# Milestone 4 Implementation: Unified Protocol, Shared Configuration & Test Harnesses

In **Milestone 4** of the synchronized multimodal recording system, the focus is on unifying the communication protocol between the Python (PC) application and the Android device, sharing configuration data across platforms, and developing robust test harnesses for offline integration testing. This guide details a complete technical implementation plan covering: a unified JSON message schema, a shared configuration file, dynamic loading of these definitions on both platforms, and the construction of Python and Android test frameworks. The design emphasizes runtime consistency across devices, maintainable monorepo structure, and reliable testing without requiring live hardware or network connectivity.

## Unified JSON Message Schema

**Schema Definition and Message Types:** Create a single source of truth for all socket message formats by defining a *unified JSON schema* in a shared repository location (e.g. a protocol/ directory at the monorepo root). Using a monorepo with a shared schema file makes it easier to keep the Android and Python components in sync[[1]](https://stackoverflow.com/questions/56205986/sharing-json-schema-files-among-projects-with-versioning#:~:text=1.%20Having%20a%20Monorepo%20). The schema (e.g. a JSON file named message\_schema.json) will enumerate each message type used in the system along with its expected fields and data types. Each JSON message will include a distinguishing "type" field (string) and a set of type-specific fields. The major message types and their payload definitions include:

* **start\_record** – Command from PC to device to begin recording. Fields: "type": "start\_record" (string), "timestamp" (numeric, e.g. epoch milliseconds) marking when recording started, and an optional "session\_id" (string) or run identifier. No additional payload is required beyond the command itself.
* **stop\_record** – Command to cease an ongoing recording. Fields: "type": "stop\_record" (string), and "timestamp" (numeric) of the stop event. This may also include the "session\_id" if needed to correlate with the start command.
* **preview\_frame** – A live preview frame data message sent from device to PC. Fields: "type": "preview\_frame" (string), "frame\_id" (integer frame counter), "timestamp" (numeric) when the frame was captured, and "image\_data" (string) containing a frame preview image encoded in a compact form (e.g. Base64 JPEG). This allows the PC application to show a low-latency preview of the camera feed.
* **file\_chunk** – A chunk of recorded data (video or sensor recording) streamed from device to PC. Fields: "type": "file\_chunk" (string), "file\_id" (string or GUID to identify the recording file), "chunk\_index" (integer sequence number of this chunk), "total\_chunks" (integer total number of chunks, if known), and "chunk\_data" (string containing binary file data encoded as Base64). The PC app will reassemble these chunks to reconstruct the complete recording file.

Each message’s JSON schema should specify required fields and their types (e.g. string, number, boolean). For example, the schema entry for start\_record might require a numeric timestamp and string session\_id, whereas preview\_frame requires an image data field, etc. If there are additional message types (such as acknowledgements or calibration-specific messages), they should be added in the schema in a similar fashion. By documenting all message formats in one JSON schema file, we ensure both platforms adhere to the same contract for communication.

**Runtime Schema Loading on Each Platform:** Both the Python application and the Android app will **load this schema dynamically at runtime** rather than hard-coding message structures. This guarantees that any change to the message definitions propagates to both platforms without code duplication. On the Python side, loading the schema is straightforward using standard file I/O. For example, at startup the Python app can do:

import json  
with open("protocol/message\_schema.json", "r") as f:  
 MESSAGE\_SCHEMA = json.load(f)

This parses the JSON schema into a Python dictionary for use in validation and message construction. On Android, the schema file can be bundled with the app either as an asset or a raw resource. For instance, if placed in app/src/main/assets/protocol/, the app can open it with the AssetManager. A code snippet to load an asset JSON file at runtime would be:

InputStream is = getAssets().open("protocol/message\_schema.json");  
int size = is.available();  
byte[] buffer = new byte[size];  
is.read(buffer);  
is.close();  
String schemaJson = new String(buffer, "UTF-8");

This reads the JSON text from the assets into a string[[2]](https://stackoverflow.com/questions/19945411/how-can-i-parse-a-local-json-file-from-assets-folder-into-a-listview#:~:text=String%20json%20%3D%20null%3B%20try,size%5D%3B%20is.read%28buffer%29%3B%20is.close), which can then be parsed (e.g. using org.json or Gson) into a data structure (e.g. a JSONObject or equivalent model class). Similarly, if the schema is placed in the **raw resources** (e.g. res/raw/message\_schema.json), one can use Resources.openRawResource() to obtain an InputStream and read it into a string[[3]](https://stackoverflow.com/questions/32518967/how-to-get-a-json-file-from-raw-folder#:~:text=9). Both approaches are viable; using the *assets* directory offers a bit more flexibility in terms of file organization (you can keep the schema and config together in a subfolder). The key requirement is that the Android app does not hardcode message formats – it reads the schema definition at runtime so that the message structure is consistently interpreted.

**Python Schema Validation Helpers:** To enforce that all messages conform to the agreed schema, implement helper functions or classes in the Python codebase (e.g. in a module protocol/schema\_utils.py). One helper could be validate\_message(msg: dict) -> bool which checks a given message dictionary against the schema definitions. This function would verify that the "type" field exists and corresponds to a known message type in MESSAGE\_SCHEMA, and that all required fields for that type are present with the correct data types. For example, if msg["type"] == "preview\_frame", the helper can ensure that msg contains keys like "frame\_id" (int), "timestamp" (number), and "image\_data" (string) since those are mandated by the schema. **Optionally**, the Python implementation can leverage a JSON Schema validation library such as **jsonschema** for a rigorous check. The jsonschema library is a standard tool for validating JSON structures in Python[[4]](https://github.com/python-jsonschema/jsonschema#:~:text=An%20implementation%20of%20the%20JSON,). We could encode our message schema file in actual JSON Schema format (draft 7 or later) and then use jsonschema.validate(instance=msg, schema=MESSAGE\_SCHEMA) to automatically verify types and required fields. This approach adds a strong guarantee of protocol compliance on the Python side. Additionally, Python helpers can expose schema-derived constants – for instance, a helper that returns the list of all valid message types (so that the rest of the code can use if msg["type"] in get\_valid\_message\_types(): ...), or default values specified in the schema. By loading these from the schema at runtime, we avoid duplicating literal strings like "start\_record" in multiple places and reduce the chance of typos or version mismatch. In summary, the schema file and Python utilities serve to **centralize message definitions** and make it easy to validate or construct messages in a schema-driven way.

## Shared Configuration (config.json)

To complement the unified protocol, maintain a **shared configuration file** (config.json) in the same protocol/ directory of the monorepo. This JSON config will hold various constants and settings that need to be consistent between the PC and Android components – such as network parameters, device settings, UI tuning values, and calibration details. Using one shared config file ensures both platforms reference identical values for these parameters[[5]](https://github.com/upes-open/OSoC-2025-ClipSync#:~:text=,Receives%20and%20decrypts%20clipboard%20updates), improving maintainability. The configuration is structured into nested sections for clarity:

* **Network:** contains network-related settings, e.g. the TCP/IP port numbers for socket communication, host addresses, and possibly timeouts or buffer sizes. For example, "network": { "host": "192.168.0.100", "port": 5000, "protocol": "TCP" }. The *host* might be the PC’s address that the Android should connect to (or localhost if running emulator), and *port* is the listening port for commands/data. Defining it here allows changing the port in one place if needed.
* **Devices:** includes device-specific parameters and sensor settings. This may cover things like camera identifiers or resolution, microphone sample rates, or any hardware toggles. For instance, "devices": { "camera\_id": 0, "frame\_rate": 30, "mic\_sample\_rate": 44100 }. On Android, the camera\_id can be used to open the correct camera, and mic\_sample\_rate to configure audio recording. If multiple device types or multiple Android units are used, this section can list each or contain an array of device configs, but at minimum it centralizes hardware parameters.
* **UI:** covers user interface and experience settings. While the PC and Android UIs differ, some constants might be shared for coherence. For example, "UI": { "preview\_scale": 0.5, "overlay": true } might indicate the PC should scale the preview window to 50% of full resolution and draw certain overlays (and the Android might also use preview\_scale for resizing images before sending). This section can also include things like a flag to enable/disable a calibration target overlay on the camera preview, etc., that both sides need to agree on.
* **Calibration:** defines calibration-related constants. For example, if using a chessboard pattern for camera calibration, we can specify the pattern size and real dimensions here. E.g., "calibration": { "pattern\_rows": 7, "pattern\_cols": 6, "square\_size\_m": 0.0245, "error\_threshold": 1.0 }. In this example, the calibration pattern is 7×6 (7 inner corners by 6 inner corners on the chessboard) and each square is 24.5 mm (0.0245 meters) – these values would be needed by the calibration algorithm on the PC to interpret image points to real-world coordinates. The error\_threshold (perhaps in pixels) defines the acceptable reprojection error for a successful calibration run. For instance, one might consider a mean error below ~1.0 pixel as acceptable[[6]](https://alphapixeldev.com/opencv-tutorial-part-1-camera-calibration/#:~:text=A%20lower%20reprojection%20error%20indicates,computing%20the%20pixel%20extent%20that). This threshold is used in tests to automatically verify calibration quality.

Below is an illustrative snippet of how config.json might look:

{  
 "network": {  
 "host": "192.168.0.100",  
 "port": 5000  
 },  
 "devices": {  
 "camera\_id": 0,  
 "frame\_rate": 30,  
 "mic\_sample\_rate": 44100  
 },  
 "UI": {  
 "preview\_scale": 0.5,  
 "show\_calibration\_overlay": true  
 },  
 "calibration": {  
 "pattern\_rows": 7,  
 "pattern\_cols": 6,  
 "square\_size\_m": 0.0245,  
 "error\_threshold": 1.0  
 }  
}

**Using Config in Python:** The Python application will load config.json on startup and apply its values to configure runtime behavior. Similar to the schema, loading is done via a simple JSON file read (e.g. using json.load). For example:

with open("protocol/config.json", "r") as f:  
 CONFIG = json.load(f)

After this, the PC server code can read configuration values like CONFIG["network"]["port"] to know which port to listen on for incoming device connections, or CONFIG["calibration"]["pattern\_rows"] when performing calibration computations. All hardcoded constants (like default sample rates or image sizes) should be replaced by reading from this config object. This design makes it easy to adjust parameters (for instance, using a higher camera frame rate or a different calibration board) by editing the JSON, without touching application code. It also ensures consistency: the same config can be version-controlled and reviewed alongside code changes.

**Using Config in Android:** The Android app will also bundle the same config.json and load it during initialization. The file can reside in app/src/main/assets (or raw resources) just like the schema. At runtime, the app reads it and parses the JSON to, for example, a JSONObject or a custom Config data class. This provides device-specific parameters to the Android code. For instance, the app can retrieve the network host and port from the config instead of having them in code. If the PC’s IP address is known ahead (or perhaps the config is updated at install time), the Android could use CONFIG["network"]["host"] and "port" to establish the socket connection. The devices section might inform the app which sensors to use or their desired settings (useful if the app runs on different phone models or if certain sensors are optional). The calibration settings in config inform the Android as well – e.g., how to draw the overlay (if show\_calibration\_overlay is true, draw guidelines on the preview), or what pattern to detect (though in many cases, the PC might handle calibration computation, the Android might still need to know pattern size if it assists in detection or just to validate images). Just like with the schema, loading the config is done by reading the file from assets. The snippet for schema loading applies here (simply opening config.json instead)[[2]](https://stackoverflow.com/questions/19945411/how-can-i-parse-a-local-json-file-from-assets-folder-into-a-listview#:~:text=String%20json%20%3D%20null%3B%20try,size%5D%3B%20is.read%28buffer%29%3B%20is.close). Once parsed, the Android code should use the config values everywhere that was previously hardcoded (for example, replacing any SERVER\_PORT = 5000 constants with a value read from config). This shared config approach means that *any difference in environment or requirements (ports, sample rates, etc.) can be adjusted in one file that both projects consume* – greatly simplifying coordination and avoiding mismatches.

*Maintainability:* Keeping config.json and the message schema JSON in a common protocol/ folder under version control ensures that changes are atomic and transparent. For example, if a new sensor is added in the future, you can update the config (under devices) and the schema (new message types or fields) in the same commit as the code changes, and both platforms will use the new definitions. This design echoes best practices in multi-component systems: colocating shared definitions in a monorepo structure[[1]](https://stackoverflow.com/questions/56205986/sharing-json-schema-files-among-projects-with-versioning#:~:text=1.%20Having%20a%20Monorepo%20) and using data-driven configuration makes the system flexible and easier to maintain.

## Python Test Harnesses for Integration

To enable **reliable offline testing**, a suite of Python-based test harnesses will be developed under a tests/ directory (using pytest for organization). These tests simulate the presence of Android devices and verify system functionality end-to-end in a controlled environment. Instead of requiring physical devices or a network, the harness provides fake devices and data, so we can test the PC application’s behavior thoroughly. The key components of the Python test harness include:

* **Fake Android Device Simulator:** This is a lightweight **simulated Android client** that connects to the Python server over a socket and follows the defined protocol. The simulator can be implemented as a Python class or script in tests/ (for example, tests/fake\_device.py) that uses Python’s socket library to act like an Android device. In a test case, the PC server is launched (either the real server code or a test instance), then the fake device connects to the designated port (from config). Once connected, the simulator listens for commands: when the PC sends a start\_record JSON, the fake device responds as a real device would – e.g., send back a preview frame message and then begin sending a series of file\_chunk messages with dummy data. It can also acknowledge stop\_record by ceasing data transmission. The dummy data for preview\_frame could be a small static image encoded in Base64 (or even just a placeholder string simulating image data), and for file\_chunk perhaps some random bytes or a known pattern to simulate a recording. The key is that the format adheres to the schema. This harness allows testing the PC side’s ability to handle the full record lifecycle: ensuring that upon sending start\_record the PC correctly receives preview frames, assembles file chunks, and then properly finalizes when stop\_record is sent. Essentially, we are writing a stub socket server/client to mimic the device’s protocol[[7]](https://stackoverflow.com/questions/53016497/pytest-how-to-create-a-mock-socket-server-to-fake-responses-while-testing-an-ap#:~:text=). By doing so, we can run automated tests of scenarios like *“start recording, receive N preview frames and M file chunks, then stop – verify the PC saved a file and no errors occurred.”* This simulator can also inject edge-case behavior, such as delayed responses or malformed messages, to test robustness. (In unit-testing terms, this is closer to a system or integration test, focusing on the network protocol handling in the PC app.)
* **Calibration Test Suite:** To verify the calibration logic without needing manual intervention, create a test module (e.g. tests/test\_calibration.py) that uses known input data to run the calibration routine. This test will load a prepared set of **object points** and **image points** (perhaps stored in test data files or generated on the fly). Object points are the real-world coordinates of calibration pattern corners (for example, a 7x6 checkerboard with 25mm squares can be generated as (0,0,0), (0.025,0,0), ... in meters), and image points would be the corresponding pixel coordinates as they might appear in an image. We can obtain a set of image/object point pairs from a prior calibration run or even synthesize a scenario with slight noise. The test then calls the same calibration function as used in the application (likely an OpenCV calibrateCamera or similar). The result is an RMS reprojection error value and camera matrices. The test asserts that the RMS error is below the threshold defined in config["calibration"]["error\_threshold"]. For example, if our threshold is 1.0 pixel, the test will fail if the calibration error exceeds 1.0[[6]](https://alphapixeldev.com/opencv-tutorial-part-1-camera-calibration/#:~:text=A%20lower%20reprojection%20error%20indicates,computing%20the%20pixel%20extent%20that). This gives us confidence that our calibration procedure works correctly (e.g., our coordinate conversions and OpenCV usage are correct) and that our chosen threshold is meaningful (the test data should be such that it *ought* to succeed). We can also include tests for edge cases: for instance, if insufficient points are provided, the calibration function should throw an error or return a high error – the test can check that the code handles this gracefully (perhaps by catching exceptions and returning a failure status).
* **Config and Schema Integrity Tests:** Since the protocol schema and config are fundamental to the system, we will have tests to validate these files’ integrity. One test can ensure the **schema covers all message types** that the application expects. For example, if the application code has handlers for start\_record and stop\_record, the test can load message\_schema.json and assert that "start\_record" and "stop\_record" entries exist. This prevents a scenario where someone adds a new message type in code but forgets to update the schema (or vice versa). Another test can iterate through each message definition in the schema and ensure that required keys like "type" are present and that no field name duplicates exist, etc., essentially a sanity check of the schema file structure. Likewise, a config test can load config.json and verify its structure (e.g. contains top-level sections "network", "devices", etc., with expected types). If desired, we can again use a JSON Schema approach: define a JSON Schema for the config format and validate config.json against it, but a simpler approach is just hardcoding a check of keys and types in the test. Additionally, we might include cross-validation (for instance, if the config specifies a pattern\_rows and pattern\_cols, the test could assert both are > 0 and perhaps give a warning if the combination seems unusual). These tests ensure that the config and schema files are complete and consistent with the code’s expectations. They would run quickly and can be part of the normal test suite to catch configuration mistakes early.

All Python tests can be run with pytest locally. We will **not enable these integration tests in continuous integration (CI) by default**, since some of them (like the fake device or calibration tests) might be slower or require certain dependencies (e.g. OpenCV). Instead, developers or QA engineers will run them manually as needed. For example, the fake device test might open actual socket ports on the machine, which is not suitable for a headless CI environment. We can mark these tests (using pytest markers) as integration tests so they can be excluded from quick unit test runs. For instance, using @pytest.mark.integration and configuring CI to skip those, or only run unit tests by default[[8]](https://stackoverflow.com/questions/47559524/pytest-how-to-skip-tests-unless-you-declare-an-option-flag#:~:text=Pytest%20,runslow%20option). This way, the heavy tests (network simulation, etc.) are available for offline use but won’t interfere with automated build pipelines. The config/schema integrity tests, on the other hand, are fast and self-contained – those can be included in CI if desired, since they don’t depend on external systems.

## Android Instrumentation Test Hooks

On the Android side, we will add **instrumentation tests** (i.e., tests that run on an Android device or emulator using the Android testing framework) to exercise the communication logic and state management of the app. The goal is to simulate the PC-server side within the Android test environment, effectively mocking the socket connection and verifying that the Android app responds correctly to commands and sends proper messages. Achieving this involves introducing hooks or test-doubles for the network layer and checking the app’s internal state transitions. Key strategies include:

* **Mocking Socket Connections:** Rather than having the Android app truly connect over Wi-Fi to a PC in a test, we provide a mock socket or local server within the instrumentation test. One approach is to abstract the network communication in the Android code behind an interface or manager class (for example, a ConnectionManager class that normally uses java.net.Socket). In the test environment, we can swap this out with a stub implementation that behaves like a PC server. For instance, the stub could override methods to immediately return preset data. When the app under test “connects” to the socket (the mock), the test code can feed it a JSON string for a command. As an example, we could trigger the app’s networking component to receive a start\_record message (simulating what the PC would send) and then verify that the app transitions to the recording state and starts producing outgoing messages. By using dependency injection or a service locator pattern for the socket, the instrumentation test can insert a fake server that simply runs in the same process. This isolates the test from real network conditions, making it deterministic and self-contained[[9]](https://blog.hotstar.com/mocking-in-android-instrumentation-tests-bf46922fc800#:~:text=Test%20what%20you%20care%20about%3A,contained%2C%20and%20deterministic). An alternative (less preferred) approach is to use the emulator’s loopback interface to have the Android app connect to a localhost server the test sets up. For example, the instrumentation test could start a local ServerSocket on the device or use the special IP 10.0.2.2 to connect to a host-side server. However, this adds complexity and points of failure. A pure in-app mock or use of libraries like **WireMock** (commonly used for HTTP, but the concept can extend to TCP) is more straightforward for our custom protocol. The primary objective is to **eliminate the need for a real PC** during testing by simulating its role within the test environment[[9]](https://blog.hotstar.com/mocking-in-android-instrumentation-tests-bf46922fc800#:~:text=Test%20what%20you%20care%20about%3A,contained%2C%20and%20deterministic).
* **Testing Message Generation:** With a mock connection in place, we can assert that the Android app sends the correct messages when it should. For example, when the app receives a start\_record command (via the mock), it should respond by sending preview\_frame messages. In our instrumentation test, we can capture these outgoing JSON messages. If using a fake ConnectionManager, it can log or store any data the app “writes” to the socket. The test then inspects this log to verify message contents. We will check that the JSON conforms to the schema – i.e., the message type and required fields are present. Since the Android code also has access to the schema (loaded from the same message\_schema.json as the PC), the test could even load that schema within the device and perform a validation similar to the Python side. This might involve writing a small validation routine in Java/Kotlin or simply checking keys manually. For instance, after a start\_record, we expect to see at least one preview\_frame JSON string output. The test can parse that string (using org.json) and assert that it has "type":"preview\_frame", and contains "frame\_id", "timestamp", "image\_data", etc. Matching the actual content (like the base64 data) is less important than structure, unless we have defined specific behavior (e.g., maybe the first preview frame should always be an all-zero dummy image – then we could check some known placeholder). By automating these checks, we ensure the Android message generation code meets the protocol specification at all times.
* **Testing State Machine Transitions:** The Android app likely has an internal state machine for its recording logic (e.g., Idle -> Recording -> Stopping, etc.). We will write instrumentation tests to validate these transitions in response to messages and user actions. For example, one test will simulate the full recording sequence: send start\_record (via the mock socket) and then verify that the app’s state changes from “idle” to “recording” (perhaps by accessing a field or through the UI state if visible). Then, confirm that certain operations start (like camera recording or a timer). Next, send a stop\_record command and verify the app transitions to “idle” (or a “saving” state and then idle). If the app UI shows indicators (like a recording LED or status text), the test can use UI Automator or Espresso to check those as proxies for state. Additionally, we can test abnormal sequences: e.g., if stop\_record is received when not recording, the app should handle it gracefully (maybe ignore it or show an error). Or if the network disconnects unexpectedly (the mock can simulate a closed connection), ensure the app returns to a safe state. Because these tests run on an emulator or device, we can also incorporate some UI verification to ensure the end-to-end behavior. For instance, after receiving preview\_frame data, the app might update the preview image on screen – we could take a screenshot or query an ImageView in the test to confirm it updated (though verifying image content might be too granular for an automated test). The main idea is to exercise the **control flow** of the app in a realistic way: feed input messages and user events, then assert the app’s outputs (both network messages and UI or internal state) are correct.

The Android instrumentation tests will not be run as part of normal CI either (especially since they require an Android environment or emulator). They are meant to be run on demand (for example, a developer can connect a device or launch an emulator and run ./gradlew connectedAndroidTest). This manual trigger approach ensures that our CI remains stable (no flaky emulator tests) but we still have the tests available to run before releases or during development to catch regressions. We will include these tests in the codebase (likely under app/src/androidTest/), and perhaps provide documentation or scripts for running them when needed. By using mocks and controlled simulations, the tests remain **deterministic and fast** – no external dependencies should make them flaky. This aligns with good testing practice: *"tests are fast, self-contained, and deterministic"* when dependencies (like external servers) are mocked out[[9]](https://blog.hotstar.com/mocking-in-android-instrumentation-tests-bf46922fc800#:~:text=Test%20what%20you%20care%20about%3A,contained%2C%20and%20deterministic). Developers can trust these tests to consistently pass when the app behavior is correct, making it easier to refactor networking or recording logic with confidence.

## Test Execution Strategy and CI Integration

All the above tests (both Python and Android) are designed to be run **offline and on-demand**. During normal development, a developer might run the Python test suite with pytest to verify business logic and integration points. The heavier integration tests (fake device, calibration) can be included but potentially skipped by default. For instance, we might instruct to run pytest -m "not integration" in CI to skip marked tests, whereas a developer can run pytest -m integration locally to execute them. Similarly, Android instrumentation tests are typically run on a developer’s machine or a dedicated testing device rather than in headless CI. We will not enable these tests in the automated pipeline by default, to avoid introducing flakiness or requiring emulator setup in CI. Instead, they serve as **manual regression tests** that can be executed before a release or when making significant changes to the communication code. We will document how to run them (e.g. “Connect an Android device and run connectedAndroidTest task” or use an emulator).

By not running these tests continuously, we ensure our CI remains green and quick, while still reaping the benefits of having a comprehensive test harness available. The configuration (config.json) and schema files are also version-controlled and can be manually verified or even automatically linted (one could add a simple CI step to load and validate the JSON syntax or run the config/schema integrity tests). They are not meant to change frequently except when intentionally modifying system parameters or protocol, so manual review of those changes is usually sufficient.

## Ensuring Consistency and Maintainability

The above design decisions work in concert to achieve runtime consistency and ease of maintenance in a multi-platform, multimodal system. By **unifying the protocol definition** in a single JSON schema, we eliminate divergence in how each platform defines messages – the Android and Python components literally reference the same file for message formats. This guarantees that a message sent by one side can be correctly understood by the other, as both use the same schema (no more, for example, one side expecting a field that the other side isn’t sending). Keeping this schema in the monorepo’s shared folder follows the best practice of co-locating shared resources for multiple services[[1]](https://stackoverflow.com/questions/56205986/sharing-json-schema-files-among-projects-with-versioning#:~:text=1.%20Having%20a%20Monorepo%20). The dynamic loading of the schema and config means that adding a new message type or changing a parameter becomes a data update rather than a code change, reducing the chance of programmer error and making updates quicker. It also allows potential future extensibility – for instance, if we wanted to add a new platform (say an iOS client or another PC program), it could reuse the same protocol/ definitions.

The **shared configuration file** ensures all parts of the system use consistent parameters (e.g., the port number or calibration constants). This avoids hard-to-track issues where one side was using a different value than the other. For example, if the camera’s frame rate was mismatched, one might overwhelm the other with data; with a single config, both agree on 30 FPS vs 60 FPS as configured. It also centralizes tuning: to try a different port or adjust the acceptable calibration error, one edits config.json and both the PC and device adhere to it. This one-file configuration is much easier to maintain than scattering these constants across multiple codebases or config files – a point underscored by prior art in cross-device tools (even a simple PC-Android app used a shared config.json for setup[[5]](https://github.com/upes-open/OSoC-2025-ClipSync#:~:text=,Receives%20and%20decrypts%20clipboard%20updates)).

The investment in **test harnesses and automated checks** pays off by enabling reliable integration testing without needing the full physical setup. Developers can work offline (e.g., on an airplane or with no device at hand) and still run the fake device simulator to verify that their changes haven’t broken the protocol. The fake device and instrumentation tests together form a kind of *virtual lab*: we can simulate a recording session entirely in software, which is invaluable for catching issues early. For instance, if someone inadvertently changes a message field name in the Android code, the Python schema validation tests will flag it. If the PC starts expecting a new message sequence, the fake device tests can catch if the Android doesn’t comply (once the Android side is updated, the instrumentation tests on that side would ensure it does send what’s expected). In essence, the tests serve as a safety net and also as **documentation by example** – they show how the protocols are supposed to work, which is helpful for new developers joining the project.

Finally, this approach is aligned with achieving **reliable offline integration testing**. By running the PC and a device simulator on the same machine, and by running Android logic in an emulator or test harness, we can repeatedly execute full workflows (start/stop, data transfer, calibration) in an isolated environment. This not only speeds up development (no need to manually start the app and click buttons for every test) but also makes the integration more robust when we do go online with real devices. When the real Android app connects to the real PC app, they have already been tested against each other’s expected behaviors via the schema and config – so integration issues should be minimal. Any discrepancy would likely be caught by our tests (for example, if network latency issues arise, we might update our fake device to simulate delays and then improve our code accordingly).

In conclusion, Milestone 4 delivers a cohesive plan for unifying protocols and configs and establishing a strong testing foundation. By using shared JSON definitions and config files loaded at runtime, we ensure **consistency** across platforms and ease the **maintainability** in the monorepo structure[[1]](https://stackoverflow.com/questions/56205986/sharing-json-schema-files-among-projects-with-versioning#:~:text=1.%20Having%20a%20Monorepo%20). The Python and Android test harnesses (fake device, calibration checks, mock sockets, etc.) provide **reliable offline integration testing**, allowing the team to verify end-to-end functionality without always needing the full hardware setup. All these measures contribute to a more robust synchronized multimodal recording system as we move towards deployment and further milestones.

**Sources:**

1. Shobhit Chittora, *Sharing JSON Schema Files Among Projects* – Advocates using a monorepo with a shared schema folder for easier maintenance[[1]](https://stackoverflow.com/questions/56205986/sharing-json-schema-files-among-projects-with-versioning#:~:text=1.%20Having%20a%20Monorepo%20).
2. *Python jsonschema Library (GitHub)* – Standard library for validating JSON data against a schema[[4]](https://github.com/python-jsonschema/jsonschema#:~:text=An%20implementation%20of%20the%20JSON,).
3. Stack Overflow – Example of reading a JSON file from Android app assets at runtime[[2]](https://stackoverflow.com/questions/19945411/how-can-i-parse-a-local-json-file-from-assets-folder-into-a-listview#:~:text=String%20json%20%3D%20null%3B%20try,size%5D%3B%20is.read%28buffer%29%3B%20is.close).
4. GitHub – *ClipSync* project README, noting use of a shared config.json for cross-device configuration[[5]](https://github.com/upes-open/OSoC-2025-ClipSync#:~:text=,Receives%20and%20decrypts%20clipboard%20updates).
5. Stack Overflow – Advice on testing socket-based applications by simulating a stub server implementing the protocol[[7]](https://stackoverflow.com/questions/53016497/pytest-how-to-create-a-mock-socket-server-to-fake-responses-while-testing-an-ap#:~:text=).
6. Heena Satyarthi, *Mocking in Android Instrumentation Tests* – On using mocks to eliminate external dependencies for deterministic tests[[9]](https://blog.hotstar.com/mocking-in-android-instrumentation-tests-bf46922fc800#:~:text=Test%20what%20you%20care%20about%3A,contained%2C%20and%20deterministic).
7. AlphaPixel (OpenCV Calibration Tutorial) – Discussion of acceptable reprojection error thresholds in camera calibration (e.g. <0.5px good, <1px acceptable)[[6]](https://alphapixeldev.com/opencv-tutorial-part-1-camera-calibration/#:~:text=A%20lower%20reprojection%20error%20indicates,computing%20the%20pixel%20extent%20that).

[[1]](https://stackoverflow.com/questions/56205986/sharing-json-schema-files-among-projects-with-versioning#:~:text=1.%20Having%20a%20Monorepo%20) Sharing Json Schema files among projects with versioning - Stack Overflow

<https://stackoverflow.com/questions/56205986/sharing-json-schema-files-among-projects-with-versioning>

[[2]](https://stackoverflow.com/questions/19945411/how-can-i-parse-a-local-json-file-from-assets-folder-into-a-listview#:~:text=String%20json%20%3D%20null%3B%20try,size%5D%3B%20is.read%28buffer%29%3B%20is.close) java - How can I parse a local JSON file from assets folder into a ListView? - Stack Overflow

<https://stackoverflow.com/questions/19945411/how-can-i-parse-a-local-json-file-from-assets-folder-into-a-listview>

[[3]](https://stackoverflow.com/questions/32518967/how-to-get-a-json-file-from-raw-folder#:~:text=9) android - How to get a json file from raw folder? - Stack Overflow

<https://stackoverflow.com/questions/32518967/how-to-get-a-json-file-from-raw-folder>

[[4]](https://github.com/python-jsonschema/jsonschema#:~:text=An%20implementation%20of%20the%20JSON,) An implementation of the JSON Schema specification for Python

<https://github.com/python-jsonschema/jsonschema>

[[5]](https://github.com/upes-open/OSoC-2025-ClipSync#:~:text=,Receives%20and%20decrypts%20clipboard%20updates) GitHub - upes-open/OSoC-2025-ClipSync: OSoC 2025

<https://github.com/upes-open/OSoC-2025-ClipSync>

[[6]](https://alphapixeldev.com/opencv-tutorial-part-1-camera-calibration/#:~:text=A%20lower%20reprojection%20error%20indicates,computing%20the%20pixel%20extent%20that) OpenCV Tutorial Part 1 - Camera Calibration - AlphaPixel Software Development

<https://alphapixeldev.com/opencv-tutorial-part-1-camera-calibration/>

[[7]](https://stackoverflow.com/questions/53016497/pytest-how-to-create-a-mock-socket-server-to-fake-responses-while-testing-an-ap#:~:text=) python 3.x - Pytest: How to create a mock socket server to fake responses while testing an applications - Stack Overflow

<https://stackoverflow.com/questions/53016497/pytest-how-to-create-a-mock-socket-server-to-fake-responses-while-testing-an-ap>

[[8]](https://stackoverflow.com/questions/47559524/pytest-how-to-skip-tests-unless-you-declare-an-option-flag#:~:text=Pytest%20,runslow%20option) Pytest - how to skip tests unless you declare an option/flag?

<https://stackoverflow.com/questions/47559524/pytest-how-to-skip-tests-unless-you-declare-an-option-flag>

[[9]](https://blog.hotstar.com/mocking-in-android-instrumentation-tests-bf46922fc800#:~:text=Test%20what%20you%20care%20about%3A,contained%2C%20and%20deterministic) Mocking, in Android Instrumentation tests | by Heena Satyarthi | Disney+ Hotstar

<https://blog.hotstar.com/mocking-in-android-instrumentation-tests-bf46922fc800>