ChatGPT

# Combining fyp-gsr-windows and fyp-gsr-android into a Monorepo

**Objective:** Set up a single repository containing both the Windows desktop application (Python/Qt/C++ hybrid) and the Android mobile application (Kotlin/Java). This will enable simultaneous development, centralized version control, and easier cross-project integration, all within JetBrains IDEs.

## Monorepo Folder Structure

Organize the new repository with clear top-level directories for each project. For example:

```
gsr-project-monorepo/
├── android/       # Android app (mobile)
│   ├── app/ ...   # (All files from fyp-gsr-android go here)
│   ├── build.gradle
│   ├── settings.gradle
│   └── ...
├── windows/       # Windows app (desktop)
│   ├── src/ ...   # (All files from fyp-gsr-windows go here)
│   ├── requirements.txt
│   ├── main.py
│   └── ...
├── README.md      # Monorepo README referencing both subprojects
├── .gitignore     # Combined ignore rules for both Android & Windows
artifacts
└── other files... (optional common docs, etc.)
```

This mirrors the typical monorepo approach of having each project in its own subfolder [1]. Use descriptive names (e.g. `android` and `windows` or `mobile` and `desktop`) to distinguish components. Each subdirectory retains its internal structure: the Android folder will contain its Gradle build files, `app/src/`, etc., and the Windows folder will contain the Python source, configs, and any C++ binaries or third-party libs. Keeping them separate at the top level prevents any mixing of files or build outputs.

**Top-Level README:** Provide an overview and then direct developers to the project-specific documentation in each subfolder. For example, "See `android/README.md` for the Android app and `windows/README.md` for the Windows app." This helps new contributors quickly find the relevant info.

**.gitignore:** Create a root `.gitignore` that includes patterns from both projects. For instance, ignore Python bytecode and virtual env folders (`__pycache__/, *.pyc, venv/`), Qt build artifacts (if any), as well as Android/Gradle outputs (`*/build/, *.apk, *.aar`) and IDE config like `.idea/` and local settings. This ensures that each project's build artifacts remain untracked.

By structuring the monorepo this way, you cleanly separate the environments while still grouping them under one repository. Developers can work on either component without the files interfering, and it's clear which part of the codebase they're in at any time.

## Combining Git Histories and Version Control

If you want to **preserve commit history** from the two original repos, use Git's subtree or filter-branch methods instead of simple copy-paste. Git **subtree merging** is a straightforward approach: it lets you pull one repository into a subdirectory of another **including its entire history** [2]. For example, you could start with a fresh empty repo (or use one of the existing repos as a base), then run:

```
# From within the new monorepo git repo:
git subtree add --prefix=windows https://github.com/buccancs/fyp-gsr-
windows.git main
git subtree add --prefix=android https://github.com/buccancs/fyp-gsr-
android.git main
```

Each command will fetch the specified repo and place its files under the given prefix directory, preserving that repo's commit history as a separate lineage [3] [4]. The result is that running `git log` in `windows/` will show the historical commits from *fyp-gsr-windows*, and similarly for `android/`. Subtree merges create a merge commit that ties the histories together, and embed metadata about the source commit for traceability. (You'll see a commit message like "Add 'windows/' from commit …" after running the above [4].)

Alternatively, you can use more manual methods (e.g. using `git filter-branch` or the newer `git filter-repo` tool) to rewrite each repository's history into a subfolder, then merge with `--allow-unrelated-histories` [5]. This gives fine-grained control (like renaming branches, handling tags, etc.), but is more complex. If preserving history isn't critical, a simpler route is to add one repo's files into the other and make a single initial commit in the monorepo (you would still likely want to keep the old repos around for reference). However, using subtree or filter-branch is recommended if you want a true monorepo with full history of both projects.

**Merging Git Settings:** After combining, review Git settings: - **Branches:** Decide on a unified branching strategy. For example, if both had a `main` branch, your monorepo will now have one `main`. You might tag the last commit of each original repo for reference (e.g. `windows-final-oldrepo` tag) before merging. - **Tags:** If both projects used overlapping tag names (e.g. `v1.0` in both), you may need to rename or prefix them before merging to avoid conflicts in the monorepo. - **Remotes:** Once merged, update your remote URLs – you'll presumably host this monorepo in a new origin (e.g., a new GitHub repo or one of the existing ones repurposed). Developers should clone/pull the new repo going forward. You can deprecate the old ones or mark them read-only.

**Git Submodules vs Monorepo:** *Avoid* using Git submodules here – a submodule would keep the projects separate and just link them, which is not the goal of a true monorepo. Instead, we're fully integrating them. In a monorepo, any commit can touch code in both subprojects, and you manage one set of issues/pull requests. This ensures, for example, that if you change a message format in the Android app, you can update the Windows app in the same commit.

**.gitignore and Git Attributes:** As mentioned, combine ignore rules. Also consider any `.gitattributes` (for e.g. end-of-line normalization or LFS settings). If one project was using Git LFS for large files (like binary assets or model files), ensure those track patterns are carried over to the monorepo's `.gitattributes`.

By carefully merging the histories and configuration, you'll have a single repository where both components' histories are intact and versioned together. This sets the stage for unified issue tracking and easier syncing of changes across the two apps.

## JetBrains IDE Setup for a Multi-Project Monorepo

With both projects in one repository, you have two main approaches for your JetBrains IDEs:

### 1. One Workspace in IntelliJ IDEA Ultimate (All-in-One)

The simplest way to develop both an Android app and a Python/C++ app together is to use **IntelliJ IDEA Ultimate** as your one-stop IDE. IntelliJ IDEA supports multiple languages via plugins [6] – in fact, IntelliJ with the Python, Android, and C++ plugins can handle all parts of this monorepo in one window. This avoids any conflicts with multiple `.idea` directories and gives you a unified view of the whole project.

**Steps to Configure IntelliJ:**

- **Open the Monorepo Root:** In IntelliJ IDEA, choose *File → Open…* and select the root `gsr-project-monorepo` directory. IntelliJ will open it as a project. Initially, it may not know how to treat the subfolders, but we will set them up as modules.

- **Enable Required Plugins:** If you haven't already, enable the Python plugin (for the Windows app) and Android support. In IntelliJ Ultimate, Android support is built-in; for Python, go to *Settings → Plugins* and install **Python**. (If it's the first time opening a Python file, IntelliJ might prompt to install the Python plugin automatically [7].) Also ensure the C/C++ plugin is installed if you plan to edit or navigate any C++ code (though for just using a pre-built binary, this isn't required).

- **Import as Modules:** Use IntelliJ's module structure to handle each sub-project as an independent module:

- Go to *File → New → Module from Existing Sources…* and select the `windows/` directory. Since this is a Python project, there is no `.iml` yet, so choose "Create module from existing sources" and select the folder. Assign it a module name like "windows" or "GSR-Windows" and finish the wizard. IntelliJ will treat this as a generic module – you can then mark the `windows/src` (or equivalent) as Sources Root if needed, and set up a Python SDK for it.
- Next, import the Android project: *File → New → Module from Existing Sources…* again, and select the `android/build.gradle` or the `android/` folder. IntelliJ should detect it's a Gradle project (with the Android plugin) and offer to import it as a Gradle module. Follow the prompts (it may ask to "Open as Project" or "Import Gradle project"; choose to import into the existing project). This will set up the Android module with its Gradle build. After syncing, you should see an "android" module (and its submodules for the app) in the Project Structure.

- After this, in **Project Structure** (`Ctrl+Alt+Shift+S` in Windows/Linux or `Cmd+;` on Mac), you should see two modules listed (plus perhaps submodules for the Gradle project). Each will have its own dependencies and language level. This multi-module setup lets IntelliJ treat them somewhat separately (each with correct classpaths, etc.) [8]. Essentially, *"import each* *subdirectory as a module. This will treat them as individual apps with their own dependency trees”* [8].

- **Module Grouping (Optional):** You can group modules to keep things organized. For example, put the Android-related modules under a group "Android" and the desktop under "Windows". This is mostly visual – it helps in the Project view if there are many modules [9] [10]. In newer IntelliJ versions, modules can also be grouped by naming conventions (e.g., prefix them with `android.` and `windows.` in the module names to auto-group) [11].

- **Configure SDKs and Interpreters:**

- For the **Android module**: IntelliJ will use the Gradle settings. Ensure you have an appropriate JDK (Java SDK) for Gradle/Android. IntelliJ will likely prompt to set one if not configured. Use Java 11 or whatever the Android project requires (Android Studio Arctic Fox corresponds to roughly Gradle plugin that uses Java 11). Also make sure the Android SDK path is set (IntelliJ should pick up the Android Studio SDK if installed, or you can configure one under *File* → *Settings* → *Appearance & Behavior* → *System Settings* → *Android SDK*).

- For the **Python module**: Set up a Python interpreter. Go to *File* → *Settings* → *Languages & Frameworks* → *Python SDK* (or use the Python interpreter widget) and **add a Python SDK**. It’s wise to create a **virtual environment** for this project to isolate Python dependencies. IntelliJ can create a new venv: choose a base Python 3.8+ installation and point it to create a virtual environment (for example, in `windows/venv`). After it’s added, install the needed packages. You can use the built-in **Python Packages** tool or simply let IntelliJ/PyCharm detect the `requirements.txt`. For instance, IntelliJ might offer “Install requirements” when it finds the `requirements.txt` file. This will run `pip install -r requirements.txt` [12] in the chosen venv, bringing in PyQt5, OpenCV, etc. as listed for the Windows app [13].

- **Project Configuration Isolation:** By using modules, each part of the project maintains its own settings:

- The Android module will have its Gradle build and won’t be affected by Python packaging.
- The Python module can have its own source roots and exclude directories (for example, you might mark the `windows/third_party` or `windows/__pycache__` as *Excluded* so the IDE doesn’t index binaries or caches).
- IntelliJ’s **Run/Debug Configurations** can be set per module or globally – we’ll cover that below.

Using IntelliJ IDEA Ultimate in this way means you only have one `.idea` directory at the root (with module configurations inside). You get a single unified Version Control view (one Git repo). You can search across the whole codebase, or limit your search to one module if desired. Navigation is smooth: the IDE will know which files belong to the Android project (and offer Android-specific features on those) and which belong to Python (with code completion, type hinting, linting for Python). This approach is recommended for the tightest integration.

> **Tip:** If you open a Python file and the IDE complains about missing interpreter or libraries, double-check that the Python SDK is set for that module and that you installed the requirements. Likewise, if Gradle sync fails, ensure the Android SDK and Gradle wrapper are configured properly (you may need to install the Android SDK Platform specified by the project, e.g. API level 30+, as noted in prerequisites [14]).

## 2. Alternative: Separate IDEs (Android Studio & PyCharm)

If you prefer to use domain-specific IDEs (e.g. Android Studio for the mobile app and PyCharm for Python) rather than one IntelliJ instance, you can still work with the monorepo – but you'll effectively treat each subdirectory as its own project in its respective IDE.

This setup requires a bit more care with IDE configuration, because you'll have **two IDEs pointing to the same Git repo**. The key is to avoid both IDEs writing conflicting project settings:

- **Android Studio for the Android app:** Android Studio is essentially IntelliJ tuned for Android. You can open the `android/` folder in Android Studio by using *Open...* and selecting the `android/build.gradle` (or just the `android` folder – AS should detect it as a Gradle project). Android Studio will create its `.idea` folder *inside* `android/` (since you opened that as the project root in that IDE). This keeps Android Studio's settings isolated to just the Android subdirectory. You'll have all the usual Android development tools (UI designer, ADB, etc.) available. Just ensure that when using Git in Android Studio, it's pointing to the monorepo (which it will, but note it will only "see" the files under `android/` by default since that's the project root for AS).

- **PyCharm (or IntelliJ with Python) for the Windows app:** Similarly, open the `windows/` folder in PyCharm as a project. PyCharm will make a `.idea` inside `windows/`. Configure the Python interpreter/venv in PyCharm as you normally would for that project (install requirements, etc.).

Now you have two separate IDE windows: one for Android, one for Windows. Each is focused on its part of the repo, with no overlap in settings. This avoids the "pop-up conflicts" that occur if two different JetBrains IDEs try to use the same `.idea` project config. Since each subfolder has its own `.idea`, Android Studio and PyCharm won't step on each other's toes.

**Version Control Considerations:** Both IDEs will detect the Git repository (the root of the monorepo). However, because each IDE only includes its subdirectory in the project, you might not see changes outside that scope in the IDE's version control window. For example, Android Studio will list changes under `android/` (and might not list changes done in `windows/` unless those files are included in the AS project). Likewise, PyCharm will track changes in `windows/` but not outside. To handle cross-project commits (e.g., you update a protocol in both Android and Windows code at once), you can still use command-line Git or a GUI Git tool to stage and commit all files. Alternatively, consider opening the whole monorepo in IntelliJ just for doing version control or searching, even if you edit in separate IDEs. If this becomes cumbersome, it's a sign you might want to switch to the single IntelliJ approach.

**Using IntelliJ and CLion (for C++):** If you need to actively develop the C++ capture engine on the Windows side, you might involve CLion (JetBrains' C++ IDE) or Visual Studio. One approach is to include the C++ source (if you have it) in the monorepo under, say, `windows/cpp/`, and add a CMakeLists.txt. You could open that folder with CLion. However, note that CLion and IntelliJ/PyCharm would again conflict if sharing `.idea`. The safer approach would be to either use IntelliJ's C++ plugin (less powerful than CLion but avoids a new IDE) or treat the C++ code as a separate project. Given the Windows app uses an external executable for C++ (as per setup instructions, dropping an EXE into `third_party`), you might not need CLion at all if you aren't modifying that code regularly.

In summary, using separate IDEs is possible but requires discipline. Many developers in this situation choose IntelliJ Ultimate with all plugins because it **"supports all languages via plugins"** out of the box [6], simplifying the workflow. If you do stick to Android Studio + PyCharm, keep the `.idea` directories separated as described, and remember that both are editing the same Git repo – you'll need to coordinate commits if you have changes spanning both projects.

## Isolating Build Tools and Dependencies

One big advantage of a monorepo is unified version control, but we still want to **isolate each project's build environment** so they don't conflict:

- **Android (Gradle Build):** The Android app will continue to use Gradle (with the Android Gradle Plugin) to compile and build the APK. All of its dependencies (AndroidX libraries, Topdon SDK `.aar` files, etc.) remain defined in the `android/app/build.gradle` and contained in the Gradle wrapper's directories. These won't interfere with anything on the Windows side. Make sure the Android project's Gradle wrapper files are included in the monorepo (`android/gradlew` and `android/gradle/wrapper/gradle-wrapper.properties`) so everyone uses the same Gradle version. Within the IDE, you'll run Gradle tasks as usual (e.g. assemble, installDebug) scoped to the Android module.

- **Windows (Python/C++ Build):** The Windows app doesn't have a single "build" like a compiled program, but it may involve installing Python packages or even compiling native extensions. Keep Python requirements separate by using a virtual environment dedicated to this project (e.g., if using IntelliJ/PyCharm, it could be in `windows/venv`). This ensures that any Python library versions (like PyQt5, NumPy, etc. listed in `requirements.txt` [13]) don't clash with other Python projects on your system. It also means team members can each create their own environment consistent via the `requirements.txt`. If the Windows app uses PyInstaller or a similar tool to create an exe, that's also confined to that subproject.

- **C++ Dependencies:** If the Windows app's C++ component requires particular DLLs or SDKs (e.g. FLIR camera SDK), handle those as per that project's docs. For example, the readme indicates needing to place the `RGBTPhys_CPP` executable and DLLs in `windows/third_party/RGBTPhys_CPP/` [15]. Keeping those in the `windows` folder means they are only relevant to the Windows module. They can even be .gitignored or stored via Git LFS if they are large binaries.

- **No Shared State by Default:** There is generally no runtime or build-time overlap between an Android app and a Python desktop app, so you don't have to do much to *maintain* isolation – just don't accidentally intertwine their build processes. For instance, avoid any single Gradle or Maven build that tries to run Python code, and conversely don't try to run Android Gradle tasks from Python scripts. Treat them as separate products that just happen to live in one repo.

- **IDE Configuration Isolation:** In IntelliJ, each module can have its own settings for things like linter rules or formatting. For example, you might use Google Java code style for Android code and PEP8 conventions for Python. IntelliJ can handle this via scopes or by recognizing file types. Prettier or ESLint configs (if you had a JS component, for instance) can also be per-folder. Indeed, modern IntelliJ has improved monorepo support to respect per-subproject settings (e.g. separate Prettier configs, etc.) [16].

- **Build Tools Versions:** Keep an eye on tool versions: for instance, if the Android project and the C++ or Python parts use the same tool (unlikely, except Git), ensure compatibility. An example might be if both use a tool like CMake (Android can use CMake for NDK builds, and your C++ might use CMake). If so, align versions or isolate usage. But in this case, the Android app's

Topdon SDK and such are self-contained, and the Python app's tools (OpenCV, PyQt) are entirely separate.

- **Continuous Integration (CI):** If you set up CI pipelines, you might have one job for building/testing the Android app and another for the Windows app. Monorepo makes it possible to coordinate them (and you can set them to trigger separately if changes detected in certain paths). Just ensure the CI environment has both Java/Android SDK and Python environments available, or split the jobs by project.

By maintaining this separation, you prevent, say, a Python library upgrade from breaking the Android build or an Android Gradle plugin update from affecting the desktop app. Each has its own dependency ecosystem: Maven/Gradle for Android vs. pip for Python, etc. The monorepo ties them together in source control but not in runtime.

## IDE Workflow: Navigation, Building, and Debugging in Tandem

One major benefit of using JetBrains tools is the ability to *run and debug both applications side by side*. Here are some tips for smooth development in this hybrid setup:

- **Run Configurations (Single Project Method):** In IntelliJ, you can create run/debug configurations for each app:
- For the **Windows app**: Create a Python run configuration that runs `main.py` (or whichever startup script) with the appropriate working directory. You can set environment variables or parameters if needed (e.g., a specific config file). Verify it uses the correct Python interpreter (your venv).
- For the **Android app**: You'll have the standard Android run configurations (generated by IntelliJ/Android support) to launch the app on an emulator or device. You can customize these (which variant, activity, etc., but presumably it's a single application module).

Once these are set up, you can **launch them simultaneously** if needed. IntelliJ provides a **Compound Run/Debug configuration** feature, which lets you start multiple configurations in parallel [17]. For example, you can create a *Compound* named "Run All GSR Apps" that includes the Python run config and the Android app config. Starting this compound will deploy the Android app (to an emulator or device) and run the Windows Python app at the same time. Each will open in its own Run/Debug tab, allowing you to monitor both. This is great for testing the end-to-end interaction (e.g., the Android phone streaming data and the Windows app receiving it) in one go.

*JetBrains documentation note: "IntelliJ IDEA provides several ways to run/debug multiple things at once... A Compound run configuration lets you launch several run/debug configurations simultaneously."* [18] [19]. This is exactly what you need for a client-server or phone/PC scenario.

- **Parallel Debugging:** You can also debug both at once. For instance, put breakpoints in the Android Kotlin code (perhaps where it sends data) and in the Python code (where it receives or processes data). Launch the compound config in **Debug** mode – IntelliJ will attach the debugger to both processes. The Android debugger will stop on a breakpoint in Android code (while the Python continues running), and vice versa. Each debugger has its own tab and controls. Be mindful of focus: when a breakpoint hits on one side, the other app may continue running unless you've designed sync points. This setup is extremely powerful for diagnosing cross-platform interaction issues.

- **Smooth Navigation:** IntelliJ's global search (`Double Shift`) will search through both projects' files, so you can quickly find where a certain keyword or class is defined, regardless of language. The **Project view** will show both modules; you can of course navigate within each. Features like "Go to Declaration" will work within the scope of each language (e.g., jump to a Python function definition or a Kotlin class as usual). They won't directly link between Python and Kotlin (they're unrelated codebases), but you might use the IDE's **TODO** or **Bookmarks** features to keep track of places that correspond between projects (for example, a TODO comment in Python referencing "update Android accordingly").

- **JetBrains Specific Features:**

- **Refactor/Rename:** Within each module, you can safely refactor as you normally would (e.g., rename a Python method, or an Android class) without affecting the other module. There's no cross-language refactoring needed in this scenario.
- **Search in Path and Scope:** Use **Scopes** if you want to limit searches or analysis to one of the subprojects. IntelliJ might automatically create scopes for each module. This can be handy if, for example, you want to run *Code Inspections* or *Reformat Code* on just the Android code or just the Python code.

- **Dual Run with Different IDEs:** If you went with the separate IDE approach (Android Studio & PyCharm), you can still run both: just launch the Android app from Android Studio and the Python app from PyCharm. They will function together. You lose the nice compound-one-click launch, but it's manageable. You might use one of the IDEs just as a glorified debugger attached to a running process if needed. For example, you could start the Python script from the command line and attach PyCharm's debugger to it, while Android Studio runs the app. However, this is more complex – using one IDE for both is simpler.

- **JetBrains Workspaces (FYI):** JetBrains has introduced a *"Multi-Project Workspaces"* feature/plugin, which allows combining separate projects (even separate repos) into one IDE window. In our case, since we already combined into one repo, you don't need this. But it's good to know you effectively have built a custom workspace by using modules in one project.

- **GUI Tools & Resources:** All the usual JetBrains perks apply:

- The **UI Designer** in Android Studio (IntelliJ) will work for your Android layouts.
- The **Python Debugger** (with visualization tools, variable viewers) works for your Python code.
- If the Windows app uses Qt for UI, you might be designing `.ui` files with Qt Designer externally; those can be checked in and you can open them in Qt Designer from the IDE or separately.

- **Profiling/Monitoring:** Android Studio provides logcat and profiling tools for the Android app. For the Python app, you might integrate profiling via plugins or run performance tests separately. They don't conflict since they target different runtimes.

- **Documentation and Common Resources:** Since both projects are in one repo, you might centralize some documentation. For example, if there's a communication protocol between the phone and PC, you can put that spec in a `docs/` folder at the root or a shared Markdown file. That way, it's versioned alongside code. You can even have code or scripts that are shared, but generally, keep platform-specific code in their module folders. If truly shared code emerges (say a Python script that is also useful for Android, or some config), you could introduce a third folder for “shared” resources. For now, it sounds like they are mostly independent but coordinated systems.

**Testing:** You can run tests for each project: - For Android, you can run unit tests on the JVM or instrumentation tests on an emulator via the IDE’s test runner. - For Python, you can run `pytest` or other tests (the Windows app has a comprehensive test suite [20]) using the Python test runner config in IntelliJ/PyCharm. Each will report in the Test Runner panel. They won’t clash, and you can even run them in parallel if desired (just like running the apps).

**Hot Reload / Instant Run:** These are mostly separate – Android’s instant run (or “Apply Changes”) can speed up iterative development on Android, while Python doesn’t exactly have a hot-reload (unless you code one). But since they are decoupled, you manage each with their normal workflow.

By leveraging the IDE’s multi-run and multi-debug capabilities, you’ll find you can develop features that span mobile and desktop quite fluidly. For example, you might set a breakpoint where the Android app sends a Bluetooth packet and another where the Windows app receives it, and step through to compare timings or data values.

## Additional Tooling & Project Management Tips

- **Consistent Naming and Conventions:** In a monorepo, it’s helpful to prefix commit messages or branch names with the part they affect, especially if some team members focus on one side. For instance, commit messages could start with `[Android] ...` or `[Windows] ...` when a change is isolated to one. This is just a convention, but it can make `git log` easier to scan. You could also use directory-based ownership (CODEOWNERS file) to assign reviewers for changes under `android/` vs `windows/` if working with others.

- **Gradle Configuration:** If you ever want the Android build to be aware of the Windows part (for example, bundling a Python script into the Android app – perhaps not applicable here), you could add a Gradle module or task that points to it. But generally, keep them separate. One minor point: Android Studio might complain if the project root isn’t a typical Android project. If you run into any issues opening the monorepo root in Android Studio, just open the `android/` subfolder as discussed.

- **Upgrading IDEs/Plugins:** With IntelliJ handling everything, ensure you update plugins when needed. JetBrains updates often include fixes for multi-language support. For example, by 2025 IntelliJ had improved handling of multiple subprojects (like respecting each project’s config for tools like Prettier) [16] – staying up to date will give the best experience in a monorepo.

- **Memory and Performance:** Large monorepos can tax IDEs, but with just two projects it should be fine. Still, if you notice IntelliJ getting slow due to indexing two very different codebases, you can fine-tune the indexing: mark large binary directories (like `android/app/build/` or any dataset folders in `windows/`) as excluded. This prevents the IDE from scanning big output files. Also, consider increasing the IDE’s heap if necessary (via `idea.vmoptions`), though likely not needed here.

- **Continuous Development Workflow:** You mentioned using JetBrains IDEs, which suggests an interactive dev style. For the Android app, you’ll frequently use the emulator or physical test devices – IntelliJ/Android Studio will manage those (AVD manager, etc.). For the Windows app, if it interfaces with hardware (GSR sensor, FLIR camera), you’ll test on a Windows machine with those attached. If you develop on Windows, IntelliJ handles both. If you develop on another OS (say Linux or Mac) for the Python part, be mindful that the Windows-specific integrations (like COM ports or certain SDKs) may not function fully on non-Windows OS. In that case, you might develop cross-platform parts on any OS, but test the full pipeline on a Windows machine as needed.

- **Communication between the apps:** The Windows README indicates support for dual phone control via WiFi/Bluetooth and mentions a network server for Android integration [21] [22]. This means you'll likely be testing networked features. It can help to have both projects open side by side as we set up, to tweak one and immediately see the effect on the other. For instance, if you change a data format in the Android app's `DataEvent` class, you can run the Windows app and quickly see if it parses it correctly (since the Windows app likely expects certain JSON or LSL streams). Developing in one monorepo makes it easier to keep such changes in sync – you can even enforce this by writing integration tests (maybe a Python script that simulates an Android device's data to the Windows app).

- **Documentation and Issues:** Consider using a single issue tracker (like the GitHub Issues of the monorepo) with labels for Android/Windows. This way, if there's a feature that touches both, you track it in one place. JetBrains IDEs can integrate with issue trackers or at least allow putting issue IDs in commit messages etc. Not directly an IDE tip, but helps in project management.

By following these guidelines, you'll set up a robust monorepo that allows you to develop the Android and Windows components in parallel without friction. The folder structure provides clarity and separation, version control is unified but preserves each project's legacy, and your JetBrains environment is configured to give you a smooth, productive workflow on both fronts. This hybrid setup (Android + Python/C++ desktop) is highly achievable – IntelliJ IDEA is well-suited for multi-module, multi-language projects [8] [6], and features like compound run configurations make cross-platform development much easier than juggling separate repos. Good luck with your combined development of the GSR-RGBT system!

**Sources:**

- JetBrains documentation and community tips on multi-module projects and multi-language support [23] [8] [6].
- Git subtree usage for preserving history in monorepos [2] [4].
- *fyp-gsr-windows* README (Python/Qt Windows app details and dependencies) [24] [13].
- *fyp-gsr-android* README (Android app overview and prerequisites) [25] [14].
- IntelliJ IDEA compound run configurations for parallel launches [17].

---

[1] [5] Merging 2 git repos with persisting commit history
https://medhat.dev/blog/merging-2-git-repos-with-persisting-commit-history/

[2] [3] [4] Merging multiple repositories into a monorepo, while preserving history, using `git subtree` · Jamie Tanna | Software Engineer
https://www.jvt.me/posts/2018/06/01/git-subtree-monorepo/

[6] phpstorm - How can I use multiple JetBrains IDEs on same project without annoying pop-ups when switching between them? - Stack Overflow
https://stackoverflow.com/questions/38179697/how-can-i-use-multiple-jetbrains-ides-on-same-project-without-annoying-pop-ups-w

[7] [9] [10] [11] [23] Modules | IntelliJ IDEA Documentation
https://www.jetbrains.com/help/idea/creating-and-managing-modules.html

[8] How to properly configure IntelliJ IDEA with a monorepo where Java services are in subdirectories? : r/IntelliJIDEA
https://www.reddit.com/r/IntelliJIDEA/comments/1igyo1m/how_to_properly_configure_intellij_idea_with_a/

[12] [13] [15] [20] [21] [22] [24] README.md
https://github.com/buccancs/fyp-gsr-windows/blob/4cd1d1074afcabad6987ac5bec39dcc8df22a644/README.md

[14] [25] README.md
https://github.com/buccancs/fyp-gsr-android/blob/bd9828a3a2899254d1e2a1e042cdd70175637d52/README.md

[16] What's New in IntelliJ IDEA
https://www.jetbrains.com/idea/whatsnew/

[17] [18] [19] Run compound tasks | IntelliJ IDEA Documentation
https://www.jetbrains.com.cn/en-us/help/idea/run-debug-multiple.html?keymap=secondary_macos