# Milestone 6: Shared Constants and Schema Synchronization Strategies

## Goals and Overview

In this milestone, we ensure that the Android (Kotlin) app and the Python PC application **share a single source of truth** for all message formats and constant values. The goal is to prevent any inconsistency – e.g. mismatched field names or different constant values – that could cause communication errors. We will establish a unified **communication schema** and strategy for keeping constants in sync across the two languages. This makes the system robust and eliminates entire classes of bugs (such as “Android expects field X but Python sends Y”).

**Key strategies in this milestone:**

* **Use JSON for now (Protobuf later):** Continue using human-readable JSON messages for simplicity, while designing the system to allow an easy switch to Protocol Buffers in the future if needed.
* **Unified message schema:** Clearly define the structure (fields and types) of each message in one place (e.g. via a JSON schema or .proto file) so both Kotlin and Python use identical schemas when serializing/deserializing.
* **Shared constants file:** Store shared constant values (camera resolutions, calibration sizes, etc.) in a **single JSON config file**. Use a Gradle task to generate a Kotlin constants class from this file, and have Python load the same file at runtime. This guarantees both sides use the exact same values.
* **Initial handshake with versioning:** Implement a handshake message at connection start that includes a protocol version number (and any other needed info) to ensure the phone and PC are running compatible versions. The PC will log a warning if versions don’t match, catching any update mismatch early.

With these measures, we achieve cross-language consistency in data formats and constants by **design, not by chance**. Below is a step-by-step guide to implementing these strategies, including class/module breakdowns, setup instructions, and test checkpoints.

## Step 1: Continue with JSON Messages (Plan for Protobuf Later)

Currently, the Android and Python communicate using JSON over sockets. We will **continue using JSON for messaging in the short term**, as it is human-readable and quick to iterate on during development. JSON’s readability helps in debugging the message flow. However, we also plan for a future migration to a more structured binary format (Protocol Buffers) once the communication needs become more complex or performance-critical.

**Why consider Protocol Buffers (Protobuf)?** Protobuf is a binary, schema-based messaging format that is language-neutral. Its key benefit is that you define the data schema once and then auto-generate code in multiple languages, which guarantees consistency. This would eliminate manual upkeep of parallel message definitions. Protobuf is also significantly faster and smaller than JSON (often 6–10x faster serialization by various sources[[1]](https://anymindgroup.com/news/tech-blog/15380/#:~:text=,reduce%20coding%20just%20compiled%20PB)). More importantly, *“by encoding the semantics of your data once in a proto schema, you can generate classes for different languages,”* reducing boilerplate and preventing mismatches[[2]](https://anymindgroup.com/news/tech-blog/15380/#:~:text=%2A%20Schema,to%20use%20in%20RPC%20environment)[[3]](https://kiranjobmailid.medium.com/protobuf-vs-json-b2e9bc460986#:~:text=%E2%80%9Cit%20is%20not%20worth%20the,schema%20definition%20for%20data%20exchange%E2%80%9D). In other words, the real reason to use Protobuf is *“the awesome cross-language schema definition for data exchange”* rather than just speed[[3]](https://kiranjobmailid.medium.com/protobuf-vs-json-b2e9bc460986#:~:text=%E2%80%9Cit%20is%20not%20worth%20the,schema%20definition%20for%20data%20exchange%E2%80%9D).

That said, integrating Protobuf introduces additional build steps and complexity (e.g. .proto files, compiler, plugins). To keep our development velocity high, we will **stick with JSON for now** and ensure our architecture is ready for Protobuf later. This means we will:

* Continue to send JSON messages over the socket (no immediate switch to binary).
* Define our message **schema clearly in one place** (so we could easily translate it to a .proto later).
* Optionally, we might start writing a .proto file in parallel to mirror the JSON schema (without using it in production yet), as a way to future-proof. If we do this, we can use the Gradle Protobuf plugin to generate Kotlin (Java) and Python code from it for testing. The Gradle plugin can generate code in multiple languages by enabling the respective built-ins (for example, enabling both python{} and kotlin{} in the plugin config will output Python and Kotlin classes from the same .proto definition[[4]](https://anymindgroup.com/news/tech-blog/15380/#:~:text=generateProtoTasks%20,our%20case%20baceuse%20of%20kotlin)). This dual-code generation ensures the two implementations never diverge in message structure.

In summary, **JSON remains our primary message format** in this milestone. We’ll design our code in a way that a switch to Protobuf later would be straightforward – essentially by having a clear schema and using data classes/structures that mirror that schema. When we’re ready, adopting Protobuf will simply mean generating classes from the shared schema and switching the serialization layer (which is made easier by the groundwork we lay now).

## Step 2: Define a Unified Message Schema (using JSON)

Regardless of format, it’s critical to have a **unified message schema** that both the Android and Python sides adhere to exactly. In this step, we will formalize the structure of all messages (commands, responses, data payloads, etc.) in a single definition and ensure both implementations conform to it.

**Schema Documentation:** First, create a document or specification listing each message type and its fields and data types. For example, define messages like:

* **handshake** – fields: protocol\_version (int), etc.
* **start\_recording** – fields: includeThermal (bool), sessionId (string).
* **stop\_recording** – fields: (maybe just a confirmation or an ID).
* **frame\_data** – fields: whatever sensor data frames contain, etc.

By enumerating these, developers can refer to one canon when implementing on each side. As a simple approach, maintain this as a Markdown table or Wiki page in the repo. This **protocol document** must be kept up-to-date with any message changes and reviewed whenever someone modifies message fields.

**JSON Schema (optional):** For a more rigorous definition, we can write a machine-readable JSON Schema file describing our message JSON structure. JSON Schema can specify required fields, types, etc., for each message type. While not strictly required, having a JSON Schema lets us validate messages and even auto-generate code. There are tools like QuickType that *“generate Kotlin models from JSON Schema”* and similarly for Python[[5]](https://quicktype.io/kotlin#:~:text=Instantly%20generate%20Kotlin%20from%20JSON). For instance, we could create a JSON Schema that defines a StartRecording object with includeThermal (boolean) and sessionId (string), and then use QuickType to produce a Kotlin data class and a Python class for it, ensuring they have the same fields. This approach can save manual effort and prevent typos.

If maintaining a JSON Schema is too heavy for now, we can proceed by **manually implementing the schema in code** on both sides, but with discipline and tests to keep them in sync (see Step 9 on testing). The main point is: **both the Kotlin and Python code must define the messages in an identical way.**

**Kotlin Implementation:** Create data classes or serializable classes for each message. For example, using Kotlinx Serialization or Gson, define:

@Serializable  
data class StartRecordingMessage(val includeThermal: Boolean, val sessionId: String)  
  
@Serializable  
data class HandshakeMessage(val protocol\_version: Int /\*, ... other fields if any \*/)  
  
// etc. for other message types

We will also maintain a central enum or constant list of message types. For instance, a Kotlin enum class MessageType { HANDSHAKE, START\_RECORDING, STOP\_RECORDING, ... } can list all allowed message identifiers. These can map to string values if our JSON uses a "type" field. For example, MessageType.START\_RECORDING could map to the string "start\_recording". This ensures we don’t make a typo in message names when constructing or checking messages.

**Python Implementation:** Similarly, define Python representations for each message. We have a few options: - Use simple dictionaries with expected keys (lightweight, but less structured). - Define data classes (using Python’s @dataclass) or Pydantic models for each message type. For example:

from dataclasses import dataclass  
@dataclass  
class StartRecordingMessage:  
 includeThermal: bool  
 sessionId: str

This lets us create StartRecordingMessage(includeThermal=True, sessionId="abc123") and easily convert it to/from dict (using dataclasses.asdict or similar). Pydantic would even enforce types at runtime, which can catch errors if a wrong type is received.

* Maintain a Python Enum for message types, e.g. class MessageType(Enum): HANDSHAKE = "handshake"; START\_RECORDING = "start\_recording"; ... so that we refer to MessageType.START\_RECORDING.value instead of raw strings in code.

Regardless of the method, the field names in the Python classes/dicts **must exactly match** those in the Kotlin classes/JSON. For example, if Kotlin uses includeThermal (camelCase), Python should use the same casing in the JSON keys. Consistency in naming is crucial – a mismatch like include\_thermal vs includeThermal would cause one side to miss the field. We should decide on a naming convention (the example uses camelCase for JSON keys) and apply it uniformly.

**Example JSON message:** To illustrate, a JSON message to start recording might look like:

{  
 "type": "start\_recording",  
 "includeThermal": true,  
 "sessionId": "ABC123"  
}

Both sides will produce and consume this format. The Kotlin app might serialize a StartRecordingMessage data class into this JSON, including a "type" field (or send the type separately), and the Python side will parse the JSON and know that "type":"start\_recording" means it should handle it as a StartRecording command (and extract includeThermal and sessionId). By defining this structure in one place, we ensure there is no ambiguity about what keys or data types to expect.

In summary, this step is about creating a **single, agreed-upon schema** for all messages. We enforce it by: - Defining message structures in both codebases (or generating them from one definition). - Using enumerations or constants for message type names to avoid typos. - Documenting every message and field. - (Future) If we move to Protobuf, the .proto file will become the single source of truth for the schema. We could even start with a .proto now as the schema reference (since proto3 can output JSON) without changing the transport yet. Protobuf would guarantee *“all types are described in the schema,” ensuring successful deserialization across languages*[*[6]*](https://anymindgroup.com/news/tech-blog/15380/#:~:text=messages%20contain%20field%20descriptors%20instead,schema%20files%20for%20different%20languages). We could then generate Kotlin and Python classes from this one schema, so that *“once the schema is defined, you just compile it to get code for different languages”*[[2]](https://anymindgroup.com/news/tech-blog/15380/#:~:text=%2A%20Schema,to%20use%20in%20RPC%20environment). In fact, using the Gradle proto plugin, we can generate both Kotlin and Python classes from one .proto to prove they match[[4]](https://anymindgroup.com/news/tech-blog/15380/#:~:text=generateProtoTasks%20,our%20case%20baceuse%20of%20kotlin).

By having this unified schema approach, any change (adding a new message type or field) will be applied in one place and propagated to both sides, reducing the chance of divergence.

## Step 3: Implement an Initial Handshake with Version Checking

To further ensure compatibility, we introduce an **initial handshake message** when the phone and PC connect. The handshake serves to exchange version information (and potentially other metadata) before any real commands/data flow. This way, if the Android app and Python app are out-of-sync in terms of expected message schema, we detect it immediately.

**Protocol Version:** We define a constant PROTOCOL\_VERSION (e.g., an integer) that we increment whenever a breaking change is made to the messaging format. Both the Kotlin and Python code will have this constant defined (preferably also sourced from the shared constants file described later). For example, in both codebases PROTOCOL\_VERSION = 1 for the initial release.

**Handshake Message Format:** When the socket connection is established, one side (let’s choose the Android device for concreteness) will send a JSON message like:

{   
 "type": "handshake",  
 "protocol\_version": 1,  
 "device\_name": "MyAndroidDevice",   
 "app\_version": "1.0.0"  
}

The exact fields can be adjusted, but protocol\_version is the crucial one. The receiving side (Python PC) will parse this handshake. It will compare the protocol\_version in the message with its own PROTOCOL\_VERSION constant:

* If they match (both 1, in this example), it means both are running the same expected protocol schema. We can proceed normally. The PC can send back a handshake acknowledgment if desired (not strictly necessary unless we want to send PC-specific info back).
* If they **do not match**, it indicates one side is newer/older. The PC should log a clear **warning** (or error) such as: *“Protocol version mismatch: phone is v2, PC is v1. Update recommended.”* This early warning informs the developer/user that the two ends might not understand each other perfectly. In our controlled environment (likely both apps will be updated in sync, especially if in one repo), this is mainly a safety net for development mistakes. If a mismatch is detected, we could also choose to halt the session or operate in a limited compatibility mode if that’s feasible. At minimum, it will alert us to potential incompatibilities rather than failing silently or with cryptic errors later.

**Integration into Code:**

* In Kotlin, implement the handshake send right after connecting the socket. For instance, after establishing connection, do something like:

socket.send(JSONObject(mapOf(  
 "type" to "handshake",  
 "protocol\_version" to PROTOCOL\_VERSION,  
 "app\_version" to BuildConfig.VERSION\_NAME  
)).toString())

(If using a serialization library, you could construct a HandshakeMessage(protocol\_version = 1, ...) and serialize it to JSON.)

* On the Python side, when a new client connects, the first message it expects is a handshake. It should read the JSON, check msg["type"] == "handshake", then retrieve msg["protocol\_version"]. Compare that to Python’s PROTOCOL\_VERSION. Log or print a warning if they differ. If they match, proceed to send any required response or just log that handshake succeeded.

Optionally, the Python can respond with its own handshake or an ACK message:

{  
 "type": "handshake\_ack",  
 "protocol\_version": 1,  
 "server\_name": "PC-Controller",  
 "pc\_software\_version": "1.0.0"  
}

This might not be strictly necessary if both sides have the version, but it could carry additional info (like PC software version or capabilities). In a simple scenario, the handshake can be one-way with just version check.

**Additional Handshake Data:** We might include other constants or flags in the handshake if needed. For example, the Android could send a flag if it has a certain sensor or feature available. However, since our goal is consistency, most of those should already be known to the PC if both are updated together. The primary purpose remains verifying the schema version.

With the handshake in place, we have a structured start to each session. This ensures that any fundamental incompatibility is signaled upfront. It’s much better to detect “these two apps speak different dialects” at handshake than to have weird errors later when a message field is missing. This step thereby **makes debugging easier** and enforces that both sides are on the same page version-wise.

*Test checkpoint:* After implementing, try running the Android app with an older/newer Python protocol version (simulate by changing the constant) to see the warning. This will validate that the version check logic works (see the Testing section for more).

## Step 4: Share Constant Values via a Common JSON Config

Aside from message schemas, both sides of the system use certain constant values that must remain consistent (for example, sensor sampling rates, image resolutions, calibration target dimensions, etc.). To avoid the situation where a value is changed in one codebase but not the other, we will create a **single source of truth for constants** in the form of a JSON configuration file.

**Create a JSON config file** (e.g., common\_constants.json) at a location accessible to both projects (if the Android and Python are in a monorepo, this could be a top-level config/ directory or similar). This file will list all relevant constants. For clarity, organize it by categories if needed. For example:

{  
 "thermal\_resolution": [256, 192],  
 "shimmer\_sampling": 51.2,  
 "calib\_board": {  
 "squares\_x": 6,  
 "squares\_y": 9,  
 "size\_mm": 30  
 }  
}

In this example: - thermal\_resolution is a 2-element array [width, height] of the thermal camera sensor. - shimmer\_sampling could be the sampling rate of a Shimmer sensor (51.2 Hz). - calib\_board is an object grouping calibration board parameters (like a chessboard pattern of 6 by 9 squares, 30 mm size each).

Feel free to adjust the structure; the key is that **every constant that needs to be the same on both sides should live in this file**. This includes protocol-related constants like PROTOCOL\_VERSION if you want to centralize it here (or you can keep version separate since it might change more frequently). For now, version can remain a code constant since it’s tightly tied to code changes, whereas physical constants and settings go in JSON.

**Android side usage:** We have two ways to use this config on Android: 1. **Read at runtime:** Place common\_constants.json in the Android app’s assets or raw resources. At app startup, open it (using assets.open("common\_constants.json")) and parse the JSON into a data structure or fields. Then use these values throughout the app. This ensures the app uses the latest values from that file. However, reading and parsing at runtime has a minor performance cost (negligible here) and you’d need to distribute the JSON with the app. 2. **Generate code at build-time (preferred):** Use the JSON to generate a Kotlin object/class with constant fields. This way, the constants are built into the app as static values, and no runtime parsing is needed. We will detail this in Step 5. The advantage is type safety (e.g., if a value is supposed to be int vs float, the code will reflect that) and simplicity of access (just call CommonConstants.X).

**Python side usage:** The Python application can simply load this JSON file from disk at startup, since having an external config is normal for Python scripts. We’ll ensure the Python knows the path to the file (perhaps assume it’s in the working directory or specify an environment variable). Once loaded (using Python’s json module), the values can be either used directly from the dictionary or assigned to Python constants for convenience.

By using a single file: - **Consistency is guaranteed:** We cannot accidentally change a constant in one place and forget the other; there is only one place to change it. - **Easy updates:** For example, if the thermal camera resolution changes with new hardware, we edit the JSON. The Android app, after rebuilding, and the Python script, after re-running, will both see the new resolution. - **No magic numbers in code:** Both codebases will refer to human-readable names (like THERMAL\_WIDTH) rather than littering the code with 256 or 192, making the code more maintainable.

**Where to store the file:** If the Android and Python live in the same repository, put common\_constants.json in a shared folder (not only inside the Android module). For instance, create a shared/ or config/ directory at the root. Our Gradle build (for Android) and Python code will both reference this location. If the projects are separate, you might store the file in one and have a script to sync it to the other, but ideally keep it version-controlled in one place to avoid divergence.

Next, we’ll cover how to integrate this JSON into each environment in detail.

## Step 5: Generate Kotlin Constants from the JSON (Gradle Build Task)

On the Android side, we will automate the incorporation of common\_constants.json into the app by generating a Kotlin source file from it during the build. This gives us compile-time constants that are guaranteed to match the JSON.

**Gradle Task for Code Generation:** In the Android app’s Gradle build script (likely app/build.gradle), create a custom task that does the following: - Reads the common\_constants.json file. - Parses it (we can use a JSON parser library in Gradle’s context; Groovy’s JsonSlurper or even just do simple string handling since the format is known). - Generates a Kotlin source file (e.g., CommonConstants.kt) with a structured representation of these constants.

For example, we might generate a file in the package com.myapp.config:

// Auto-generated from common\_constants.json. Do not edit manually.  
package com.myapp.config  
  
object CommonConstants {  
 const val THERMAL\_WIDTH: Int = 256  
 const val THERMAL\_HEIGHT: Int = 192  
 const val SHIMMER\_SAMPLING: Double = 51.2  
 object CALIB\_BOARD {  
 const val SQUARES\_X: Int = 6  
 const val SQUARES\_Y: Int = 9  
 const val SIZE\_MM: Int = 30  
 }  
}

This is an example based on the JSON above. We split thermal\_resolution into two constants for clarity (or we could make it a Pair<Int,Int> or a data class, but two constants are straightforward). We use const val for compile-time constants where appropriate (the float may be a Double or Float – here we used Double for 51.2). The nested CALIB\_BOARD object holds calibration board details.

**Implementation details:** - In Gradle (Groovy syntax), you can use JsonSlurper to parse the JSON file. Then programmatically write the Kotlin file. This can be done with simple string concatenation/writing to a file. - Register the task to run in the build process *before* Kotlin compilation. For example, preBuild.dependsOn generateConstantsTask or similar. Also add the output directory to the source sets so that Android Studio recognizes the generated file. For instance:

android.sourceSets.main.java.srcDir("$buildDir/generated/constants")

So if our task writes the file to build/generated/constants/CommonConstants.kt, it will be included in the compilation. - Alternatively, use an existing plugin like gradle-buildconfig-plugin or the built-in BuildConfig class. But those are typically for build variants and simple constants. Our approach is custom because we want to parse a JSON file. A custom task gives us flexibility.

**Integrating PROTOCOL\_VERSION:** We can consider including protocol\_version in this JSON as well, so that even the protocol version number is centralized. However, changing protocol version typically coincides with code changes that require simultaneous updates to message handling, so some teams prefer to keep it as a code constant (to ensure you don’t forget to bump it when you actually change code). It’s up to us. For now, we can keep PROTOCOL\_VERSION in code, but if we add it to JSON, our generation task would incorporate it as a constant too.

**After generation:** The Android code can simply use CommonConstants.THERMAL\_WIDTH, etc., anywhere needed. This is much safer than having scattered 256 or duplicating a constant. If tomorrow we decide the app should downsample the thermal image to 128x96, we edit one JSON file, and the next build will update the Kotlin constant accordingly.

One thing to note: since the constants are now compile-time, if the JSON changes, you **must rebuild the app** to get the new values (which is obvious but worth noting). In development that’s fine. In production, you’d anyway ship a new app version if constants change. The Python side reading the JSON will always get the latest at runtime, so it’s even more flexible.

By generating code, we also get **type checking**. For instance, if someone mistakenly writes "shimmer\_sampling": "51.2" (as a string) in the JSON, our generator can treat it as a string and maybe our Kotlin code expecting a Double will break – which is good, it surfaces the error. We should ensure the generator writes the type correctly (if a value has a decimal, treat it as Double; if integer, Int; booleans as Boolean; strings as String). This way, any type mismatch is caught at compile time on Android, forcing us to correct the JSON or adjust usage.

**Gradle Task Example (pseudo-code):**

import groovy.json.JsonSlurper  
  
def constantsFile = "$rootDir/common\_constants.json" // path to the JSON  
def outputDir = "$buildDir/generated/constants"  
def outputFile = "$outputDir/CommonConstants.kt"  
  
task generateConstants {  
 inputs.file constantsFile  
 outputs.file outputFile  
 doLast {  
 def json = new JsonSlurper().parse(new File(constantsFile))  
 new File(outputDir).mkdirs()  
 def kotlinCode = new StringBuilder()  
 kotlinCode.append("package com.myapp.config\n\n")  
 kotlinCode.append("object CommonConstants {\n")  
 // Example for thermal\_resolution  
 def therm = json.thermal\_resolution  
 kotlinCode.append(" const val THERMAL\_WIDTH: Int = ${therm[0]}\n")  
 kotlinCode.append(" const val THERMAL\_HEIGHT: Int = ${therm[1]}\n")  
 // Example for shimmer\_sampling  
 kotlinCode.append(" const val SHIMMER\_SAMPLING: Double = ${json.shimmer\_sampling}\n")  
 // Example for calib\_board  
 kotlinCode.append(" object CALIB\_BOARD {\n")  
 kotlinCode.append(" const val SQUARES\_X: Int = ${json.calib\_board.squares\_x}\n")  
 kotlinCode.append(" const val SQUARES\_Y: Int = ${json.calib\_board.squares\_y}\n")  
 kotlinCode.append(" const val SIZE\_MM: Int = ${json.calib\_board.size\_mm}\n")  
 kotlinCode.append(" }\n")  
 kotlinCode.append("}\n")  
 new File(outputFile).write(kotlinCode.toString())  
 }  
}  
preBuild.dependsOn generateConstants

*(The above is illustrative – in practice handle types carefully, but it shows the idea.)*

After adding this, when you build the Android app, the task will run and produce CommonConstants.kt. Android Studio (with sourceSets configured) will pick it up, so you can use CommonConstants in your code as if you wrote it manually.

**Don’t edit the generated file by hand.** If you need to change a constant, edit the JSON and rebuild. We might also add a comment at the top of the file indicating it’s auto-generated.

This automation ensures **Kotlin constants always match the JSON file**.

## Step 6: Load and Use Constants in Python

On the Python side, using the shared constants is straightforward. We will have the same common\_constants.json available to the Python application. The Python code will load this file at runtime and populate its constants.

For example, in a module constants.py in the Python project:

import json  
import os  
  
# Determine path to the JSON file. If the working directory is known, or use \_\_file\_\_ reference.  
json\_path = os.path.join(os.path.dirname(\_\_file\_\_), '..', 'common\_constants.json')  
with open(json\_path, 'r') as f:  
 \_config = json.load(f)  
  
# Extract values into Python constants or variables:  
THERMAL\_WIDTH = \_config["thermal\_resolution"][0]  
THERMAL\_HEIGHT = \_config["thermal\_resolution"][1]  
SHIMMER\_SAMPLING = \_config["shimmer\_sampling"]  
CALIB\_BOARD\_SQUARES\_X = \_config["calib\_board"]["squares\_x"]  
CALIB\_BOARD\_SQUARES\_Y = \_config["calib\_board"]["squares\_y"]  
CALIB\_BOARD\_SIZE\_MM = \_config["calib\_board"]["size\_mm"]  
  
# (Optionally, also define PROTOCOL\_VERSION = \_config["protocol\_version"] if we included it in JSON.)

Now elsewhere in the Python code, instead of using numeric literals, you import these constants. For instance, if building an image of the thermal camera, you can use constants.THERMAL\_WIDTH and constants.THERMAL\_HEIGHT. If configuring a sensor sampling rate, use SHIMMER\_SAMPLING from the constants.

Because the Python side reads the JSON fresh on each run (assuming we run the script on-demand), it will always use the latest values. There’s no separate build step needed for Python; just ensure the file path is correct.

A couple of considerations: - The path resolution in the example uses a relative path (assuming the Python project is structured so that the JSON file is one directory up from the constants.py module). You may need to adjust this depending on your project layout. Another way is to set an environment variable or command-line argument for the config file path. - If the Python project is eventually packaged (e.g., into an executable), you’d want to include the JSON or bake the values at package time. For development and testing, reading the external JSON is fine.

By doing this, **whenever the JSON is updated, the Python will reflect it immediately**. For example, if we change "size\_mm": 25 in the calibration config, the next run of the Python app will use 25. On the Android side, you’d rebuild the app to get the new constant (which is expected anyway if you’re making such a change).

This approach means **zero duplication** of constant values. Both systems literally draw from the same file, removing any chance of divergence.

## Step 7: Class and Module Structure Breakdown

To implement the above, here’s a breakdown of the classes/modules we will have on each side and their roles:

**Android (Kotlin) side:**

* **ProtocolMessage classes** – These are data classes or serializable classes representing each message type’s payload. e.g. StartRecordingMessage, HandshakeMessage, FrameDataMessage, etc. Each has fields exactly as defined in the schema. These classes make it easy to (de)serialize JSON using a library (like Kotlinx serialization or Moshi). For instance, we can annotate them with @Serializable (if using Kotlinx) and then do json.decodeFromString<StartRecordingMessage>(jsonString) to parse. This ensures we parse exactly the expected fields.
* **MessageType enum/constants** – An enum or sealed class listing the message identifiers (like HANDSHAKE, START\_RECORDING, etc). This can help in switch/when statements when handling incoming messages. Alternatively, if using polymorphic serialization, you might not need a separate enum because the JSON library can infer the type from a discriminator field. But having an enum for message types is useful for clarity and to avoid stringly-typed code.
* **Communication Handler** – This could be an existing class from previous milestones (perhaps a SocketManager or MessageHandler). Its job is to send and receive messages. After this milestone, it will:
* On receive: parse incoming JSON. Likely it will first examine the "type" field. For example, if type == "handshake", it knows to parse it into a HandshakeMessage class (or handle it accordingly). If type == "start\_recording", parse into StartRecordingMessage, etc. Then it calls the appropriate logic (maybe passes the data to the recording controller, etc).
* On send: when the app needs to send something (e.g., a sensor reading), it will construct the appropriate message object/class, serialize it to JSON string, and send over the socket.
* The communication handler ensures that the JSON keys it looks for or produces match the schema exactly (preferably by relying on the data classes and a JSON library to avoid manual key strings).
* It also will implement the **handshake sequence**: e.g., after socket connect, send handshake message (as described in Step 3), then wait/check for ack or version check result.
* **CommonConstants** – This is the **generated constants class** from Step 5 (or an equivalent config reader if we did runtime approach). It contains static values like THERMAL\_WIDTH. This will likely reside in a package like com.myapp.config or com.myapp.util. The generation script will place it accordingly. Other classes will import and use it wherever needed (e.g., the camera initialization code will use CommonConstants.THERMAL\_WIDTH when setting image size).
* *(Optional)* **Protocol buffer classes** – If we decide to write a .proto schema now, we would have generated classes in Kotlin (and Java) for those messages. They would typically be in a package based on the proto definition (for example, proto.Messages.StartRecording etc). These wouldn’t be used for JSON directly, but we might use them in tests or later when switching to binary. For now, consider them as **not actively used** but existing if we set it up. (We will likely not generate them until we decide to integrate Protobuf fully, to avoid confusing the codebase. Keeping the .proto as a reference is sufficient.)
* **Utilities** – If needed, we might have utility functions like fun toJson(message: Any): String or fun parseMessage(json: String): ProtocolMessage to encapsulate JSON serialization logic. However, using the library’s built-in is fine too (e.g., Kotlinx can parse directly with the class, Moshi could use a polymorphic adapter on a base interface, etc.). Another utility could be a test function to compare CommonConstants against the JSON file (to ensure generation integrity, discussed later).

All these will be integrated into the Android Studio project. Make sure to add any needed dependencies: - If using Kotlinx Serialization: add org.jetbrains.kotlinx:kotlinx-serialization-json dependency and enable the Kotlin serialization plugin. - If using Moshi or Gson: add those dependencies and perhaps annotations (@SerializedName etc., if needed for matching JSON keys exactly). - Ensure the Gradle generate task (if implemented) is wired up so that Android Studio sees CommonConstants.kt (as mentioned earlier).

**Python side:**

* **constants.py** – Module that loads common\_constants.json and exposes constants (as illustrated in Step 6). This will be imported wherever needed (for example, in the image processing module or calibration module, etc., to get those values).
* **Message classes or schemas** – Depending on our approach:
* If using simple approach: we might not create dedicated classes for each message. Instead, we handle messages as dictionaries. For instance, when we receive a JSON string, do data = json.loads(string), then check data["type"] and handle accordingly. This is fine for simpler logic.
* If using structured approach: create classes or Pydantic models for each message type. For example, a Pydantic model:
* class StartRecordingMessage(BaseModel):  
   includeThermal: bool  
   sessionId: str
* Pydantic can parse a dict into this model easily and will validate types. Similarly define others. Then you could have a mapping from message type to model class for parsing. Alternatively, use dataclasses as shown earlier and manually instantiate them from dict.
* If using an Enum for types: define class MessageType(Enum) with the same names/values as the Kotlin MessageType. This can be used to avoid string literals in code. For example:
* if data["type"] == MessageType.START\_RECORDING.value:  
   msg = StartRecordingMessage(\*\*data) # unpack dict into dataclass  
   handle\_start\_recording(msg)
* This ensures if the string is slightly wrong, it won’t match the enum and we might catch an error.
* **Communication handler** – The Python side likely has a main loop reading from the socket (or an async event). That part will:
* Receive JSON strings, decode them with json.loads.
* Look at data["type"], then dispatch to the appropriate handler function or class. For example, if type == "handshake": do\_version\_check(); elif type == "start\_recording": start\_recording\_handler(data); ... etc.
* When sending data back to Android, construct a dict or use the message classes to create the response, then json.dumps it to send. E.g., if sending a result or an acknowledgement.
* **Protocol version constant** – We can keep PROTOCOL\_VERSION = 1 in a config or directly in code (perhaps define it in constants.py too, or in a separate protocol.py). The handshake handler will use this constant for comparison.
* **(Optional) Generated Protobuf module** – If we had a .proto and ran protoc for Python, we would get a \_pb2.py module containing classes for each message. If we go that route later, the Python code could use those classes instead of dicts (and they can even output JSON via Message.ToDict() or so). For now, this is optional and not used in the JSON approach. When needed, we’ll integrate it.

**Project structure considerations:** Ensure the common\_constants.json is accessible. If the Python project is separate, you might copy the JSON file into the Python project directory as part of a release process. If in the same repo, just refer to it via a relative path. Keep the file path logic robust (maybe allow an environment variable override for flexibility).

With this breakdown, each side has a clear set of modules handling protocol concerns. Each message’s definition lives in exactly two places (Kotlin class and Python class or schema) or ideally one if code-generated. The constants live in exactly one place (the JSON and its generated artifacts). This structure will be easier to maintain as we add new message types or constants.

## Step 8: IDE and Build Configuration

Implementing the above will involve some configuration in our development tools:

**Android Studio / Gradle setup:**

* **Gradle Plugins/Deps:** If we choose to use Kotlinx Serialization for JSON, enable the Kotlin serialization plugin in Gradle (plugins { id "kotlinx-serialization" }) and add the dependency. If using Gson/Moshi, add those dependencies. Ensure the JSON parsing library is configured and you have the necessary proguard rules if needed (for release builds).
* **Gradle Generate Task:** Add the generateConstants task (from Step 5) to the app/build.gradle. After adding, sync the Gradle project. You should see the task in the Gradle tasks list. Run a build to ensure it generates CommonConstants.kt. Open the generated file to verify content matches the JSON. Mark the build/generated/constants (or your chosen dir) as "Generated Sources Root" if Android Studio doesn’t auto-detect it (the Gradle script adding to sourceSets usually handles this).
* **Protobuf (optional config):** If you decide to create a protocol.proto for the message schema:
* Include the Protobuf Gradle plugin in build.gradle (e.g., id "com.google.protobuf" version "0.8.17" as in the example[[7]](https://anymindgroup.com/news/tech-blog/15380/#:~:text=plugins%20%7B%20...%20id%20,%2F%2F%20protobuf%20plugin)).
* Configure the plugin: specify the proto source directory and enable generation for Java/Kotlin and Python. As shown in the reference, you can add:
* protobuf {  
   protoc { artifact = "com.google.protobuf:protoc:3.21.12" } // for example  
   generateProtoTasks {  
   all().each { task ->  
   task.builtins {  
   kotlin {}  
   python {}  
   }  
   }  
   }  
  }
* This will invoke the proto compiler to generate Java/Kotlin and Python code from your .proto. The Kotlin (Java) code will be in build/generated/source/proto/... and the Python code in build/extracted-protos/main/python by default[[4]](https://anymindgroup.com/news/tech-blog/15380/#:~:text=generateProtoTasks%20,our%20case%20baceuse%20of%20kotlin).
* You also need to add implementation "com.google.protobuf:protobuf-kotlin:<version>" in Gradle so that the Kotlin generated classes have the runtime library[[8]](https://anymindgroup.com/news/tech-blog/15380/#:~:text=id%20,%2F%2F%20protobuf%20plugin).
* Note: The Python code generation via this Gradle plugin will place the .py files in the build directory, which isn’t part of the Python project by itself. You’d have to copy them to your Python project or adjust how you run the Python code (you could add that path to PYTHONPATH). Given that, if we’re not using them yet, you might hold off on enabling Python generation to avoid confusion. But at least we know it’s possible to generate both from one build for consistency[[4]](https://anymindgroup.com/news/tech-blog/15380/#:~:text=generateProtoTasks%20,our%20case%20baceuse%20of%20kotlin).
* **Version control:** Add the common\_constants.json to version control. The generated Kotlin file should generally **not** be in version control (since it’s derived). You may want to add the generated file path to .gitignore to avoid accidental check-ins. The .proto file (if created) *should* be in version control, as it’s a source artifact.
* **Android Studio config:** Nothing special besides the above. The developer just needs to remember to re-run the generation if they edit the JSON. Because we wired it to Gradle’s build, a normal build will do it. If using Android Studio’s Apply Changes (which might not re-run a full build), be cautious – you might need to do a full rebuild if you change the JSON.

**Python IDE / environment:**

* **Path setup:** Ensure that when running the Python app (e.g., in PyCharm or VSCode), the working directory is set such that the relative path to common\_constants.json is correct (or adjust the code to an absolute path). For instance, if the code uses os.path.dirname(\_\_file\_\_) to find the JSON, this should work regardless of CWD. Alternatively, you can configure a PyCharm environment variable like COMMON\_CONFIG\_PATH and use that. The simplest is usually to keep the JSON alongside the Python code or at a known relative location.
* **Dependencies:** If using Pydantic for schema classes, ensure it’s in requirements.txt and installed. If just using built-in dataclass and json, no extra deps needed.
* **Protobuf tools:** If you plan to generate Python code from .proto without using Gradle, you’d need to install the protoc compiler separately and run a command like protoc --python\_out=<output\_dir> --proto\_path=<proto\_dir> protocol.proto. Also have protobuf library in requirements for the runtime. This is only if/when we adopt Protobuf for real. Currently, since we remain on JSON, you don’t need this setup yet. But it’s good to note for the future.
* **Testing in IDE:** You might create a small test script that loads constants.py and prints the values to ensure the JSON was read correctly. Also test the handshake logic (maybe by simulating a handshake JSON string through the parsing function) within the IDE’s test configuration.

**General Documentation:** It’s useful to document these configurations for other developers: - Explain in a README where to edit constants (the JSON file) and how codegen works. - Document the protocol version usage and remind that bumping it is needed on breaking changes. - If using the .proto, document how to regenerate code if done manually, etc.

By properly setting up the build and IDE configuration, we make it easy for developers to maintain this system. Android devs just edit JSON and rebuild; Python devs run the app and always get the latest config. The overhead of maintaining consistency is thus minimized by tooling.

## Step 9: Testing and Verification Checkpoints

Finally, to ensure our synchronization strategies truly work, we will establish several **tests and checkpoints**. These will catch any inconsistency early in development:

1. **Schema Consistency Test:** We can write tests to ensure that the set of message types and fields are the same on both sides. For example, a Python test could load a list of message types from the Kotlin code. How? Perhaps export the Kotlin MessageType enum names to a text file as part of the build, or have the Python test read the Kotlin source file via regex. This is a bit hacky, but even a manual check or a unit test where we hardcode expected types can help. The goal is to catch if a developer added a new message type in Kotlin but forgot to implement it in Python (or vice versa). Similarly, tests can send a sample JSON of each message and verify the other side’s parser accepts it. If we have a JSON Schema, we can validate an Android-produced JSON against the schema and ensure Python’s output would pass the same schema. These tests enforce that **no one side has messages or fields the other doesn’t** (as suggested, we want to avoid “one side has a field the other is unaware of”).
2. **Handshake Version Test:** Write a test (or perform a run) where you deliberately mismatch protocol versions to see the handling. For instance, set Android’s PROTOCOL\_VERSION = 2 while Python is 1, then run the connection. The Python log should emit the warning about version mismatch. This test confirms that the handshake mechanism is in place and works. In normal operation, you’d keep versions in sync, but this test is important when you do eventually bump the version – you can ensure the older version of the counterpart warns properly. Additionally, test the normal case: both at version 1, handshake passes with no issues.
3. **Round-Trip Message Tests:** For each message type, do an integration test of the full loop:
4. Android sends the message (JSON) and Python receives and interprets it correctly.
5. Python then possibly responds (if that message expects a response) and Android parses it correctly. You can automate this in several ways. For example, create a dummy Python server that echoes messages, and an Android instrumentation test that sends each message and verifies the echo. Or vice versa: a Python test that connects to a test instance of the Android messaging (if exposed). If setting up a full integration test harness is difficult, at least unit test the serialization/deserialization. E.g., in Kotlin, serialize a StartRecordingMessage to JSON, and in a Python unit test load that JSON and check that the fields match expected values. And the reverse: take a Python-generated JSON sample and use Kotlin’s parsing to see if it fills the data class correctly. Essentially, we want to ensure the keys and data types line up perfectly. If using the same schema or codegen, this is almost guaranteed, but writing a couple of these tests will catch any oversight (like a field name casing issue or a number vs string mistake).
6. **Constant Synchronization Test:** We should verify that common\_constants.json is correctly reflected in both applications. On Android, a simple unit test can load the JSON file at runtime (from assets or the project path in a test) and compare value by value with CommonConstants. For example, read the JSON in a test, and assert that json["calib\_board"]["size\_mm"] == CommonConstants.CALIB\_BOARD.SIZE\_MM. This test will fail if someone changed the JSON but forgot to rebuild/generate, or if the generation logic broke. On Python, it’s less needed (since Python directly uses the JSON), but we could still test that the constants in constants.py match the JSON file (basically by re-loading the JSON in the test and comparing to the imported constants). These tests ensure our single source of truth is truly in sync with the code.
7. **Unknown Field Tolerance Test:** Because we’re using JSON, it’s likely the JSON parsing on each side will ignore fields it doesn’t recognize (for example, if Android sends an extra field that Python’s code doesn’t use, Python’s json.loads will still have it in the dict; if our code doesn’t expect it, it will just be unused). We should verify this behavior. A test scenario: add a dummy field in a JSON message (simulate an Android of the future sending a new field), and ensure the current Python code doesn’t crash – it should simply ignore the unknown field. Similarly, ensure the Kotlin JSON library is configured to ignore unknown keys when deserializing into data classes. Kotlinx Serialization, for instance, has an option to ignore unknown keys. This ensures forward compatibility: newer versions can add fields without immediately breaking older ones (within reason). Write a unit test where you manually craft a JSON string with an extra field and attempt to parse it with the current data class – it should succeed (or at least fail gracefully, not corrupt other data).
8. **Performance/Load Test (optional):** While not directly about consistency, it’s worth testing the communication under load using JSON. Send large messages or rapid sequences to see if there’s any performance issue. This can inform us when we might need to switch to Protobuf. Protobuf’s binary format is more efficient; if JSON proves to be a bottleneck (high CPU or too much latency), we know we should accelerate the Protobuf integration. In a test, measure how long serialization/deserialization of a typical message or data frame takes in both Kotlin and Python. If it’s well within acceptable limits (likely it is for moderate data sizes), we’re fine. If not, that test will justify moving to the binary format sooner.
9. **Protobuf Consistency Test (future):** If we have set up a proto schema, we can test that it matches the JSON schema. For example, use the Protobuf classes to serialize a message and then compare its JSON representation (Protobuf can output JSON) with the JSON produced by our manual method. They should align. Or simply ensure that every field in the JSON schema exists in the proto schema. This is more of a sanity test to ensure when we cut over to Protobuf, we don’t accidentally drop or rename a field. This is only applicable if we maintain a proto definition in parallel.
10. **Continuous Integration (CI) checks:** Incorporate some of the above tests in CI, so that if a developer introduces a change that breaks schema sync, it’s caught immediately. For example, if someone modifies common\_constants.json but doesn’t run the generator, the Android unit test comparing JSON vs CommonConstants might fail on CI (because the developer’s local build might have been out of date). This prompts them to regenerate and commit if needed. Or if someone adds a new message type in Android and doesn’t update Python’s MessageType enum, a test that compares the lists will fail. These automated checks act as guardians of the contract between the two sides.

By following these verification steps, we ensure that our shared schema and constants remain truly synchronized throughout development. It’s much easier to fix a discrepancy caught by a unit test or handshake warning in advance than to debug why a feature isn’t working during a demo. In essence, these tests uphold the guarantee that **“field names and data types match exactly on both sides”**, which is the whole point of this milestone’s work[[9]](https://anymindgroup.com/news/tech-blog/15380/#:~:text=all%28%29.each%20%7B%20task%20,our%20case%20baceuse%20of%20kotlin).

## Conclusion

Milestone 6 establishes a robust foundation for cross-language consistency in our project. By continuing with JSON for ease of development but structuring our code around a **single shared schema and config**, we get the best of both worlds: human-readable development now, and a clear path to a schema-driven binary protocol later. Adopting a formal schema like Protobuf in the future will be trivial since we’ve already enforced consistency – indeed, using an IDL like Protobuf is a proven method to avoid mistakes in field naming/typing across languages[[10]](https://anymindgroup.com/news/tech-blog/15380/#:~:text=I%20hope%20this%20quick%20was,fast%20performance%20or%20grpc%20integration). But even with JSON, our strategy of code generation and version checks ensures no magic number or field goes out of sync.

These measures eliminate many potential bugs (for example, no more wondering if 51.2 Hz was coded as 51.2 or 512 somewhere – it’s defined once in JSON). Development will be smoother since both app and server “speak” the same structured language. As we move forward, any change to the protocol or constants will be done in a controlled, synchronized manner. In summary, we have made the system **schema-driven and self-consistent**, which will pay off in reliability and maintainability as the project grows[[2]](https://anymindgroup.com/news/tech-blog/15380/#:~:text=%2A%20Schema,to%20use%20in%20RPC%20environment).

[[1]](https://anymindgroup.com/news/tech-blog/15380/#:~:text=,reduce%20coding%20just%20compiled%20PB) [[2]](https://anymindgroup.com/news/tech-blog/15380/#:~:text=%2A%20Schema,to%20use%20in%20RPC%20environment) [[4]](https://anymindgroup.com/news/tech-blog/15380/#:~:text=generateProtoTasks%20,our%20case%20baceuse%20of%20kotlin) [[6]](https://anymindgroup.com/news/tech-blog/15380/#:~:text=messages%20contain%20field%20descriptors%20instead,schema%20files%20for%20different%20languages) [[7]](https://anymindgroup.com/news/tech-blog/15380/#:~:text=plugins%20%7B%20...%20id%20,%2F%2F%20protobuf%20plugin) [[8]](https://anymindgroup.com/news/tech-blog/15380/#:~:text=id%20,%2F%2F%20protobuf%20plugin) [[9]](https://anymindgroup.com/news/tech-blog/15380/#:~:text=all%28%29.each%20%7B%20task%20,our%20case%20baceuse%20of%20kotlin) [[10]](https://anymindgroup.com/news/tech-blog/15380/#:~:text=I%20hope%20this%20quick%20was,fast%20performance%20or%20grpc%20integration) [Tech Blog] A quick guide into Protobuf

<https://anymindgroup.com/news/tech-blog/15380/>

[[3]](https://kiranjobmailid.medium.com/protobuf-vs-json-b2e9bc460986#:~:text=%E2%80%9Cit%20is%20not%20worth%20the,schema%20definition%20for%20data%20exchange%E2%80%9D) ProtoBuf vs JSON vs FlatBuffers. Protocol buffers, also known as… | by kiran kumar | Medium

<https://kiranjobmailid.medium.com/protobuf-vs-json-b2e9bc460986>

[[5]](https://quicktype.io/kotlin#:~:text=Instantly%20generate%20Kotlin%20from%20JSON) JSON to Kotlin • quicktype

<https://quicktype.io/kotlin>