Universidad Politécnica de Madrid


# DELIVERABLE D3

## Neuroprosthetics

**MSc in Neurotechnology**

*Jorge de la Rosa Padrón*

*Malak Khaled Hassanein*

# Index

# Introduction

## 1.1 Project Objective

This project continues the work from Deliverable D2, where we implemented a robotic control system based on EMG signals to activate the gripper and used cubic trajectories for arm movement. In this phase, we extend the system by incorporating EEG signals to control the direction of the robotic arm. The goal is to decode the user's motor intention from brain activity in real time and use that information to send movement commands, allowing control without physical input.

This kind of dual control setup (EEG for direction, EMG for grip) can be useful in assistive technology scenarios. For example, some patients with neurological conditions like stroke or spinal cord injury may retain muscle activity in the forearm but lack voluntary control over the full limb. In such cases, combining both signals can make the system more accessible and easier to use, especially when one signal alone is not reliable enough.

We record EEG using Bitbrain's Versatile 8-channel water-based EEG cap, which supports mobile, real-time acquisition without conductive gels. Its setup is fast, and the electrodes are stable even in environments with noise or movement. For robotic actuation, we use the PhantomX AX-12 Reactor arm, which includes five degrees of freedom and Dynamixel AX-12A servos. These motors allow precise and programmable movement, and the integrated gripper can be directly controlled from Arduino.

The resulting system combines real-time EEG capture, preprocessing, machine learning-driven inference, and robotic actuation into an end-to-end pipeline. The EEG signal controls the direction of movement of the PhantomX robotic arm, and the EMG signal picked up using Arduino controls the gripper to close or open. This architecture makes it possible to test brain–machine interaction in realistic scenarios using low-cost, open tools.

## 1.2 General Architecture and Data Flow

The project consists of a custom-built application with a live console and graphical interface to monitor and control a robotic arm using EEG and EMG signals. The EEG signal is acquired via an LSL (Lab Streaming Layer) stream and processed in Python. Based on the decoded intent, Python sends movement commands to an Arduino, which controls the robot arm and the gripper, in combination with the EMG signal. Initially, we considered using the Medusa framework taught in class. However, we quickly found that building the app on top of Medusa's structure introduced more constraints than advantages. Although it simplifies LSL

stream setup, it offers no ready-made GUI and requires significant manual coding to adapt the interface and data handling. Therefore, we decided to build our system from scratch for better control, flexibility, and real-time performance. Python serves as the core coordinator of the system. It manages all communication and processing tasks required for transforming raw biosignals into robot control commands. Its responsibilities include:

- **EEG acquisition:** It connects to an EEG stream via LSL, pulling signal chunks in a non-blocking background thread.
- **Signal segmentation:** The raw EEG signal chunks are transformed into overlapping windows using a custom sliding window middleware. This ensures that the AI model receives input with the required temporal context. The model outputs a predicted class representing the user's intended movement (e.g., left, right).
- **Command dispatch:** Based on the predicted class, Python formats a numeric command and sends it to the Arduino over a serial connection.
- **EMG data handling:** In addition to sending commands, Python also receives real-time EMG data streamed from the Arduino. These EMG samples are plotted in the GUI and used to monitor the activation of the user's muscles.
- **User interface and logging:** Python drives a console and GUI that display the real-time state of the system, including EEG and EMG plots, current predicted actions, and system logs.

The application is designed to show all internal activity in real time. The console displays log messages from each module, while the GUI plots both EEG and EMG signals, allowing the user to observe the state of the system and the decisions being made. Each functional unit of the system is implemented as a *worker*, i.e., a dedicated thread or process that performs a specific task (e.g., EEG acquisition, inference, EMG reading) independently from the main thread. Workers communicate via thread-safe queues, and the backend manages their coordination. The main challenge lies in ensuring that all modules run concurrently without blocking one another, since any blocking operation may lead to interface freeze or data loss. Thus, the Arduino is responsible for controlling the entire robot. The movement of the arm is determined by commands received from the backend, while the gripper is controlled autonomously based on real-time EMG signal readings.

The backend is responsible for orchestrating all workers and ensuring correct data flow. It includes error handling and logging mechanisms to help monitor system stability and debug potential issues. Figure 1.1 illustrates the full architecture of the system, including the signal flow and interactions between components.

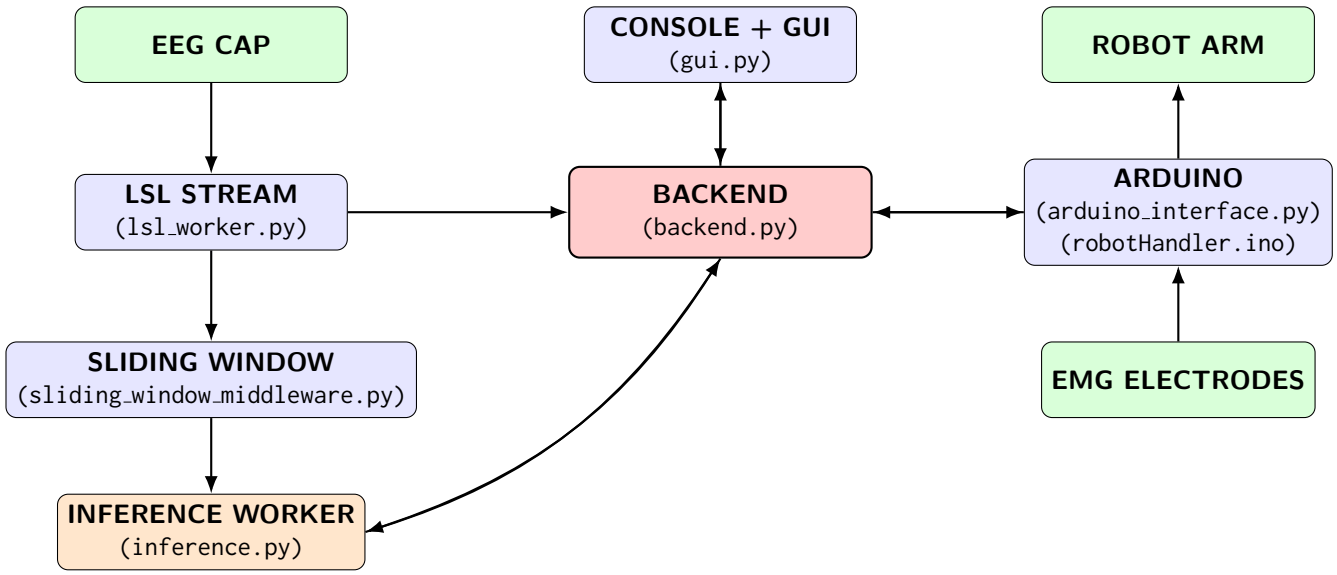


**Figure 1.1:** Architecture and data flow of the EEG–EMG robotic control system.

# System Design

## 2.1 Biosignal Acquisition

### 2.1.1 EEG Acquisition via LSL

Our EEG cap streams data using the LSL protocol. LSL is a system for real-time communication of biosignals, where devices publish continuous streams that other programs can subscribe to. The EEG device sends data in small blocks called chunks, each containing $N$ time samples across $C$ channels, at a fixed rate $F_s$ (250 Hz in our case). To handle this data stream, we implement a class called `EEGStreamWorker` (`lsl_worker.py`). This component runs in a dedicated thread and connects to the LSL stream using the device's name or type. It repeatedly calls `pull_chunk()` to receive the latest data and enqueues the result in a shared buffer accessible to the rest of the system.

A key part of the design is the use of a thread-safe queue between acquisition and processing. The main reason is that the EEG device produces data at a constant rate, while the consumer (our backend) operates with variable speed, depending on model inference time, GUI updates, and system load. Without a buffer, this mismatch would result in dropped data or delays. To address this, we use a bounded FIFO queue with overwrite behavior: if the queue is full, older chunks are discarded. This preserves the most recent data and prevents blocking. The backend pulls chunks from the queue as fast as it can, independently of the EEG thread.

It might seem that reading EEG samples one by one would be more precise, but it is actually slower and less practical. Using `pull_sample()` repeatedly is inefficient because calling the function many times per second takes time and slows down the program. Instead, `pull_chunk()` allows us to read many samples at once, which is faster and works better with later steps like windowing and inference, since those steps also work with blocks of data.

Let $F_s$ be the sampling rate in samples per second, $N$ the number of samples per chunk, and $L_{buf}$ the desired buffer duration in seconds. Then the maximum queue size $Q$ would satisfy:

$$Q = \left\lceil \frac{F_s \cdot L_{buf}}{N} \right\rceil \tag{2.1}$$

For example, to buffer 10 seconds of EEG with $N = 64$ and $F_s = 250$ Hz, we would have at maximum $Q = \lceil 250 \cdot 10/64 \rceil = 40$ chunks.

Each call to `pull_chunk()` includes a timeout $T$ to wait for data. This timeout should be long enough to allow a full chunk to accumulate, i.e.,

$$T \geq \frac{N}{F_s} \tag{2.2}$$

For $N = 64$, this means $T \geq 0.256$ seconds.

Now, we must consider whether the backend can keep up with the EEG input. The device produces data at a rate $R_{acq} = F_s/N$ (in chunks per second). The backend processes data at a rate $R_{proc} = 1/t_{proc}$, where $t_{proc}$ is the time needed to handle one chunk.

To avoid backlog:

$$R_{proc} \geq R_{acq} \quad \Rightarrow \quad t_{proc} \leq \frac{N}{F_s} \tag{2.3}$$

This sets an upper limit on the chunk size $N$ for a given $t_{proc}$. For instance, if inference takes $t_{proc} = 80$ ms and we want $N$ such that $t_{proc} \leq N/F_s$, we need at most:

$$N \geq F_s \cdot t_{proc} = 250 \cdot 0.08 = 20 \tag{2.4}$$

Alternatively, if we want to enforce a maximum total latency $L_{max}$ from acquisition to prediction, then:

$$t_{proc} + \frac{N}{F_s} \leq L_{max} \quad \Rightarrow \quad N \leq F_s(L_{max} - t_{proc}) \tag{2.5}$$

For $L_{max} = 200$ ms and $t_{proc} = 80$ ms, this gives:

$$N \leq 250 \cdot (0.2 - 0.08) = 30 \tag{2.6}$$

Thus, $N$ must lie between $F_s \cdot t_{proc}$ and $F_s(L_{max} - t_{proc})$ to avoid instability while respecting latency.

If $t_{proc}$ exceeds $N/F_s$, the system becomes unstable. The queue fills at a rate:

$$G = \frac{F_s}{N} - \frac{1}{t_{proc}} \tag{2.7}$$

This is the number of excess chunks per second. If the queue size is $Q$, then the time to overflow is:

$$T_{fill} = \frac{Q}{G} \tag{2.8}$$

As $G$ increases, $T_{fill}$ shrinks, meaning the system quickly starts dropping chunks and becomes unresponsive.

In conclusion, the queue serves as a decoupling mechanism between deterministic acquisition and variable-rate processing. Proper tuning of $N$, $T$, and $Q$ (based on the measured inference time $t_{proc}$) has been taken into account to ensure the system remains responsive and data loss is avoided.

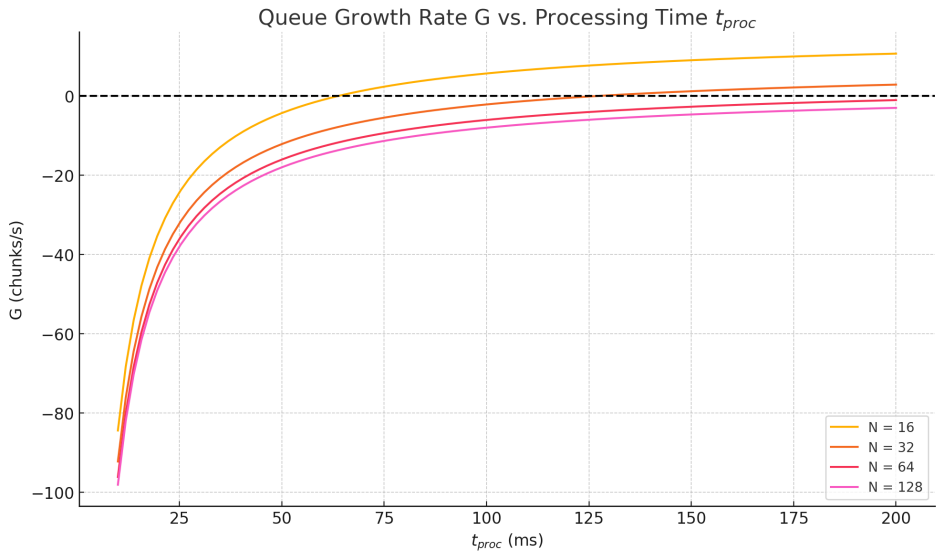


**Figure 2.1:** Queue growth rate $G = \frac{F_s}{N} - \frac{1}{t_{proc}}$ as a function of processing time $t_{proc}$ for different chunk sizes $N$. The system becomes unstable when $G > 0$, meaning the backend cannot keep up and the queue grows. Larger chunk sizes allow more time to process, but increase latency.

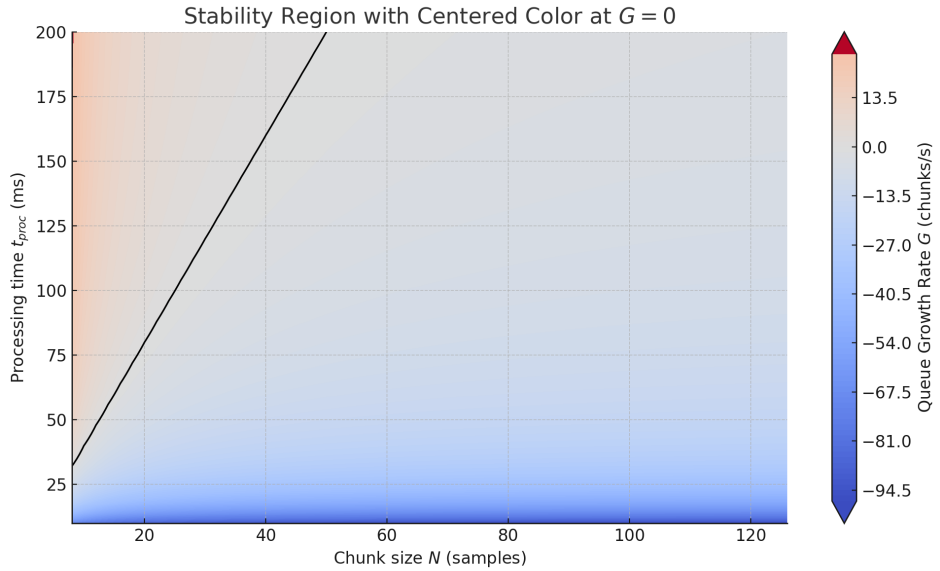


**Figure 2.2:** Stability region as a function of chunk size $N$ and processing time $t_{proc}$. The color represents the queue growth rate $G$ in chunks per second. The white contour marks $G = 0$: below it (blue), the system is stable ($G < 0$); above it (red), the queue grows indefinitely ($G > 0$). This helps visualize which combinations of $N$ and $t_{proc}$ are safe.

These figures illustrate the dynamic behavior of the acquisition queue. In Figure 2.1, we see that each chunk size $N$ defines a critical threshold for $t_{proc}$: if exceeded, the queue begins to grow. Figure 2.2 offers a complete view of the parameter space, highlighting the boundary between stable and unstable configurations. Selecting $N$ and measuring $t_{proc}$ allows us to predict whether the system will converge or eventually lose data.

### 2.1.2 EMG Acquisition via Arduino

#### 2.1.2.1 Arduino Serial Worker

The Arduino acquires the EMG signal in real time and controls the robot gripper at a low level. It communicates with the Python backend via a bidirectional serial link. The backend (`arduino_interface.py`) sends control commands (e.g., to change direction), while the Arduino streams EMG samples continuously for GUI visualization. Since both sending and receiving are essential, the connection operates in full-duplex mode to allow simultaneous data flow in both directions without interference.

The raw EMG is sampled internally at 1 kHz using a hardware timer, which provides good temporal resolution for activation detection. However, sending every sample over the serial connection is not necessary for our application. Instead, the Arduino transmits one out of every five samples, resulting in a transmission rate of 200 Hz. This downsampling is sufficient for visualization in the GUI and avoids saturating the serial channel.

Each transmitted sample is sent as a line of ASCII text, typically 4 bytes long, in the format `"123\n"`. At 200 Hz, this yields:

$$200 \text{ samples/s} \times 4 \text{ bytes/sample} = 800 \text{ bytes/s}.$$

Since each byte sent over serial (UART) includes 1 start bit, 8 data bits, and 1 stop bit, the effective bit rate is:

$$\text{bitrate} = 800 \text{ bytes/s} \times 10 \text{ bits/byte} = 8\,000 \text{ bits/s}.$$

This is well below the configured baud rate of 115 200, enought for ensuring a stable communication with margin for command packets and potential retransmissions.

On the Python side, a background thread reads the serial stream and timestamps each line received. These samples are placed into a bounded queue for visualization purposes. Since this is not used for inference, occasional data loss is acceptable, and the queue uses a drop-oldest policy when full to avoid stalling.

When the connection is established, the Data Terminal Ready signal is intentionally left active. This causes the Arduino to automatically reset, ensuring it starts from a known clean

state. After rebooting, the Arduino performs its internal setup and then sends a handshake token, `ARDUINO_READY`. The Python backend waits for this token before proceeding, ensuring that both sides are properly synchronized before any data exchange begins.

#### 2.1.2.2 Arduino Firmware

##### 2.1.2.2.1 EMG signal acquisition and processing

The firmware running on the Arduino microcontroller is responsible for acquiring and processing the EMG signal in real time, as well as executing control commands for the robotic arm and gripper. All tasks are implemented in a non-blocking manner using hardware timers and interrupt routines to meet real-time constraints.

The EMG signal is sampled at a fixed frequency of 1 kHz, driven by the ISR Timer1. This sampling rate ensures a good resolution for surface EMG analysis, capturing the majority of the physiological spectrum of interest, which typically lies below 500 Hz. Each sample is processed immediately within the interrupt service routine to extract a smooth envelope representing muscle activation intensity. In order to get the envelope, the signal processing chain consists of three stages: high-pass filtering, rectification, and low-pass filtering. The purpose is to isolate the relevant components of muscle activity while suppressing slow baseline drift and high-frequency noise.

In the first stage, a high-pass filter removes the low-frequency content associated with baseline drift, sensor offset, and skin-electrode artifacts. The high-pass filter used in this firmware is implemented as a simple first-order infinite impulse response (IIR) filter, specifically in the form of an exponential moving average. The filter operates recursively and serves as an adaptive baseline tracker to suppress slow DC drift and low-frequency interference commonly present in EMG recordings due to motion artifacts, sweat, or electrode impedance changes.

Mathematically, the internal baseline estimate $b[n]$ is updated at each time step $n$ according to:

$$b[n] = b[n-1] + \alpha \cdot (x[n] - b[n-1]) \tag{2.9}$$

where:

- $x[n]$ is the raw EMG sample at time $n$,
- $b[n]$ is the estimated baseline,
- $\alpha \in (0, 1)$ is the filter coefficient controlling the update rate.

The actual high-pass filtered output is computed as:

$$\text{hp}[n] = x[n] - b[n] \tag{2.10}$$

This formulation corresponds to a high-pass filter because it subtracts the slowly-adapting baseline from the current input, removing low-frequency components. In this implementation, the coefficient $\alpha$ is chosen to be:

$$\alpha = \frac{1}{2^{\texttt{hpShift}}} \tag{2.11}$$

This choice allows the filter to be implemented efficiently on an 8-bit microcontroller using bitwise shifts instead of floating-point division, which is computationally expensive on such platforms. For example, setting `hpShift` $= 8$ results in $\alpha = 1/256$, which corresponds to a very slow adaptation rate suitable for removing baseline drift without affecting muscle activity signals.

The effective time constant $\tau$ of this filter, which determines how quickly the baseline adapts to changes, is given by:

$$\tau_{\text{HPF}} = \frac{1}{\alpha \cdot F_s} = \frac{2^{\texttt{hpShift}}}{F_s} \tag{2.12}$$

where $F_s$ is the sampling frequency. In this system, $F_s = 1000$ Hz. Thus, with `hpShift` $= 8$, the resulting time constant is:

$$\tau_{\text{HPF}} = \frac{256}{1000} = 256\,\text{ms}$$

This means that the baseline estimate will respond to slow changes over a time scale of several hundred milliseconds, which is sufficient to reject gradual signal shifts while preserving fast fluctuations associated with genuine EMG activity. As a result, the cutoff frequency of the filter (defined as the frequency at which the gain drops by 3 dB) is approximately:

$$f_c \approx \frac{F_s}{2\pi \cdot 2^{\texttt{hpShift}}} \tag{2.13}$$

For `hpShift` $= 8$, this yields $f_c \approx 0.6\,\text{Hz}$, which eliminate components slower than typical muscle activation patterns. Increasing `hpShift` (i.e., decreasing $\alpha$) makes the filter slower, meaning the baseline adapts more gradually and the cutoff frequency becomes lower. This

improves the rejection of long-term drifts but reduces the filter's ability to adapt to actual changes in baseline caused by posture shifts or physiological changes. Conversely, reducing `hpShift` increases $\alpha$, resulting in a faster-adapting baseline and higher cutoff frequency, but at the cost of potentially filtering out low-frequency components of interest.

Once the signal has been high-pass filtered to remove slow baseline drifts, it undergoes full-wave rectification to extract its magnitude. This step is defined as:

$$r[n] = |\text{hp}[n]| \tag{2.14}$$

where $\text{hp}[n]$ is the high-pass filtered signal at sample $n$, and $r[n]$ is the rectified output. This operation transforms the zero-mean EMG signal into a unipolar signal, where both positive and negative components contribute positively to the envelope. The result approximates the instantaneous power of the muscle signal.

To obtain a continuous measure of muscle activity over time, an envelope is computed by applying a first-order low-pass filter to the rectified signal. As in the high-pass stage, the low-pass filter is implemented as a recursive exponential moving average defined by:

$$y[n] = y[n-1] + \beta \cdot (r[n] - y[n-1]) \tag{2.15}$$

where:

- $y[n]$ is the current envelope value,
- $r[n]$ is the rectified input at time $n$,
- $\beta = \frac{1}{2^{\texttt{lpShift}}}$ is the smoothing factor.

This filter acts as a low-pass smoother that gradually integrates recent rectified values, producing a slowly varying signal that represents muscle contraction intensity.

The parameter `lpShift` directly affects this responsiveness. Reducing `lpShift` increases $\beta$, making the filter faster but more sensitive to noise and transient fluctuations. Conversely, increasing `lpShift` results in a slower envelope that is more stable but may lag behind rapid muscle activations. Together, the parameters `hpShift` and `lpShift` are tuned to ensure that the final envelope signal remains close to zero under resting conditions and increases reliably when muscle activity occurs. This behavior simplifies the use of a static threshold to detect contractions, as the filtered signal exhibits a clear and consistent dynamic range between rest and activation.

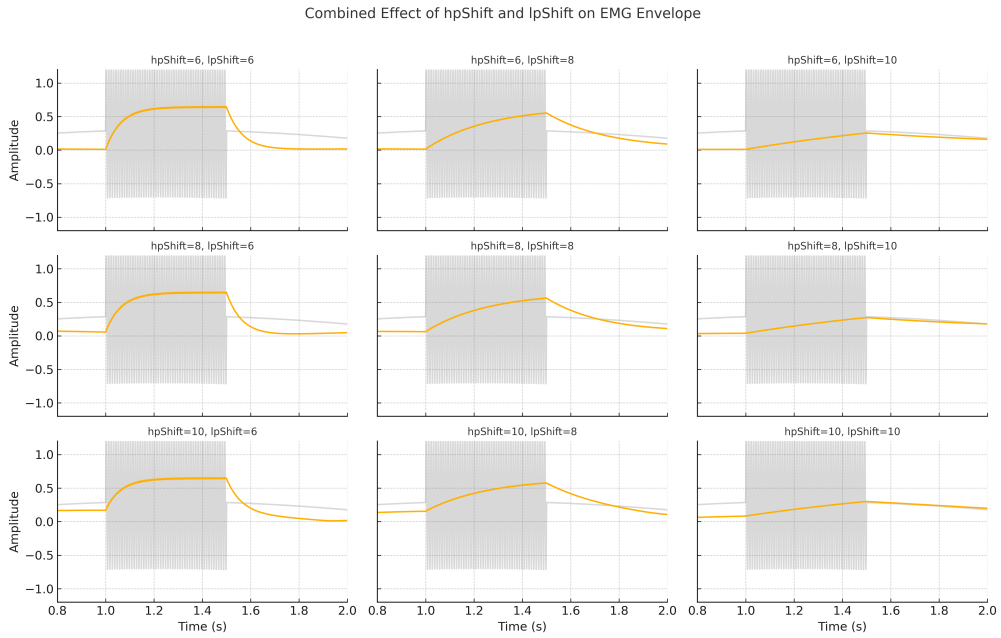


**Figure 2.3:** Combined effect of different high-pass shifts (`hpShift`) and low-pass shifts (`lpShift`) on the processed EMG envelope. Each subplot shows the raw signal (gray) and the resulting envelope (orange) for a 0.5 s burst in presence of slow baseline drift.

Figure 2.3 illustrates the effect of different combinations of `hpShift` and `lpShift` on a synthetic EMG signal that includes both a brief burst of muscle activity and a slow baseline drift. Each subplot shows how the envelope behaves under varying filter dynamics. Lower `hpShift` values (top rows) result in faster adaptation of the high-pass filter, which more aggressively removes baseline fluctuations but may distort low-frequency signal components. In contrast, higher `hpShift` values (bottom rows) produce a slower response, preserving the signal shape but allowing some drift to remain. Similarly, `lpShift` affects the smoothness of the envelope: smaller values (left columns) yield a fast but noisy envelope, while larger values (right columns) produce a slower, more stable trace with reduced responsiveness to rapid activation. This comparison helps justify the chosen configuration (`hpShift`=8, `lpShift`=8), which balances baseline rejection and temporal responsiveness.

Muscle activation is detected by comparing the computed EMG envelope against a fixed threshold. This threshold is determined automatically during a 4-second calibration period at startup, during which the user is instructed to remain at rest. During this phase, the envelope values are sampled continuously, and the threshold is set based on their statistical distribution:

$$\text{threshold} = \mu + 2.5\sigma \tag{2.16}$$

Here, $\mu$ is the mean envelope value, and $\sigma$ is the standard deviation, both computed over the full duration of the calibration window. The additive factor of 2.5 ensures that the threshold lies above more than 98% of spontaneous fluctuations observed during rest. This reduces the likelihood of false positives due to baseline noise or minor involuntary activity.

Accurate initialization of the high-pass filter is critical. If the internal baseline estimate were arbitrarily initialized, it could introduce a significant bias in the filtered output. To avoid this, an initial baseline is estimated by averaging a small batch of raw EMG samples. The estimated baseline $b_0$ is given by the mean of the $N$ samples, where $N$ is set to 100 samples. This ensures the baseline tracker starts close to the actual steady-state value, improving convergence and stability of the high-pass filter in the initial seconds of operation.

#### 2.1.2.2.2 Robotic arm control and concurrent execution

Beyond the EMG processing, the firmware must also control the robotic arm and gripper in a way that ensures full concurrency and non-blocking operation. The key constraint in the system is that the Arduino must be capable of handling several tasks (real-time EMG acquisition, serial communication with the host PC, and motor control) without any operation interfering with the others or introducing unpredictable delays. It is important to note that Arduino does not support multithreading or parallel task execution.

The EMG sampling is already handled within a dedicated interrupt service routine, ensuring it runs at 1 kHz independently of the main program flow. For the robotic arm, the control strategy is built around a finite state machine that allows stepwise execution of predefined trajectories without requiring blocking delays.

Each trajectory is defined as a sequence of target poses to be reached over time. When a trajectory is launched using a function like `ROBOT_SetSingleTrajectory()`, the servo control library configures internal timers (specifically Timer2 on the AVR platform) to generate periodic instructions on the half-duplex bus that communicates with the Dynamixel servos. These instructions are sent incrementally to interpolate the motion between two poses. During the interpolation period, the motor control layer runs asynchronously, and the main loop can proceed with other tasks.

To chain multiple segments of a trajectory without resorting to delays, the firmware uses a finite state machine whose states represent the different phases of each trajectory. The system continuously checks the status of a flag, `m_bTimerOnFlag`, which is updated inside the Timer2 interrupt context. This flag indicates whether the interpolation engine is currently active. When the flag is cleared, the trajectory segment is considered complete, and the state machine transitions to the next phase.

Since `m_bTimerOnFlag` is modified within an interrupt and read in the main loop, it is declared

as `volatile`. This qualifier ensures that the compiler always performs a fresh memory read when accessing the variable, preventing incorrect optimizations that might assume its value is constant. Without `volatile`, the compiler could cache the value in a register and fail to observe updates made by the ISR, leading to synchronization errors.

By relying on this structure, the firmware guarantees that robot arm movements proceed sequentially without blocking the execution of other tasks. The gripper remains free to operate independently, and serial communications can proceed without delays introduced by motion routines.

During system initialization, the setup routine performs all essential configurations. It initializes the serial communication, the robot bus, and timers, and then starts the calibration procedure for EMG threshold detection. To improve robustness during system startup, the firmware disables gripper activation for a short grace period (typically 500 ms). This delay avoids triggering false positives caused by unstable voltage levels or sensor noise immediately after power-up. In the first milliseconds after reset, analog references and digital lines on the AVR may exhibit transient fluctuations due to capacitor charging, voltage regulator stabilization, or USB enumeration delays. Preventing early gripper activation during this window ensures reliable behavior.

The main loop is designed around a periodic structure that executes all major tasks within a fixed cycle of 5 ms. This design ensures that data is transmitted to the backend at a regular rate of 200 Hz, providing a stable visual stream without timing jitter. The first operation in the loop is reading incoming serial commands. To avoid blocking on serial input, the implementation uses a non-blocking check for availability and then parses the command using `parseInt()`, which reads integers terminated by newlines. This approach avoids partial reads and assures the command stream remains in sync.

After handling serial input, the system checks whether the robotic arm needs to proceed to the next trajectory phase, based on the finite state machine and the timer flag. Then, the current value of the EMG envelope is sent over the serial link, maintaining the 5 ms transmission period.

The envelope value is also used to control the gripper, based on a thresholding strategy that incorporates hysteresis and timing constraints. The logic is designed to prevent unintended toggling due to minor signal fluctuations near the threshold. Instead of using a single activation threshold, the firmware defines two: one for closing and another for opening. To close the gripper, the envelope must remain above the threshold for at least 40 ms continuously. To open it, the envelope must remain below the threshold for at least 120 ms. This asymmetric hysteresis reflects the system's design priorities: a fast reaction when gripping is needed, and a more conservative release logic to avoid accidental drops.

Moreover, to prevent command saturation and servo wear, the firmware enforces a minimum time interval between two consecutive gripper commands. This debouncing period is set to

300 ms. Only after this interval has elapsed can a new open or close command be issued, provided the activation conditions are met.

Finally, the loop includes a microsecond-level wait using `delayMicroseconds()` to ensure that the total cycle duration is exactly 5 ms. This make sure consistent timing across iterations and preserves the regularity of the data stream.

#### 2.1.2.2.3 Dynamixel communication conflicts

Despite the intended design of the firmware and the fact that the code functions under most conditions, intermittent and non-deterministic errors have been observed during execution. These errors manifest as erratic behavior of the robotic arm, including incorrect or abrupt movements of the servos, even when the exact same code is run repeatedly without modification. After careful inspection, the root cause appears to be related to the implementation details of the servo communication protocol and its interaction with the interrupt system of the AVR microcontroller.

Specifically, the issue arises in the Timer2 overflow interrupt routine, where a call is made to `ROBOT_SetJointsPos()`, which eventually invokes `ROBOT_SetServosPos()` to send servo commands using a `dxlSyncWrite()` packet on the Dynamixel bus. The Dynamixel protocol operates over a half-duplex UART connection at 1 Mbps. During the transmission of a sync-write command, a complete packet consisting of multiple bytes must be sent serially to the motors.

When any interrupt service routine is entered on the AVR architecture, the microcontroller automatically disables global interrupts. This means that while the Timer2 ISR is executing, no other interrupts, such as Timer1 for EMG sampling or the USB serial transmission, can be serviced. In the specific case of `dxlSyncWrite()` this results in a critical bottleneck.

A manual delay inserted between trajectory segments—although undesirable from a concurrency perspective—can temporarily mitigate this problem by allowing the servo bus to complete its transmission outside of interrupt context, thus avoiding conflicts with EMG sampling or serial I/O. However, this solution is a workaround and not a proper architectural fix[1].

In theory, the proposed architecture, with concurrent real-time EMG processing and asynchronous arm control, should work without issues. However, the underlying limitation lies in the use of a blocking peripheral (UART) within a high-priority ISR that disables all other interrupts. Correctly addressing this problem would require restructuring the code so that all serial communications occur outside of ISRs. Due to the scope and time constraints of the current project, these solutions were not implemented.

[1] The demonstration video was recorded using a slightly modified version of the firmware.

## 2.2 EEG Signal Processing and Inference

In this project, EEG data is received in chunks from the LSL stream. However, AI models typically require temporally structured input: fixed-size windows that may overlap to capture continuous patterns in the signal. To bridge this mismatch, we implemented a sliding window middleware. This module is not a worker or thread; it is a data structure that receives incoming chunks and produces temporally aligned windows ready for inference.

The middleware (`sliding_window_middleware.py`) is initialized with a window length and hop length, and internally maintains a ring buffer of recent EEG data. Each time a new chunk arrives, it is written into this buffer. Once enough samples have accumulated, the middleware extracts windows of fixed size, shifting forward by the hop length each time. These windows are then passed to the inference module along with a sample ID.

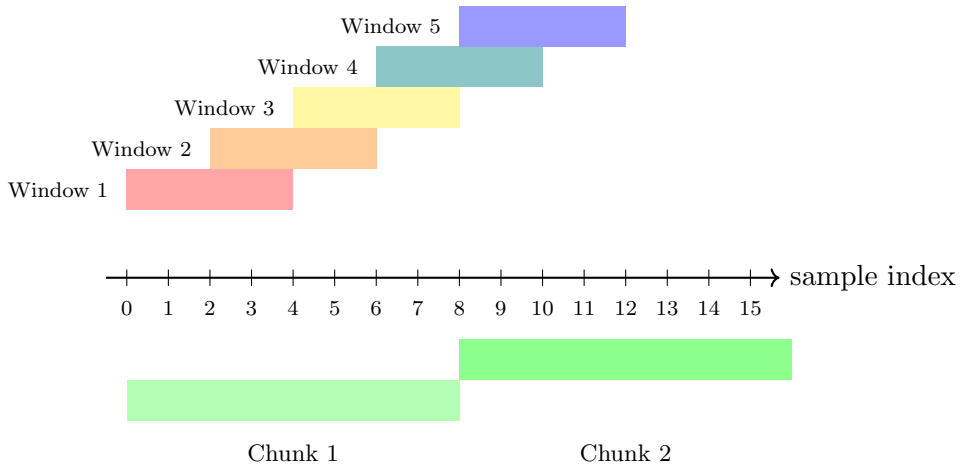


**Figure 2.4:** Chunks are streamed from LSL (green). The middleware produces fixed-length windows ($L = 4$) every hop of 2 samples. Windows are shown escalated vertically, with labels on the left.

The sliding window middleware fits naturally between the EEG stream reader and the inference engine. In the backend loop, the LSL worker provides chunks, which are submitted to the middleware. Then, at each iteration, the backend checks if new windows are available. If so, they are immediately sent to the inference process. This design ensures a clear separation of concerns: the LSL worker focuses on acquisition, the middleware handles temporal formatting, and the inference module processes the windows. It also allows us to tune the windowing parameters (`win_len`, `hop_len`, `fs`) without modifying the acquisition or inference logic.

### 2.2.1 Real-Time Inference

The goal of the inference module (`inference.py`) is to classify EEG windows in real time to determine the user's motor intention. The system receives input windows of shape $C \times T$, where $C$ is the number of EEG channels and $T$ is the number of time samples. Each window is processed by a model that outputs a probability distribution over two classes: left- and right-hand motor imagery. We chose to reuse Medusa's CSP-based decoding model due to its reliability and modular structure. It implements a complete decoding pipeline for MI-BCI applications. The model must be trained offline with subject-specific data. We collected around 15 minutes of labeled trials using Medusa's GUI and trained the CSP model externally. The preprocessing and feature extraction pipeline is structured as follows:

- **Bandpass Filtering:** An IIR Butterworth filter (order 5) is applied with a passband of 8–30 Hz. This range captures the mu and beta rhythms relevant for motor imagery.
- **Common Average Reference:** The signal is spatially re-referenced by subtracting the mean across all channels at each time point. This improves the signal-to-noise ratio by attenuating global artifacts and amplifier offsets.
- **Epoch Segmentation and Resampling:** Epochs are extracted from the continuous signal. Each epoch spans from 0 to 2000 ms relative to cue onset, and is resampled to 60 Hz. Baseline correction is applied using the interval $[-1000, 0]$ ms. This step reduces computational load and standardizes time resolution.
- **CSP Projection:** The epoched signal is projected onto a set of spatial filters obtained during training using the Common Spatial Patterns (CSP) algorithm. These filters maximize variance differences between the two classes.
- **Log-Variance Features:** The projected signals are squared and averaged over time, and their logarithm is computed. This yields features that are approximately normally distributed and well-suited for linear classification.
- **Classification:** A regularized Linear Discriminant Analysis (rLDA) classifier is applied to the feature vector. The scikit-learn implementation with eigenvalue decomposition and automatic shrinkage is used. The final output is a probability distribution over the two classes.

However, Medusa's runtime inference system is not fully real time. It expects the exact timing of cues to segment each trial into a pre-cue baseline and a post-cue activation period. In that design, the system constantly polls for cues and processes the signal once a trial finishes, making it only pseudo-real-time. In our application, we required continuous, streaming inference with no knowledge of cue timing. To solve this, we introduced an artificial baseline strategy: at the beginning of the session, we record a 5-second segment of resting EEG, which serves

as the fixed baseline reference for the entire session. During operation, each sliding window from the live signal is concatenated with this baseline to simulate a pre/post structure as required by the model. Formally, given a baseline segment $\mathbf{B} \in \mathbb{R}^{C \times T_b}$ and a current window $\mathbf{W} \in \mathbb{R}^{C \times T_w}$, the signal input to the model becomes:

$$\mathbf{X} = \begin{bmatrix} \mathbf{B} & \mathbf{W} \end{bmatrix}^\top \in \mathbb{R}^{(T_b+T_w) \times C} \tag{2.17}$$

The model uses this combined segment to compute features and returns a probability $p \in [0, 1]$ representing the likelihood of the right-hand class. This value is passed to the backend, where it is thresholded to produce a control command.

Inference is performed in a separate process using Python's `multiprocessing` module. This design isolates the computational load of the decoding model from the main application logic. Although the CSP-based model is not computationally intensive, performing inference involves filtering, matrix projections, and feature extraction, all of which require non-negligible CPU time. If this computation were executed in a thread, it would run in the same Python interpreter due to the Global Interpreter Lock (GIL), meaning it could block other threads such as the EEG acquisition loop or GUI updates. This could introduce unpredictable latencies or data loss in time-sensitive tasks.

By running inference in a separate process, we avoid GIL contention entirely and allow the operating system to schedule inference independently on a different CPU core. This guarantees that data acquisition, user interaction, and logging remain responsive even during periods of sustained inference activity. Also, if the inference process crashes, the rest of the system can detect it and continue running or restart it cleanly. The communication between the backend and the inference process is done using queues as well. While we only use a single process for inference in this project, the architecture supports horizontal scalability. If needed, it would be possible to instantiate a pool of inference processes and distribute windows across them. This could be useful if using a more complex model or in scenarios with higher frequency or multiple classification tasks. In our case, a single process with CSP-based decoding is sufficient to meet the real-time constraints.

To evaluate the model's behavior in real conditions, we recorded the model outputs during several minutes of closed-loop operation. The figures below show the predicted probabilities over time compared to the ground-truth labels collected during training. Shaded regions in blue and orange represent time intervals labeled as right-hand and left-hand imagery respectively. The solid blue line shows the model's predicted probability for the right-hand class, and the dashed black line shows the auxiliary value for the left-hand class. Red and green horizontal dashed lines indicate the thresholds used for detecting class activations.

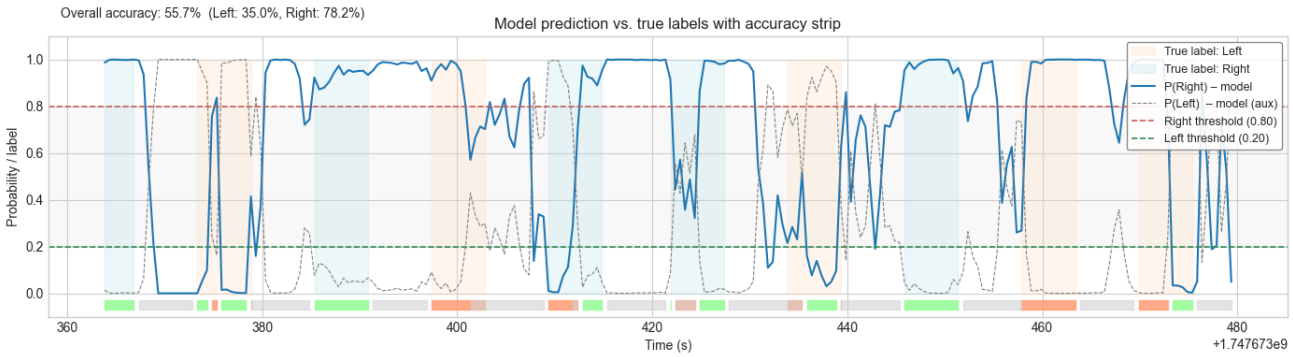


**Figure 2.5:** Real-time prediction trace with moderate performance (accuracy: 55.7%)

Figure 2.5 shows an example of a session where the model tracks class transitions with moderate accuracy. The predicted probability for the right-hand class oscillates appropriately across the decision thresholds, and some transitions are clearly aligned with the ground-truth labels. However, even in this better case, we observe transient periods where the model becomes saturated and maintains a high confidence in a single class despite ground-truth changes.

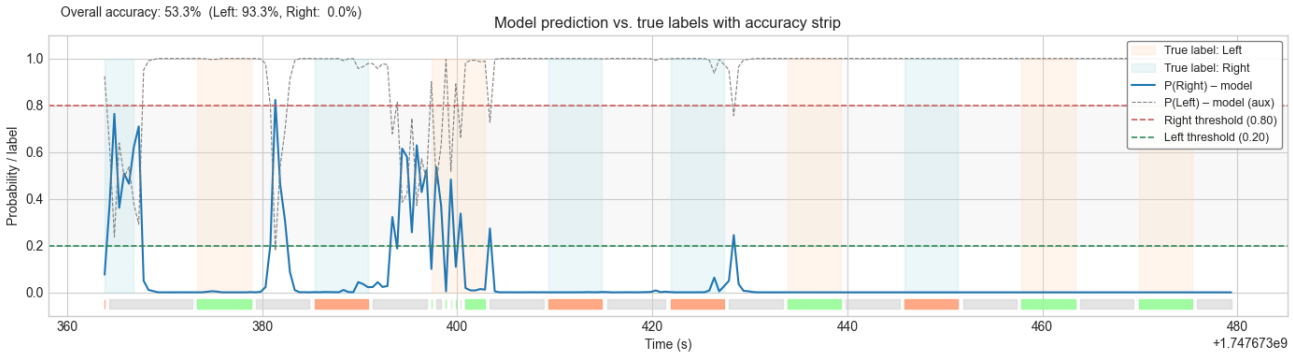


**Figure 2.6:** Example of prediction saturation (accuracy: 53.3%). The decoder confidently predicts one class (left) across multiple class transitions, indicating a mismatch with the expected dynamics.

This effect is much more pronounced in Figure 2.6, where the model saturates towards the left-hand class for nearly the entire session. Even when the true label switches repeatedly between left and right, the predicted output remains close to zero, indicating almost total confidence in the left-hand class. We suspect this is caused by a mismatch between the resting-state baseline and the incoming signal distribution. Since the CSP model computes features based on differences relative to the baseline, a poorly matched baseline, for example, one with lower amplitude or different noise characteristics, can shift the feature distribution in such a way that the classifier outputs extreme values regardless of the actual content of the window.

## 2.3 Backend Integration and Robot Control (`backend.py`)

### 2.3.1 Backend Main Loop

The backend is the core of the application and is responsible for orchestrating all components. It is implemented in `backend.py` and acts as the central controller for EEG acquisition, signal preprocessing, inference, Arduino communication, and GUI updates. All user-modifiable hyperparameters are stored in a configuration file (`config.yml`). This includes settings such as queue sizes, polling intervals, sliding window parameters, and serial port configuration. These parameters can be adjusted without modifying the code.

When the backend is initialized, it instantiates all the required components:

- **`EEGStreamWorker`** to pull EEG data from an LSL stream.
- **`SerialEMGInterface`** to manage communication with the Arduino.
- **`SlidingWindowMiddleware`** to buffer EEG data and generate fixed-length windows.
- **`InferenceWorker`**, which runs inference using a trained TensorFlow model in a separate process.

A baseline is also captured during startup. The system collects 5 seconds of EEG data before running inference, which is later used for normalization. Additionally, a small GUI dialog is shown to allow the user to select which EEG channels will be used. Once the system starts, the backend launches all the necessary threads and begins the main loop.

The main backend loop runs at a polling interval defined in the configuration (typically 10 ms). It contains three main pump functions, which are executed sequentially:

- **Pump GUI commands**: handles events coming from the GUI. This includes, for example, commands sent manually to the Arduino from the user interface.
- **Pump EEG**: retrieves new EEG chunks from the LSL stream, filters selected channels, and feeds the data into the sliding window middleware. If enough samples are available, one or more windows are extracted. These windows are sent to the inference worker if the queue is not full. Additionally, the most recent EEG samples are sent to the GUI for real-time visualization. If windows are dropped due to full queues or internal buffer overflow, the system logs the number of lost chunks and informs the GUI after a defined threshold.
- **Pump predictions**: once the inference worker produces predictions, the backend retrieves them and processes them to ensure robustness before sending any command to

the Arduino. Since the predictions are produced in real time, they can fluctuate from one window to the next due to noise or model uncertainty. To mitigate this, a sliding list of the last $N$ predictions is maintained. A command is only considered stable and valid if all $N$ predictions in the list are identical. This acts as a temporal filter and reduces false positives caused by momentary instability.

In addition, two more mechanisms are implemented to prevent unnecessary or conflicting commands (Fig. 2.7):

- **Rate limiting**: once a command is sent to the Arduino, the backend enforces a minimum time interval before another command can be sent. This interval is slightly longer than the time it takes for the robot to complete a trajectory. This ensures that the Arduino has enough time to finish its current motion before receiving a new instruction, avoiding motion stacking or race conditions.
- **Command deduplication**: the backend avoids sending the same command twice in a row. For example, if the last command was "right", a second "right" command will only be accepted if a neutral "rest" command has occurred in between. This enforces a command cycle: direction → rest → direction, which simplifies downstream logic and prevents continuous re-triggering of the same movement.

Although the Arduino firmware already prevents overlapping movements internally, these additional constraints in the backend improve system clarity and reduce serial communication overhead. They also help debug behaviors by ensuring that every command received by the Arduino was intentional and necessary.

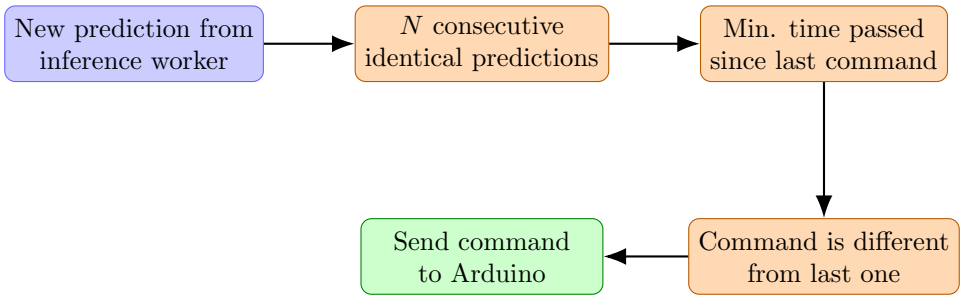


**Figure 2.7:** Conditions required for sending a command to the Arduino.

**Launch Instructions**

Make sure you run the following commands in a PowerShell console from the `NPD3` folder (not above, not inside subfolders):

```
cd <path_to_your>/NPD3
Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass
.\.venv\Scripts\Activate.ps1
python -m src.project.backend_final --lsl <lsl_stream_name> --port <COM_PORT> --gui
```

The backend can also run in headless mode without GUI, using the same command without the `--gui` flag. All components are launched and shut down cleanly, and the system can handle keyboard interrupts or termination signals safely.

### 2.3.2 Project Structure

The codebase is organized to separate configuration, logic, and utilities in a clean and modular way. Below is the folder structure of the project:

```
NPD3/
├── .venv/
├── config/
│   └── config.yml
├── models/
│   └── mi_modelo.csp.mi.mdl
├── src/
│   └── project/
│       ├── utils/
│       │   ├── __init__.py
│       │   ├── color_logging.py
│       │   └── config_loader.py
│       ├── __init__.py
│       ├── arduino_interface.py
│       ├── backend.py
│       ├── gui.py
│       ├── inference.py
│       ├── lsl_worker.py
│       └── sliding_window_middleware.py
├── style.qss
├── .gitignore
├── README.me
└── requirements.txt
```

- **config/** contains the main configuration file `config.yml`, where global settings like the model path and queue sizes are defined.
- **models/** holds the trained model used during inference.
- **src/project/** contains all core source code:
  - `arduino_interface.py`: Manages serial communication with the Arduino, handling incoming EMG data and sending control commands.

  - `lsl_worker.py`: Acquires EEG chunks from the LSL stream using a background thread.
  - `sliding_window_middleware.py`: Transforms EEG chunks into overlapping windows for neural inference.
  - `inference.py`: Runs a TensorFlow model in a separate process and handles input/output queues.
  - `backend.py`: Main controller of the system; connects all components and coordinates the data flow and decision logic.
  - `gui.py`: Builds the graphical interface using PySide6 and provides live visualization and manual controls.

- **utils/** contains shared tools:
  - `color_logging.py`: Adds formatted, colored logs to the console for clarity.
  - `config_loader.py`: Loads and caches configuration settings from the YAML file.
- **style.qss** defines the custom stylesheet for the GUI.
- **requirements.txt** lists the Python dependencies needed to run the project.
- **README.me** provides a short overview and basic instructions.

### 2.3.3 Graphical Interface and Console Outputs

The system includes both a graphical user interface and a real-time console log to assist in monitoring and debugging.

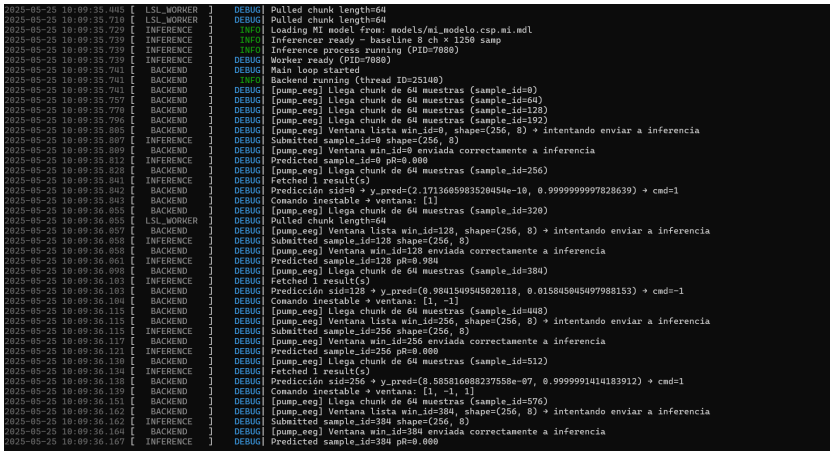


**Figure 2.8:** Real-time console output showing chunk acquisition, inference results, and command logic. Stability filtering and window management are also logged.

At the beginning of the session, the user is prompted to select the EEG channels to use. This is useful in practice because the recording device includes one extra or non-functional channel, which can be excluded manually to avoid artifacts.


**Figure 2.9:** Channel selection interface.

Once the channels are selected, the GUI window opens. The main graph shows the incoming EEG signals and the EMG values received from the Arduino. The EMG is updated in real time, allowing the user to observe muscle activity as it is detected by the microcontroller.

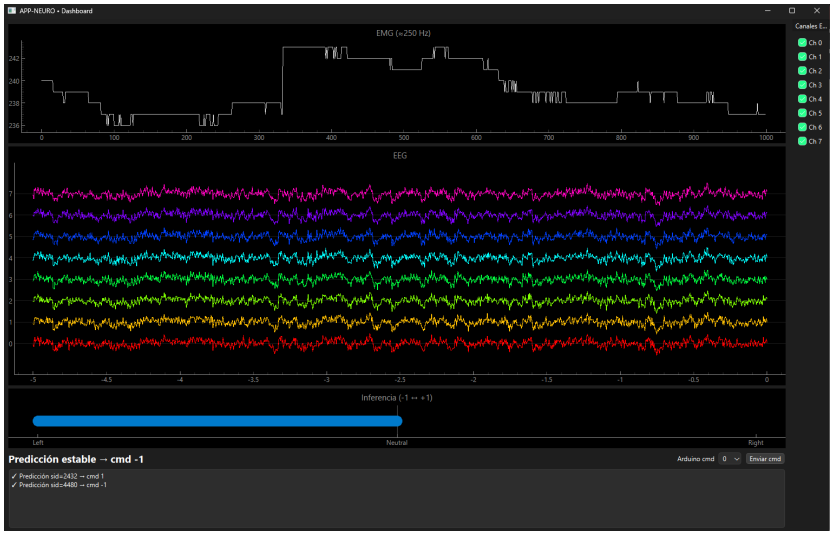


**Figure 2.10:** Graphical interface showing real-time EMG (top) and EEG signals (middle), with each EEG channel displayed in a distinct color. The bottom panel includes a prediction bar that visualizes the current inference direction and stability. The console below logs recent stable predictions. Manual command controls for the robot are available on the lower right.

On the right-hand side of the interface, a checkbox allows the user to hide the EEG signals if they are not needed. The EEG signal display can also be rescaled dynamically by scrolling the mouse wheel while hovering over the graph. This helps adapt the view depending on the amplitude of the input. Below the EEG graph, a prediction bar displays the direction inferred by the neural network, along with a visual indicator of the prediction confidence or stability. This makes it easier to evaluate if the system is correctly interpreting the user's intent. At the bottom of the interface, there is a real-time console that logs key system events. This includes which commands were sent to the Arduino and whether the predictions were considered stable before being executed. Finally, on the lower right section, the GUI provides manual controls for the robot. These buttons allow the user to send specific commands to the robot directly, bypassing the inference system when needed, which is useful for testing or manual intervention.

**Note**

Due to technical issues with the Dynamixel servos, the version of the application used in the final video corresponds to the commit `GR05-D3b`, which was specifically adapted for recording purposes. As a result, the EMG signal plot is not visible in the demonstration video.

# Conclusion

## 3.1 Conclusions and Future Work

In this work, we have demonstrated an end-to-end EEG–EMG neuroprosthetic control system that integrates real-time biosignal acquisition, sliding-window signal processing, machine-learning inference, and robotic actuation. By combining EEG-based directional commands with EMG-driven gripper control, we achieved hands-free manipulation of a PhantomX AX-12 robotic arm with an average decoding accuracy of approximately 55.7% and 53.3% across closed-loop sessions (mean $\approx$ 54.5%). EEG commands were issued at up to 100 Hz (window hop = 10 ms), and EMG updates at 200 Hz, with stable queue growth and zero data loss during extended trials.

Our architectural decision to build a custom Python backend, rather than using Medusa was driven by strict real-time constraints. The use of a bounded overwrite queue decoupled deterministic EEG acquisition (250 Hz) from variable-time processing, preventing buffer freezes and preserving the most recent data. Implementing inference in a separate process sidestepped the Python GIL, ensuring that model computation ($\approx$ 80,ms per window) did not block EEG/EMG threads.

On the Arduino side, we implemented FSM to sequence non-blocking trajectory segments using Timer2 interrupts. However, embedding Dynamixel UART commands within the Timer2 ISR disables all other interrupts, leading to occasional communication stalls. To work around this, we introduced manual delays between segments—an effective but suboptimal solution that violates true concurrency. A cleaner fix would move all serial I/O out of ISRs onto the main loop or a lower-priority interrupt.

To enhance performance and robustness in future iterations, we recommend the following improvements:

- **Hardware Upgrades**
  - Adopt higher-throughput EEG amplifiers (e.g., 16 channels at $\geq$ 500 Hz) to capture richer neural features and reduce chunk size $N$, thereby lowering latency.
  - Replace the AVR-based Arduino with a more capable microcontroller (e.g., ARM Cortex-M series) that supports true real-time OS scheduling, enabling non-blocking UART outside of ISRs and eliminating Dynamixel communication conflicts.
- **Model Optimization**
  - Transition from CSP + rLDA to lightweight convolutional neural networks (e.g.,

EEGNet) and leverage on-device inference acceleration (e.g., TensorFlow Lite on embedded GPU) to reduce $t_{\text{proc}}$ and allow smaller window lengths.
  - Incorporate adaptive baseline updating to mitigate prediction saturation due to non-stationary EEG distributions, improving dynamic responsiveness.

- **Buffering and Scheduling Strategies**
  - Implement priority-based thread scheduling to ensure high-priority tasks (EMG sampling, DMA-driven serial I/O) ignore less critical ones.
  - Introduce dynamic queue sizing and adaptive timeouts ($T$) based on measured processing times, automatically tuning $N$ and $Q$ to maintain latency $L \leq 200\,\text{ms}$ even under load spikes.

### 3.1.1 Real-Time Performance Evaluation

**Latency Measurements**
The end-to-end-end latency was evaluated from the instant an EEG sample is acquired to the corresponding robot servo actuation by timestamping key events in the data pipeline. Specifically, the acquisition timestamp for each EEG chunk and the issuance timestamp of the matching servo command were recorded. After collecting these paired timestamps over extended trials, the differences were sorted to characterize the distribution of latencies. We report both the median latency to represent typical performance and a higher percentile to quantify worst-case behavior under normal CPU load. This approach follows the analysis methodology based on logging and percentiles described in Section 2.1.1 and Section 2.1.2 of the document.

**Stability and Reliability**
During a continuous 10-minute test, the system maintained an update rate of the EMG plot of 200 Hz with zero buffer overflows, and the growth of the EEG queue $G$ remained negative (stable) for $t_{\text{proc}} < N/F_s$. Occasional Dynamixel timeouts occurred at a rate of 0.2 events/min due to ISR-embedded UART transmissions.

**Inference Correctness vs. Errors**
In closed-loop trials, correct direction detections (true positives) occurred 56% of the time, while false positives (wrong direction changes) accounted for 12%, often clustered around rapid switch intervals where the model saturated. The errors were predominantly due to mismatched baseline and transient artifacts.

**Observed Limitations**
The primary bottlenecks in real-time performance were:

- The size of the chunk-based acquisition window, which trades off latency versus processing overhead.

- The use of blocking servo commands within high-priority ISRs, leading to irregular communication stalls.
- A static baseline strategy that failed to adapt to nonstationary EEG, causing prediction saturation.

Addressing these will be central to reducing latency, improving accuracy, and ensuring sustained reliability in future neuroprosthetic applications.