subproject has its own code and build config, but they live in one repo for easy management.

## 3. Android Studio Project Setup (Gradle Multi-Project)

Next, set up the Android Studio project to recognize this multi-module structure. We assume you’re starting from scratch or integrating an existing app:

* **Create the Android module:** In Android Studio, create a new project (e.g. “MyMonorepoProject”) with an **Android app module**. For example, choose the *Empty Activity* template in Kotlin. This will create an Android app (by default named “app”). You can use that as the AndroidApp module. If the wizard names it app/, you can rename it by closing Android Studio and renaming the folder to **AndroidApp** (update settings.gradle accordingly). The Android module’s build.gradle will be the standard one created by Android Studio (with com.android.application plugin, compileSdk version, default config, etc.). Verify you can build and run this Android app module by itself first (e.g., run on an emulator a “Hello World” activity).
* **Add the Python module to the Gradle settings:** Open the project’s **settings.gradle** (in the root). You will see it already includes the Android app (e.g., include ':AndroidApp'). Now include the Python module by adding:

include ':AndroidApp', ':PythonApp'

This registers **PythonApp** as a subproject. Gradle now knows about both modules[[3]](https://earthly.dev/blog/gradle-monorepo/#:~:text=mkdir%20subproject). After editing settings.gradle, hit **“Sync Project with Gradle”** in Android Studio. The IDE will pick up the new module (though we haven’t configured its build yet, we’ll do that next).

* **Define the Python module’s Gradle build:** Create a new **PythonApp/build.gradle** file to configure the Python app module. We will use a Gradle plugin to integrate Python. In the Python module’s build.gradle:
* Apply the **Gradle Use-Python plugin**. This plugin (ID ru.vyarus.use-python) will allow Gradle to manage a Python virtualenv and pip packages for us. Add at the top of PythonApp/build.gradle:
* plugins {  
   id 'ru.vyarus.use-python' version '3.0.0'  
  }
* *(Gradle will download this plugin from Maven Central on sync. Make sure the root build.gradle or settings plugin management includes Maven Central. Alternatively, use the older buildscript {} block to add the classpath, but the plugins DSL is cleaner.)*
* Configure Python details in Gradle: We need to tell the plugin which Python packages our app needs. The plugin will automatically create a **virtual environment** and install these. For example:
* python {  
   // Use an existing Python on the system (default). Optionally specify path:  
   // pythonPath = "C:/Python39/python.exe"  
   // List required pip packages (name:version):  
   pip 'pyqt5:5.15.7'  
   pip 'opencv-python:4.8.0.74'  
   pip 'numpy:1.24.3'  
  }
* Here we specify exact versions (version ranges are *not* allowed to ensure reproducible builds[[4]](https://xvik.github.io/gradle-use-python-plugin/3.0.0/getting-started/#:~:text=Important)). You would list all Python dependencies (PyQt5 for the GUI, OpenCV for image processing, numpy if needed, etc. – plus any others your controller app needs). When Gradle syncs or when we run a Python-related Gradle task, it will ensure these are installed in the virtualenv. By default, the plugin will create a virtual environment under PROJECT\_ROOT/.gradle/python and install the packages there, **sharing one venv for all modules** in the project[[5]](https://xvik.github.io/gradle-use-python-plugin/3.0.0/guide/multimodule/#:~:text=When%20used%20in%20multi,same%20environment%20for%20all%20modules). This means both PythonApp (and any other submodule using Python plugin) will use the same environment, so you don’t reinstall packages per module.
* Add Gradle tasks to run the Python app. The plugin provides a PythonTask type to execute Python commands/scripts. We can define a convenient task to launch our PyQt app. For example, in PythonApp/build.gradle:
* import ru.vyarus.gradle.plugin.python.task.PythonTask  
    
  task runDesktopApp(type: PythonTask) {  
   dependsOn pipInstall // ensure deps are installed  
   command = "src/main.py" // path to the main PyQt Python script  
  }
* This creates a Gradle task runDesktopApp that will run python src/main.py using the managed venv. (On Windows it will use the venv’s python.exe automatically.) The plugin automatically makes each PythonTask depend on checkPython (verifies a Python interpreter is available) and pipInstall (install/update the pip packages) before running your command[[6]](https://xvik.github.io/gradle-use-python-plugin/3.0.0/getting-started/#:~:text=Note). So the first time you run this task, it will set up the venv and install PyQt5, OpenCV, etc., then launch your app. In subsequent runs, it will reuse the environment (only installing again if you change the pip list).
* Save the build.gradle and sync Gradle again. Gradle will download the use-python plugin and then attempt to create the virtualenv. **Ensure that** a system **Python 3.x is installed and on your PATH** (e.g., running python --version in a CMD works)[[7]](https://xvik.github.io/gradle-use-python-plugin/3.0.0/getting-started/#:~:text=Make%20sure%20python%20and%20pip,are%20installed). The plugin by default uses python (or python3) from PATH. If you have multiple Python versions or a custom path, set python.pythonPath in the gradle config as shown (e.g., to "C:\\Python39\\python.exe" on Windows). On first sync, you might see Gradle executing tasks like CheckPython and creating the env under .gradle/python.

Gradle multi-module config summary: We now have a root settings.gradle including both modules, an AndroidApp module with the Android Gradle plugin, and a PythonApp module with the Python plugin. Gradle can build the Android APK and also manage the Python environment. Each subproject remains logically separate (different languages and outputs) but share the same version control and can be worked on in tandem. *Gradle treats each subproject as a separate project with its own build file and sources*[*[8]*](https://earthly.dev/blog/gradle-monorepo/#:~:text=tells%20Gradle%20which%20subprojects%20are,and%20where%20they%20are%20located)*.*

## 4. Python Virtual Environment Integration via Gradle

**Virtualenv management:** The **Gradle Use-Python plugin** greatly simplifies Python integration. By default it will create a **virtual environment in the project’s .gradle/python folder** (so it doesn’t clutter your source tree) and install all specified pip packages into it[[9]](https://xvik.github.io/gradle-use-python-plugin/3.0.0/getting-started/#:~:text=,avoid%20permission%20problems%20on%20linux). This means every developer gets an isolated, consistent Python environment for this project, without manually creating a venv. The environment is created on demand during Gradle sync or when running any PythonTask. You don’t have to commit the venv to Git – it’s generated per user (and ignored via .gitignore).

Some implementation details: the plugin will use venv or virtualenv if available to make the env. If for some reason virtualenv isn’t installed globally, the plugin will attempt a user-level install or fallback to --user scope for pip installs[[9]](https://xvik.github.io/gradle-use-python-plugin/3.0.0/getting-started/#:~:text=,avoid%20permission%20problems%20on%20linux). In practice, as long as you have a working Python and pip, Gradle will handle the rest. All specified packages (like PyQt5, OpenCV) will be installed at specified versions.

**Using the Python environment:** Once setup, you can run **Gradle tasks** to execute Python code. For example, the runDesktopApp task we created will launch the PyQt5 GUI. You can run this task in two ways: (1) via command line: .\gradlew :PythonApp:runDesktopApp (on Windows, or ./gradlew :PythonApp:runDesktopApp on Mac/Linux), or (2) from **Android Studio’s Gradle tool window**, expand PythonApp -> Tasks -> runDesktopApp and double-click it. The task will invoke the Python interpreter from the venv and run main.py.

Gradle also allows creating other Python tasks as needed – e.g., a task to run calibration routines or tests. For example, you could add:

task runCalibration(type: PythonTask) {  
 dependsOn pipInstall  
 command = "src/calibrate.py --test-data data/sample.jpg"  
}

This would run a script with arguments. You can also call a module with module = 'your.module.name' if needed[[10]](https://xvik.github.io/gradle-use-python-plugin/3.0.0/guide/usage/#:~:text=task%20mod%28type%3A%20PythonTask%29%20,). The key benefit is that all developers can run these Gradle tasks to execute Python code in a consistent environment, without manually activating a virtualenv each time. Gradle will also **ensure the correct versions** of packages are present before running (via the pipInstall dependency on each task). This addresses reproducibility across the team.

## 5. Configuring Android Studio for Kotlin + Python Development

**Android Studio IDE Setup:** Since Android Studio is built on IntelliJ IDEA, it can support multiple languages through plugins. Out-of-the-box, Android Studio is tailored for Android/Java/Kotlin development. To get Python support (like syntax highlighting, code insight) in Android Studio, install the **“Python Community Edition”** plugin by JetBrains. In Android Studio: go to **File > Settings > Plugins > Marketplace**, search for “Python”, and install **Python CE**. Restart Android Studio if prompted. This plugin provides basic Python editing features (similar to PyCharm Community).

After installing the plugin, Android Studio should recognize .py files and provide coloring, indentation, and some code completion. However, note that **Android Studio might not fully integrate the Python interpreter configuration**. In some cases, you may still see a warning like “**No Python interpreter configured for the module**” at the top of Python files. Unfortunately, Android Studio’s UI doesn’t always expose the interpreter settings (older versions lacked the SDK configuration for Python). **This is expected and can be ignored** – it does *not* prevent running the Python code via Gradle. The Chaquopy documentation (Chaquopy is an Android Python runtime) notes that even with the Python plugin, *Android Studio will show “No Python interpreter” and unresolved reference errors, but these are harmless*[[11]](https://chaquo.com/chaquopy/doc/current/android.html#:~:text=To%20add%20Python%20editing%20suppport,be%20displayed%20in%20the%20Logcat). In our case, since we run Python through Gradle, we don’t need Android Studio to run it directly. You can safely ignore the red squiggly lines complaining about imports in Python files – as long as your code runs via our Gradle tasks, it’s fine. (If you want to eliminate these warnings, one workaround is to **open the Python module as a separate project** in IntelliJ IDEA or PyCharm just to configure an interpreter for code analysis. But this isn’t necessary for functionality.)

That said, ensure that the Python plugin is **enabled for the project**: Android Studio should automatically activate it when it sees Python files. If needed, you can try to configure a Python SDK by going to **File > Project Structure > Platform Settings > SDKs** and adding a Python SDK pointing to your system Python or the project’s venv. In recent Android Studio versions (e.g. Arctic Fox / Chipmunk and later), this might be possible. If it is, select the **PythonApp** module in Project Structure and assign the Python SDK to it. This will give the IDE knowledge of library paths. If the UI does not allow it (as often in AS, the SDKs section is hidden), don’t worry. Your development workflow will be: write Python code in the editor (with basic syntax assistance) and run it via Gradle.

**Run/Debug configurations:** For convenience, you can create Run configurations in Android Studio for each app:

* **Android App:** Use the default “app” configuration (generated by the template) to launch the Android application on a device or emulator. This uses Gradle to build the APK and install it. You can run and debug the Android app normally.
* **Python App:** Create a new **Gradle** run configuration. Go to **Run > Edit Configurations**, click “+”, and choose “Gradle”. Set it to run the task :PythonApp:runDesktopApp. Now you can start the Python desktop app from the Run menu like any other app. This will execute our runDesktopApp Gradle task, launching the PyQt5 GUI. While Android Studio won’t attach a Python debugger this way, you can see the console output of the Python app in the Run console. If you need to debug Python code, you might use print/logging or run the script in a separate Python debugger outside AS. But for most development, this integration should suffice.

With both configurations, developers can **press one play button to launch the Android app on a device and another to launch the desktop app** on their PC. This makes it easy to test interactions (e.g., the phone app connects to the desktop app via sockets – you can run both simultaneously).

**Editor tips:**  
- In the Android Studio Project view, you might want to switch the view mode to “Project Files” instead of “Android”, so you can easily see both modules (Android view tends to hide non-Android modules). In Project Files view, you’ll see the AndroidApp and PythonApp directories.  
- You can mark the Python source directory as “Sources” root by right-clicking PythonApp/src > Mark as Sources Root (if available), which might help the IDE treat it as source for code navigation.  
- The Python plugin allows basic features like “Go to Definition” within Python code, and code completion for standard library. It may not fully index third-party packages without an interpreter configured, but once you add an SDK or ignore the warnings, you can still write code effectively.  
- Use Android Studio’s version control integration for Git to manage commits, review diffs, etc., for all files (Kotlin and Python). It will treat this as one project.

## 6. Git Repository Setup and .gitignore

With the project structure in place, initialize a Git repository in the project root (if not already done by Android Studio’s new project wizard). Since we are using GitHub, create a new GitHub repo and add it as remote. Now let’s configure the .gitignore to avoid committing build artifacts and local config files for both Android and Python. We need to ignore:

* **Android/Gradle files:** compiled binaries, build folders, local config. This includes the build/ directories (for each module and at root), the .gradle/ cache directory, and Android Studio files. A typical Android .gitignore will exclude \*.apk, \*.dex, \*.class files, and the local.properties (which contains the SDK path)[[12]](https://stackoverflow.com/questions/16736856/what-should-be-in-my-gitignore-for-an-android-studio-project#:~:text=,properties)[[13]](https://stackoverflow.com/questions/16736856/what-should-be-in-my-gitignore-for-an-android-studio-project#:~:text=,externalNativeBuild). We also ignore the .idea/ folder and \*.iml module files (Android Studio can regenerate these from Gradle). For example:

# Android/Gradle build outputs and config  
\*.apk  
\*.aab  
\*.dex  
\*.class  
/build/  
.gradle/  
local.properties  
\*.iml  
.idea/ # IDE project settings (avoid sharing workspace specifics)  
.externalNativeBuild/  
captures/

(We list both /build and module build directories. The pattern /build covers the root and AndroidApp/build, but to be safe you can also ignore AndroidApp/build/ explicitly. The snippet above shows general patterns.) The above is based on Android Studio’s default gitignore template[[13]](https://stackoverflow.com/questions/16736856/what-should-be-in-my-gitignore-for-an-android-studio-project#:~:text=,externalNativeBuild). It’s important to exclude local.properties (each dev’s Android SDK path) and .gradle/ (Gradle’s cache) as they are machine-specific.

* **Python files and venv:** We should ignore Python bytecode and virtual env directories. Common patterns include the \_\_pycache\_\_/ folders and \*.pyc files that Python generates[[14]](https://www.pythoncentral.io/python-gitignore-clean-repository-management/#:~:text=%23%20Byte,py.class). Also, if any virtual environment or environment files are present, ignore those (in our case, the venv is under .gradle/python, which is already ignored by ignoring .gradle/). If you decided to create a venv manually inside PythonApp (not needed with our Gradle approach), make sure to ignore that (e.g., PythonApp/venv/). Also ignore Python distribution build folders if any (dist/ or build/ under PythonApp if you run packaging). A Python .gitignore example:

# Python byte-compiled files  
\_\_pycache\_\_/  
\*.py[cod] # .pyc, .pyo, .pyd  
\*$py.class  
  
# Virtual environments  
venv/  
.venv/  
ENV/  
.env/  
env/  
  
# Distribution / packaging  
PythonApp/dist/  
PythonApp/build/  
\*.egg-info/

These entries (derived from common Python gitignore templates[[15]](https://www.pythoncentral.io/python-gitignore-clean-repository-management/#:~:text=%23%20Byte,py.class)[[16]](https://www.pythoncentral.io/python-gitignore-clean-repository-management/#:~:text=,venv)) will keep out compiled artifacts and environment folders. Since our Gradle plugin uses .gradle/python, ignoring .gradle/ already covers the env, but listing venv/ is good practice in case someone creates one.

* **OS-specific junk:** It’s good to ignore OS-generated files like macOS .DS\_Store or Windows Thumbs.db[[17]](https://stackoverflow.com/questions/16736856/what-should-be-in-my-gitignore-for-an-android-studio-project#:~:text=). Also ignore any log files or temporary files. For Windows developers, also ignore any \_temp or such if applicable.

Combining the above, your .gitignore (at root) will have entries covering Android, Python, and IDE/OS artifacts. This ensures that when developers use the repo, they don’t accidentally commit large binaries or local settings. Each dev will generate those locally as needed (Gradle build outputs, the venv, etc.).

**Git attributes:** Since the team is on Windows, consider normalizing line endings. You can add a .gitattributes with \* text=auto to handle CRLF/LF issues, or ensure core.autocrlf is enabled. This prevents newline mismatches between Windows and any potential Unix environments (like CI).

**Pushing to GitHub:** Once .gitignore is set, commit the initial project structure: include all \*.gradle files, the gradlew scripts, the AndroidApp/src and PythonApp/src code (even if just templates or a sample script for now), etc. The Gradle wrapper JAR and properties should be committed as well so others can build easily. Don’t commit the local.properties or any secrets. Then create a GitHub repo (e.g., “MyMonorepoProject”) and push the main branch. On GitHub, you’ll now have a single repository containing both the Android and Python apps.

## 7. Initial Test and Validation Steps

After setting everything up, it’s crucial to verify the setup works on a fresh clone. Here’s a checklist of tests and checkpoints:

* **Open and Sync the Project:** Clone the repository onto another Windows machine (or after deleting local build caches) and open project-root in Android Studio. The IDE should detect the Gradle project. Gradle sync will run – ensure it completes without errors. This will do things like download Gradle wrappers, the Android Gradle plugin, and our Python plugin, then set up the Python venv. You should see in the Gradle Console logs that it created a virtual environment and installed the pip requirements (it might log installing PyQt5, etc.). If something like “Python not found” error appears, make sure Python is installed and python.pythonPath is correctly set.
* **Verify Android module builds:** In Android Studio, click “Build > Make Project”. The Android app (AndroidApp) should compile. Gradle will compile the Kotlin source, etc. Since initially it might be just the template code, it should build an APK successfully. If you have any native libraries (like the USB thermal SDK might include native .so files), ensure they are properly referenced (that may involve placing .so in src/main/jniLibs or adding the SDK AAR as a dependency – those specifics would be handled in Android code, not affecting the structure setup).
* **Run the Android app:** With a device or emulator connected, run the Android run configuration. The app should install and display the default activity (e.g., “Hello World” if using template). This ensures the Android side of the monorepo is correctly configured in the IDE.
* **Verify Python venv and packages:** Open a Terminal (the Android Studio embedded terminal or Windows PowerShell) in the project directory. You can manually activate the virtualenv to inspect it. The venv is likely at .gradle/python/<some\_env\_name> (the plugin may name it something like Python-3.10 depending on version). You can do: .\.gradle\python\bin\activate (or the Scripts\activate for Windows) to enter it, then run pip list to see installed packages. You should see the packages (PyQt5, opencv-python, etc.) with the versions you specified. This confirms the Gradle plugin did install them. (Alternatively, simply run the Gradle task next and catch errors if any package is missing).
* **Run the Python desktop app:** In Android Studio, run the Gradle configuration for runDesktopApp (or execute gradlew :PythonApp:runDesktopApp in Terminal). The first time, this will also trigger pipInstall if not done yet. You should see the Python script execute. If your main.py creates a PyQt window, a GUI window should appear on your Windows desktop. If it’s a console test (say it prints “Hello”), you’ll see that output in the Run console. This confirms that Android Studio/Gradle can successfully launch the Python code. If there’s an error like “Module not found”, it might indicate the package didn’t install or perhaps a naming mismatch (double-check your pip 'package:version' strings and that you ran pipInstall).
* **Socket communication test (basic):** Since eventually the Android and Python apps will communicate, you can do a quick sanity test. For example, implement a small test in both: have the Android app open a socket server on localhost (if emulator, use 10.0.2.2 to reach host) or vice versa. Or simpler, print a known message from one and have the other log that it received it (perhaps using adb logcat for Android). This might be beyond initial setup, but even without fully implementing, at least verify that the Android app can reach the network and the Python app can open a socket. (This is more of a Part 2 concern when building features, but a ping test early can validate there are no firewall or config issues on your dev machine).
* **Check IDE integration:** Within Android Studio, confirm you can navigate the project: open a Kotlin file and a Python file. The Kotlin file should have full IntelliJ support (code completion, refactoring, etc.). The Python file should have syntax highlighting. If you start typing in a Python file, does it suggest standard library functions? If the Python plugin is working, you should get basic suggestions. If not, ensure the Python plugin is enabled (File > Settings > Plugins). The “No interpreter” warning can be ignored as discussed.
* **Git operations:** Create a test commit modifying one Kotlin file and one Python file to ensure that version control picks up changes in both. Android Studio’s VCS window should list changes from both modules. Commit and push to GitHub – verify on GitHub that the repo reflects changes correctly. This ensures .gitignore is not accidentally ignoring something important (e.g., your source files). Only derived files should be ignored. Check that none of the following appear in Git: the build/ directories, local.properties, .gradle/, any .pyc or \_\_pycache\_\_ files, or the venv directory. If they do, adjust .gitignore and remove them from Git.

By completing these steps, you validate that a fresh developer can clone the repo, open it in Android Studio on Windows, and with minimal setup (just having Android SDK and Python installed) be able to build the Android app and run the Python app. Everything should work “out of the box” after the one-time Gradle sync.

## 8. Bootstrapping and Automation Scripts

To further streamline onboarding and repetitive tasks, consider adding **bootstrapping scripts**:

* **Environment Setup Script:** Although Gradle will create the Python env on the fly, you can provide a simple PowerShell or batch script to initialize everything for a new developer. For example, setup.ps1 could run gradlew pipInstall (to set up the Python packages) and perhaps gradlew assembleDebug (to pre-build the Android app). This one-liner script can ensure that after cloning, the dev just runs it to get all dependencies. On Windows, a PowerShell script might be preferable. It could also check if Python is installed and give a user-friendly message if not. For instance:

# setup.ps1 (pseudo-code)  
if (!(Get-Command python -ErrorAction SilentlyContinue)) {  
 Write-Error "Python is not installed or not on PATH. Please install Python 3.x before continuing."  
 exit 1  
}  
./gradlew --no-daemon pipInstall assembleDebug

This would install Python deps and build the Android app (download Gradle dependencies, etc.). The --no-daemon is optional but can avoid leaving background gradle processes. After this, the developer can immediately run the apps.

* **Gradle tasks for common workflows:** We already added runDesktopApp. You could add tasks for other dev workflows, e.g., a combined task to **run both apps**. While Android Studio can’t run two configurations simultaneously easily, you can make a Gradle task that spawns the Python app and then launches the Android app on an emulator via ADB. This is advanced (would require Gradle to talk to ADB or use the Android Gradle plugin tasks). It might be easier to just run the two separately. However, a script could be made to automate starting an Android emulator then launching the app and the desktop app. This could be a .bat script using adb install and then launching the Python exe. Such conveniences can be added as the project matures.
* **Documentation:** In the repo’s README.md, document the setup steps for others: e.g. “Install Android Studio, install Python, clone repo, run gradlew pipInstall assembleDebug, then open in Android Studio.” Also mention any quirks (like the interpreter warning). This will help new team members.

By providing these scripts and documentation, you reduce the chance of environment issues. Each Windows developer’s machine should be set up similarly (Android Studio installed with an SDK, and Python installed). With the scripts, they don’t need to manually pip install anything or fiddle with Gradle commands – one command can prep everything.

## 9. Collaboration and CI Considerations

This monorepo approach is designed to support a **team of Windows developers** and integration with GitHub:

* **Consistency for Developers:** Everyone uses the same IDE (Android Studio) and the same process to run both apps. This consistency means fewer “it works on my machine” problems. For example, if one dev adds a new Python library, they add it to build.gradle and commit. When others pull, Gradle will auto-install it for them – no need for each dev to manually figure out the new dependency. Similarly, changes to Gradle build config (like adding a new Android library or changing SDK version) propagate to all. The use of Gradle and the plugin enforces consistent versions for both Java/Kotlin and Python dependencies across all devs.
* **GitHub CI Integration:** With one repo, you can set up **GitHub Actions** (or another CI service) to build and test both components on each push/PR. For example, you might configure one job to run on a Windows VM (since the Python GUI might require a display or at least Windows for PyQt, and building Android on Windows is possible with the right SDK installed). The CI steps could include:
* Set up Java and Android SDK, then run gradlew assembleDebug to ensure the Android app compiles with no errors.
* Install Python 3 and run gradlew -p PythonApp pipInstall (or a custom Gradle test task) to ensure Python dependencies install. You could even run a headless test of some Python functionality (though PyQt5 GUI can’t be easily tested in CI without a display, you can separate logic to testable modules).
* Alternatively, split jobs: one Android build job (using the official Android CI images) and one Python lint/test job (using a Windows or Ubuntu runner with Python). Both pulling from the same repo. This ensures that neither side is broken by a commit.

The monorepo makes it straightforward to ensure compatibility: if you change a message format in the Android code, you can update the Python parsing code in the same commit and have CI verify both build. There’s no need to coordinate PRs across two repos.

* **Collaboration with Git and GitHub:** All code is reviewed in one place. Code reviews can span both languages – reviewers can see, for instance, that a change in the Android app’s data logging is paired with a corresponding change in the Python app’s data reading. This helps with **protocol consistency**. Using pull requests, you can require that both the Android and Python subprojects pass their checks. For multiple developers, feature branches will contain changes to either or both subprojects as needed. When merging, the whole feature goes in together, reducing the risk of version skew between app and controller.
* **Team Workflow on Windows:** Since everyone is on Windows, ensure that project instructions cover Windows specifics. For example, how to handle line endings as mentioned, using PowerShell vs CMD vs Git Bash if needed. In Android Studio on Windows, developers should install the latest **Google USB driver** if testing on physical devices, and ensure their Android SDK path is configured (Android Studio handles this). The local.properties not being in Git means each dev may need to have one – but Android Studio will auto-generate it when they open the project and select an SDK. This is normal.
* **Dependency Management:** The Android side uses Gradle/Maven for dependencies, and the Python side uses pip (via our Gradle plugin). Both allow repeatable builds. You might want to lock versions (we pinned them in build.gradle). Over time, update dependencies in a controlled manner. If the Python app grows complex, you could consider using a requirements.txt for easier reading of dependencies; the Gradle plugin even supports pointing at a requirements.txt (it can use pipInstall on a requirements file, or you could call a PythonTask to pip install -r). However, maintaining them in Gradle is fine for now.
* **Gradle and Build Performance:** In a multi-module project, Gradle will handle tasks for each module. It’s generally efficient – if you run gradlew build, it will build the Android APK and also run any Python tasks that are part of the default lifecycle (by default, the Python plugin might not tie into build task unless you add it). Typically, assembleDebug will ignore Python tasks unless explicitly depended on. You might integrate further, but it’s often best to keep them separate (no need to run Python on every Android build). Developers can build Android frequently without reinstalling Python packages each time (since those only install when changed).
* **Future Enhancements:** To improve onboarding, you could check in some IDE configs like code style schemes or inspections profiles so that Kotlin and Python code have consistent formatting (Android Studio can format Kotlin by default; for Python you might use an external tool like Black – could integrate that via a Gradle PythonTask). Encourage writing documentation in the repo (maybe a Wiki or docs folder) for any setup that isn’t automated.

By following this guide, you set up a robust monorepo structure where an Android mobile app and a Python desktop app co-exist and can be developed in parallel from one environment. Each developer can pull the repo and be productive quickly, running both halves of the project on their Windows machine. The combination of **Gradle multi-project** and **Python venv integration** provides a cohesive build system, and Android Studio (with the Python plugin) serves as the one-stop IDE. This setup lays the groundwork for Part 2 of the project, where you will focus on the actual implementation of features (camera integration, sensor data streaming, calibration algorithms, etc.) on this solid foundation.

**Sources:** The above recommendations draw on best practices for multi-module Gradle projects[[2]](https://earthly.dev/blog/gradle-monorepo/#:~:text=contains%20all%20of%20the%20subprojects,and%20where%20they%20are%20located)[[3]](https://earthly.dev/blog/gradle-monorepo/#:~:text=mkdir%20subproject), using Gradle plugins to manage Python environments[[5]](https://xvik.github.io/gradle-use-python-plugin/3.0.0/guide/multimodule/#:~:text=When%20used%20in%20multi,same%20environment%20for%20all%20modules)[[6]](https://xvik.github.io/gradle-use-python-plugin/3.0.0/getting-started/#:~:text=Note), and standard gitignore patterns for Android and Python projects[[18]](https://stackoverflow.com/questions/16736856/what-should-be-in-my-gitignore-for-an-android-studio-project#:~:text=,navigation)[[14]](https://www.pythoncentral.io/python-gitignore-clean-repository-management/#:~:text=%23%20Byte,py.class). The approach is inspired by prior art of mixing Java/Kotlin with Python in a single repo (e.g., Chaquopy documentation for editing Python in Android Studio[[11]](https://chaquo.com/chaquopy/doc/current/android.html#:~:text=To%20add%20Python%20editing%20suppport,be%20displayed%20in%20the%20Logcat)). By adhering to these guidelines, your team should enjoy a smooth development experience despite the polyglot nature of the project. Good luck with your implementation!

[[1]](https://earthly.dev/blog/gradle-monorepo/#:~:text=Gradle%20is%20a%20powerful%20and,many%20other%20languages%20as%20well) [[2]](https://earthly.dev/blog/gradle-monorepo/#:~:text=contains%20all%20of%20the%20subprojects,and%20where%20they%20are%20located) [[3]](https://earthly.dev/blog/gradle-monorepo/#:~:text=mkdir%20subproject) [[8]](https://earthly.dev/blog/gradle-monorepo/#:~:text=tells%20Gradle%20which%20subprojects%20are,and%20where%20they%20are%20located) Building a Monorepo with Gradle - Earthly Blog

<https://earthly.dev/blog/gradle-monorepo/>

[[4]](https://xvik.github.io/gradle-use-python-plugin/3.0.0/getting-started/#:~:text=Important) [[6]](https://xvik.github.io/gradle-use-python-plugin/3.0.0/getting-started/#:~:text=Note) [[7]](https://xvik.github.io/gradle-use-python-plugin/3.0.0/getting-started/#:~:text=Make%20sure%20python%20and%20pip,are%20installed) [[9]](https://xvik.github.io/gradle-use-python-plugin/3.0.0/getting-started/#:~:text=,avoid%20permission%20problems%20on%20linux) Getting started - Gradle use-python plugin

<https://xvik.github.io/gradle-use-python-plugin/3.0.0/getting-started/>

[[5]](https://xvik.github.io/gradle-use-python-plugin/3.0.0/guide/multimodule/#:~:text=When%20used%20in%20multi,same%20environment%20for%20all%20modules) Multi-module - Gradle use-python plugin

<https://xvik.github.io/gradle-use-python-plugin/3.0.0/guide/multimodule/>

[[10]](https://xvik.github.io/gradle-use-python-plugin/3.0.0/guide/usage/#:~:text=task%20mod%28type%3A%20PythonTask%29%20,) Usage - Gradle use-python plugin

<https://xvik.github.io/gradle-use-python-plugin/3.0.0/guide/usage/>

[[11]](https://chaquo.com/chaquopy/doc/current/android.html#:~:text=To%20add%20Python%20editing%20suppport,be%20displayed%20in%20the%20Logcat) Gradle plugin - Chaquopy 16.1

<https://chaquo.com/chaquopy/doc/current/android.html>

[[12]](https://stackoverflow.com/questions/16736856/what-should-be-in-my-gitignore-for-an-android-studio-project#:~:text=,properties) [[13]](https://stackoverflow.com/questions/16736856/what-should-be-in-my-gitignore-for-an-android-studio-project#:~:text=,externalNativeBuild) [[17]](https://stackoverflow.com/questions/16736856/what-should-be-in-my-gitignore-for-an-android-studio-project#:~:text=) [[18]](https://stackoverflow.com/questions/16736856/what-should-be-in-my-gitignore-for-an-android-studio-project#:~:text=,navigation) git - What should be in my .gitignore for an Android Studio project? - Stack Overflow

<https://stackoverflow.com/questions/16736856/what-should-be-in-my-gitignore-for-an-android-studio-project>

[[14]](https://www.pythoncentral.io/python-gitignore-clean-repository-management/#:~:text=%23%20Byte,py.class) [[15]](https://www.pythoncentral.io/python-gitignore-clean-repository-management/#:~:text=%23%20Byte,py.class) [[16]](https://www.pythoncentral.io/python-gitignore-clean-repository-management/#:~:text=,venv) Python .gitignore: Clean Repository Management | Python Central

<https://www.pythoncentral.io/python-gitignore-clean-repository-management/>