

Droni e modellismo... dal punto di vista di un developer

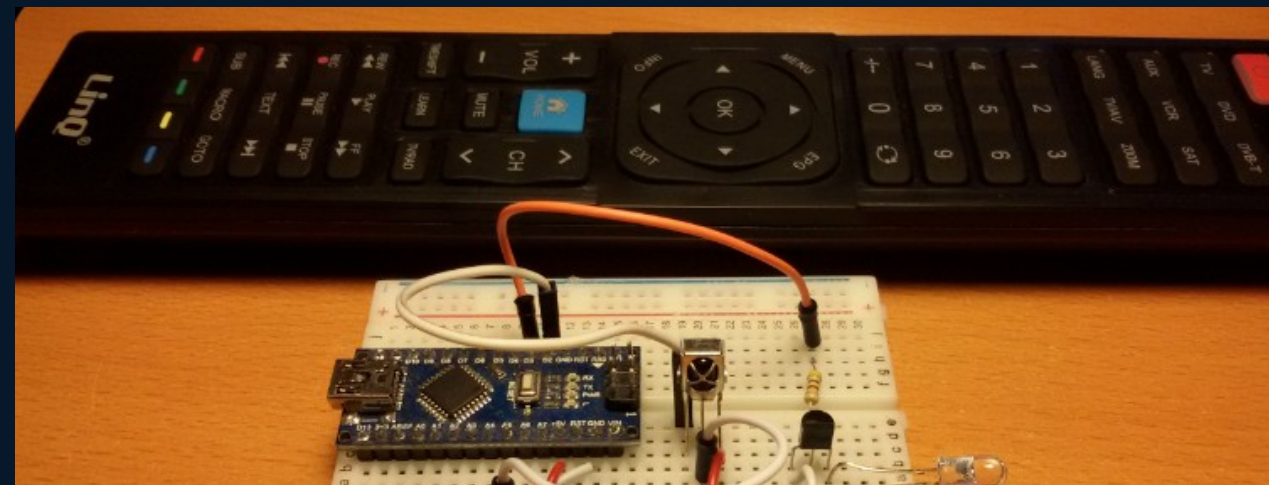
Tra i vari hobby del tempo libero, ne ho uno che unisce tecnologia, creatività e adrenalina: i droni. Grazie ad esso, posso stare all'aria aperta, senza rinunciare alla mia passione di sperimentare.



Chi sono



Appassionato di tecnologia, con una laurea in Informatica, attualmente lavoro come Fullstack Developer e consulente IT. Dedico parte del tempo libero per approfondire AI, robotica, homelab, tinkering e progetti IoT (oltre a dilettermi in alcuni sport).

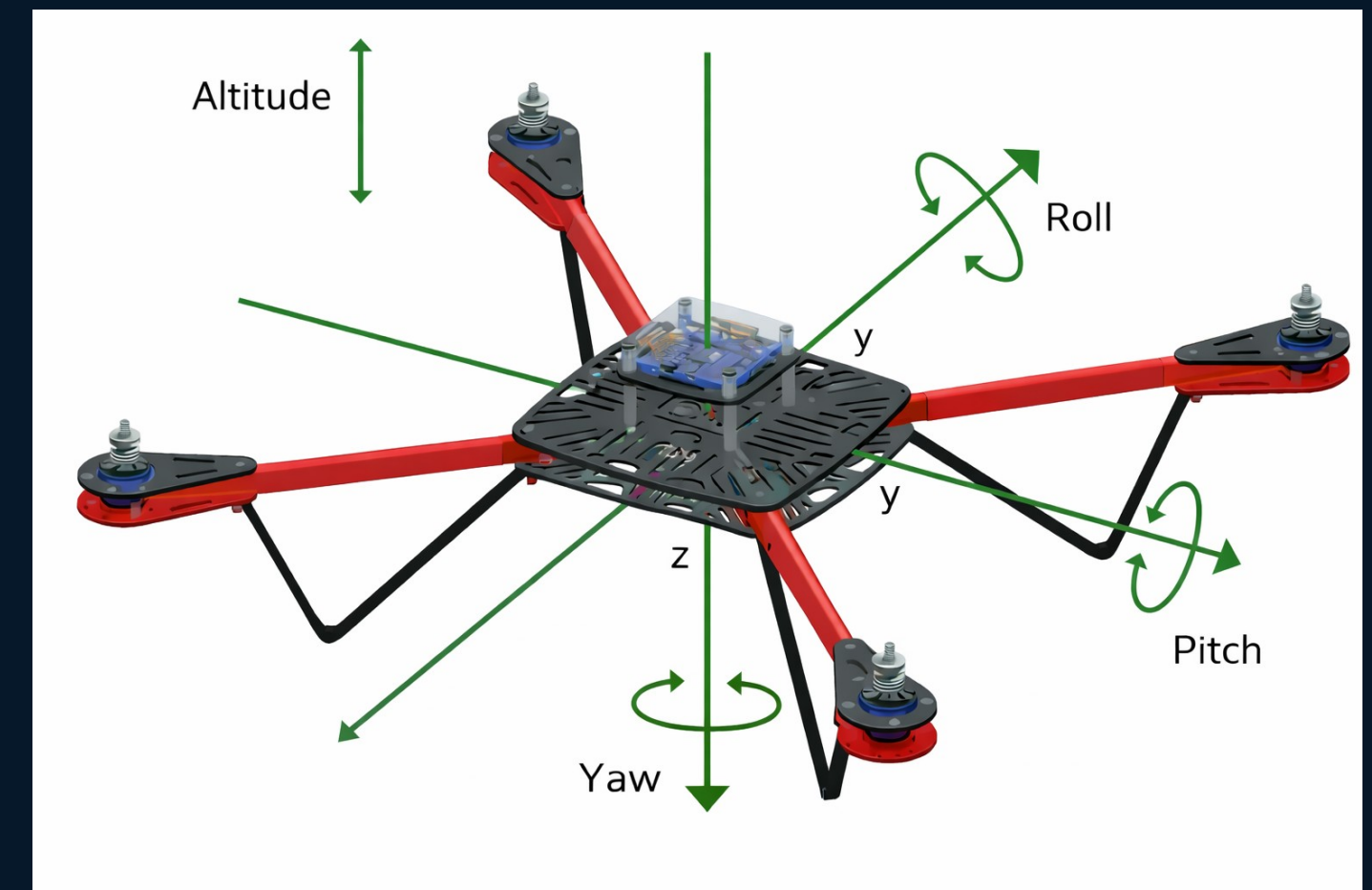
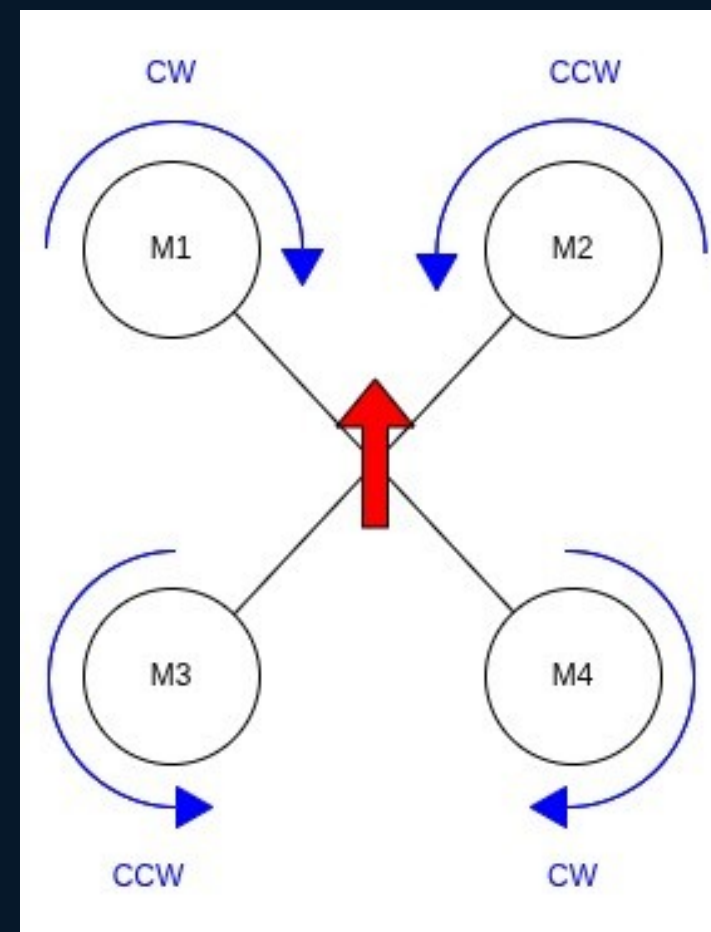
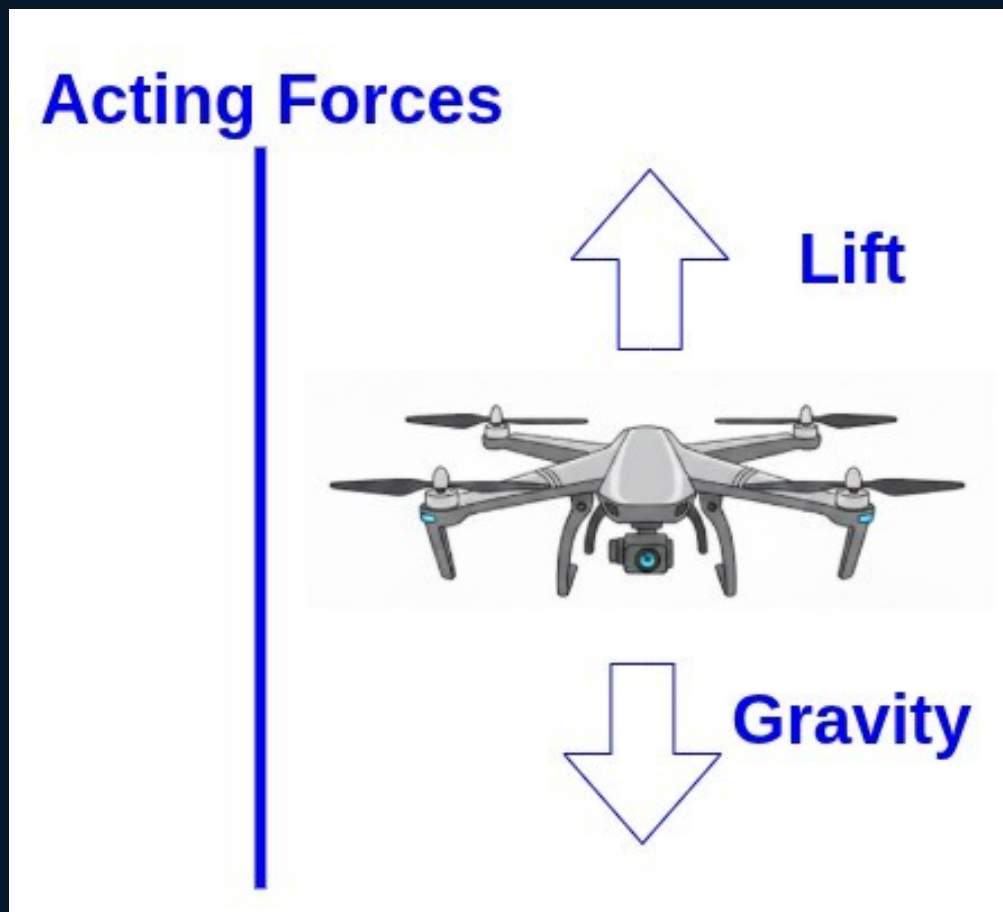


Argomenti affrontati

- Movimenti di un quadricottero
- Tipologie di droni
- hardware di un drone FPV
- Flight Stack: hardware FC, architettura software, protocolli di messaggistica e simulazioni
- Programmazione in Python e missioni di volo: simulazione con PX4 e JMavsim
- Interfacciare Mavlink con intelligenza artificiale
- Integrazione con ROS
- Idee (Brainstorming)

Quadricottero: come fa a muoversi?

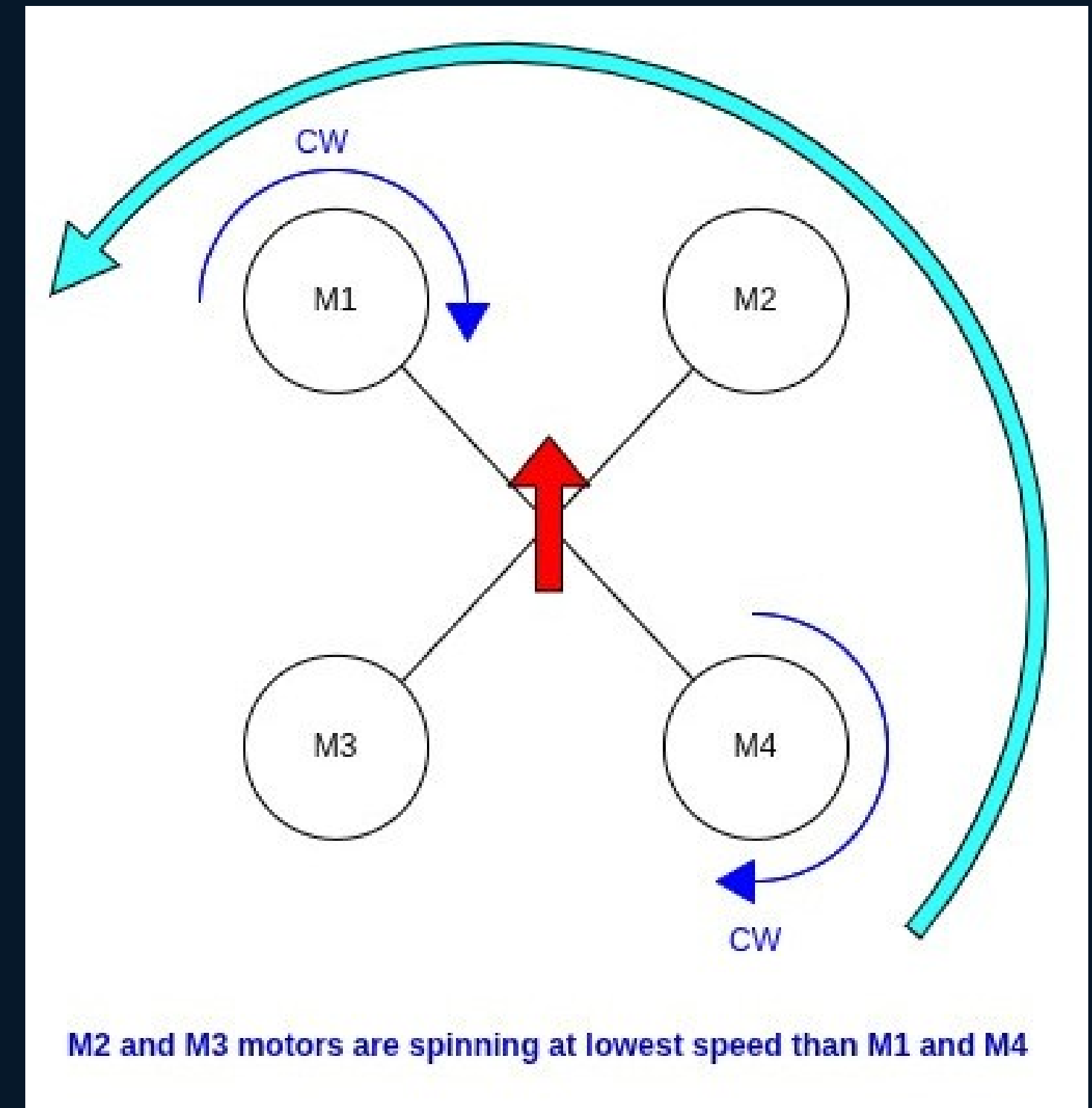
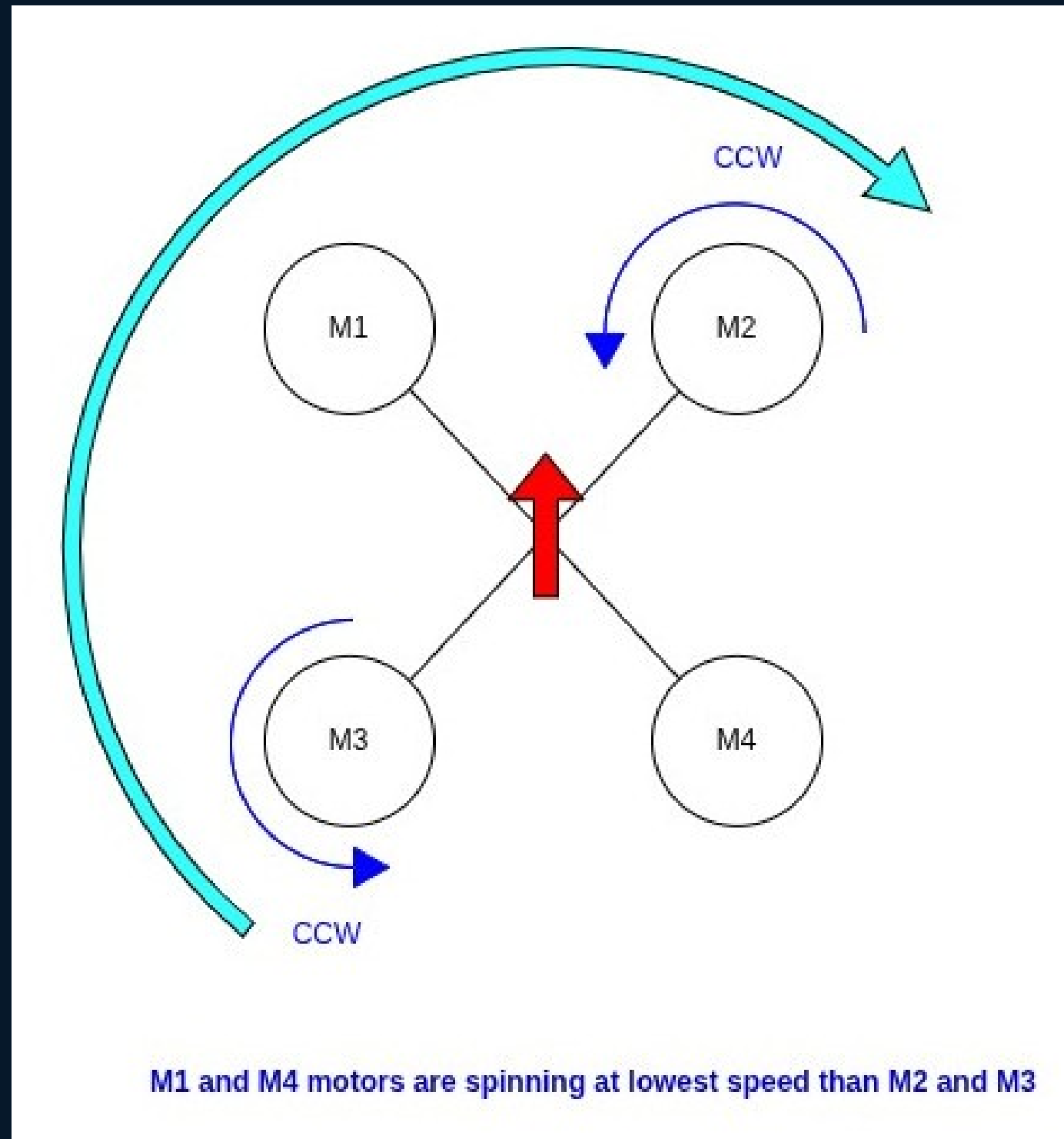
- Un quadricottero è una velivolo senza equipaggio, pilotato da terra o computer di bordo, dispone di 4 rotori
- Si muove sfruttando il concetto fisico di coppia e controcoppia (la coppia si riferisce alla forza rotazionale applicata da un'elica, mentre la controcoppia è la forza reattiva che tende a far ruotare il corpo del drone in direzione opposta)
- Si muove lungo tre assi (Pitch, Roll, Yawn)



Velocità dei motori e rotazione / traslazione lungo gli assi

- La velocità di ogni singolo motore è gestita dal controllore di volo (centralina) in base allo stato interno e ai valori in input (comandi + sensori)
- Pitch (rotazione lungo l'asse Y): aumentando la velocità dei motori posteriori, la coda si alza e il muso si abbassa, inclinando il drone in avanti (avanzamento), discorso analogo per muovere il drone all'indietro.
- Roll (rotazione lungo l'asse X): aumentando la velocità dei motori di sinistra, il lato sinistro si alza e il drone trasla verso destra, discorso analogo per la traslazione a sinistra.
- Yaw (rotazione lungo l'asse Z): per ruotare a destra, la centralina rallenta i motori che ruotano a destra (CW) e accelera quelli che ruotano a sinistra (CCW), caso analogo per la rotazione a sinistra... il rallentamento è importante: se la centralina si limitasse ad accelerare due motori, il drone non solo ruoterebbe su se stesso, ma schizzerebbe anche verso l'alto!

YAW e rotazione a destra / sinistra sull'asse Z



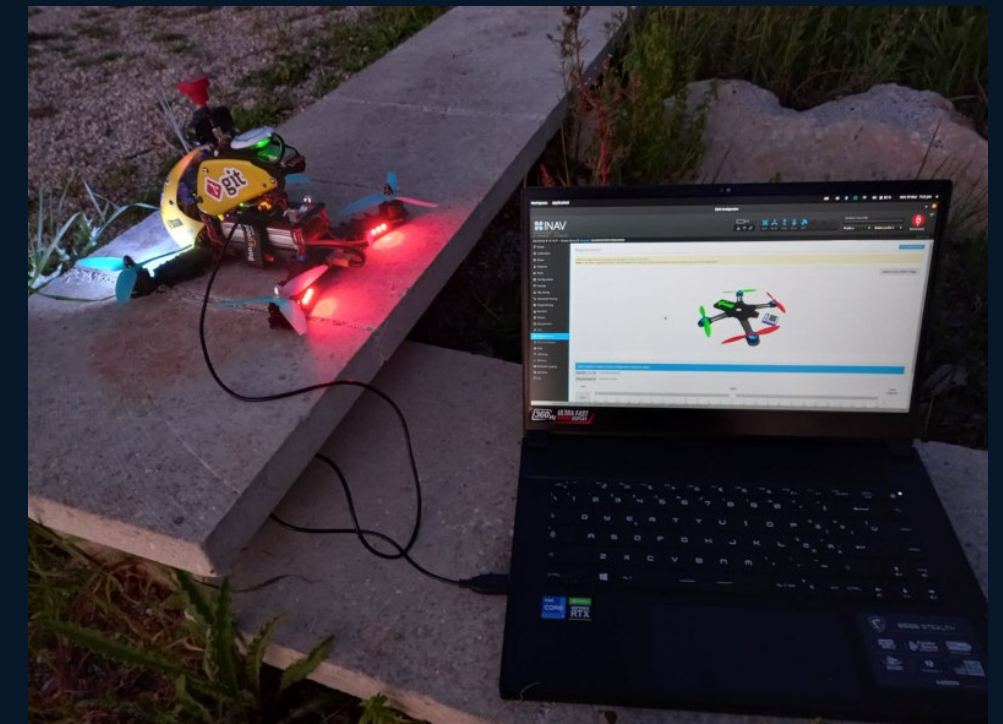
Droni da ripresa (Consumer - Ready to Fly)

- **Utilizzo:** riprese aeree professionali e amatoriali
- **Modelli noti:** DJI Mini, Air, Mavic, DJI Phantom, Autel Evo, Hubsan
- **Caratteristiche:** stabilizzazione su 3 assi, risoluzioni fino a 5K, modalità automatizzate e guida assistita (es. follow me, orbit)
- **Ideali per** fotografia, video, documentari, vlog



Droni FPV (First Person View)

- Utilizzo: racing e freestyle, esperienza immersiva con visore
- Modelli noti: DJI Avata, Cinelog, modelli SpeedyBee
- Caratteristiche: dotati di visori e radiocomando, alcuni modelli hanno un GPS limitato al solo recovery (nessuna guida assistita)
- Ideali per puro divertimento, competizioni sportive, riprese acrobatiche

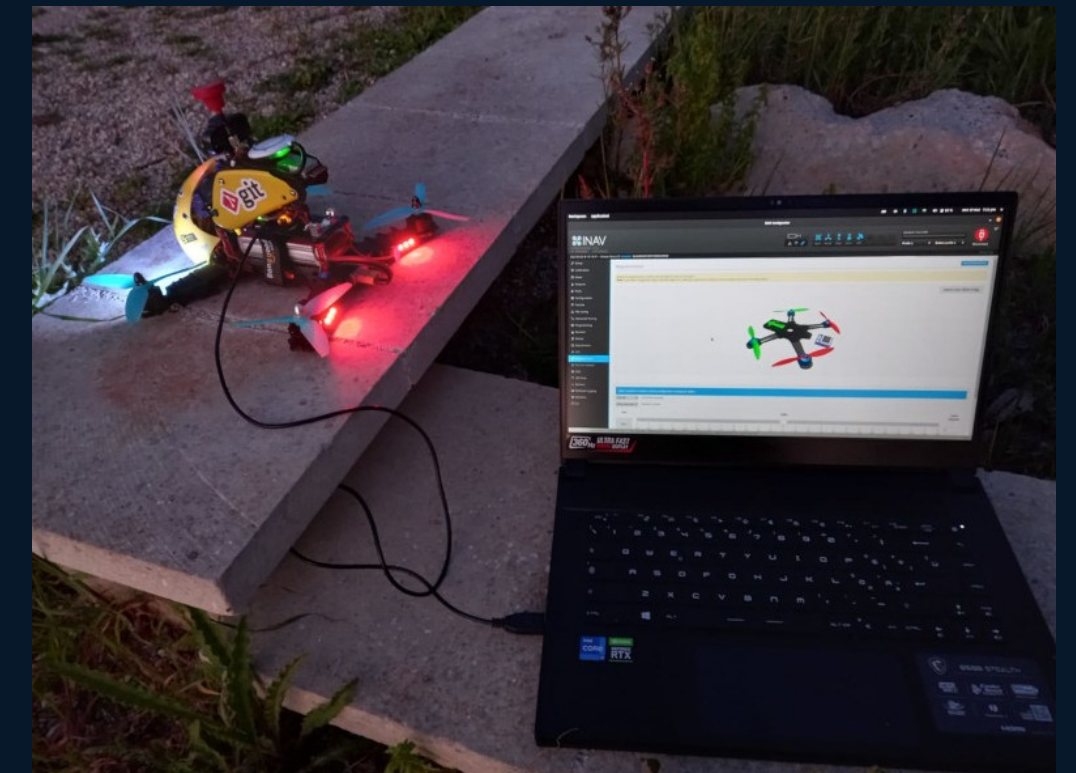


FPV Racing: uno sport ufficiale



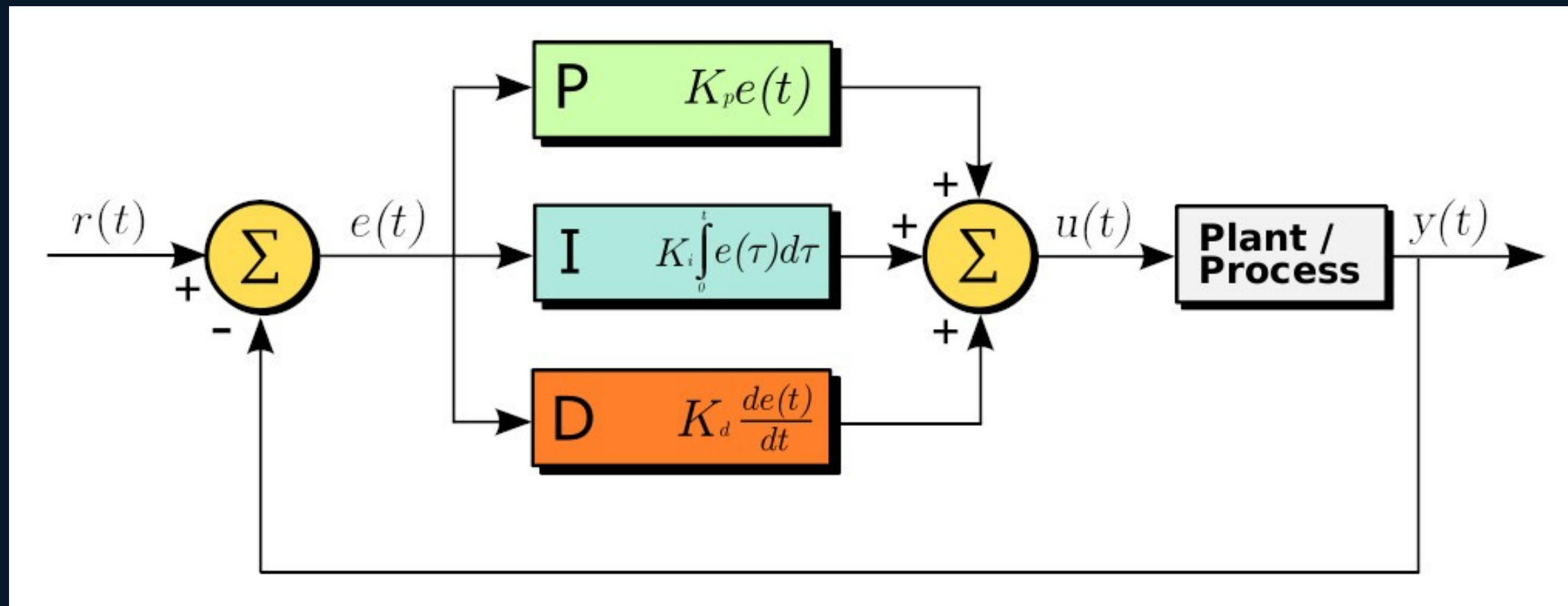
Hardware di un drone FPV

- Flight controller (su vengono eseguiti gli algoritmi di stabilizzazione)
- Sistema di propulsione (esc, motori, eliche)
- Sistema trasmissione video (analogico/digitale) + visore (goggles)
- Sistema di controllo (radiocomando + ricevente)
- Sensori interni (integrati nella centralina come IMU) ed esterni (barometro, bussola, GPS, etc)
- Rispetto ad un drone classico, un drone fpv ha il minimo necessario (confronto analogo: auto da rally VS auto GT)



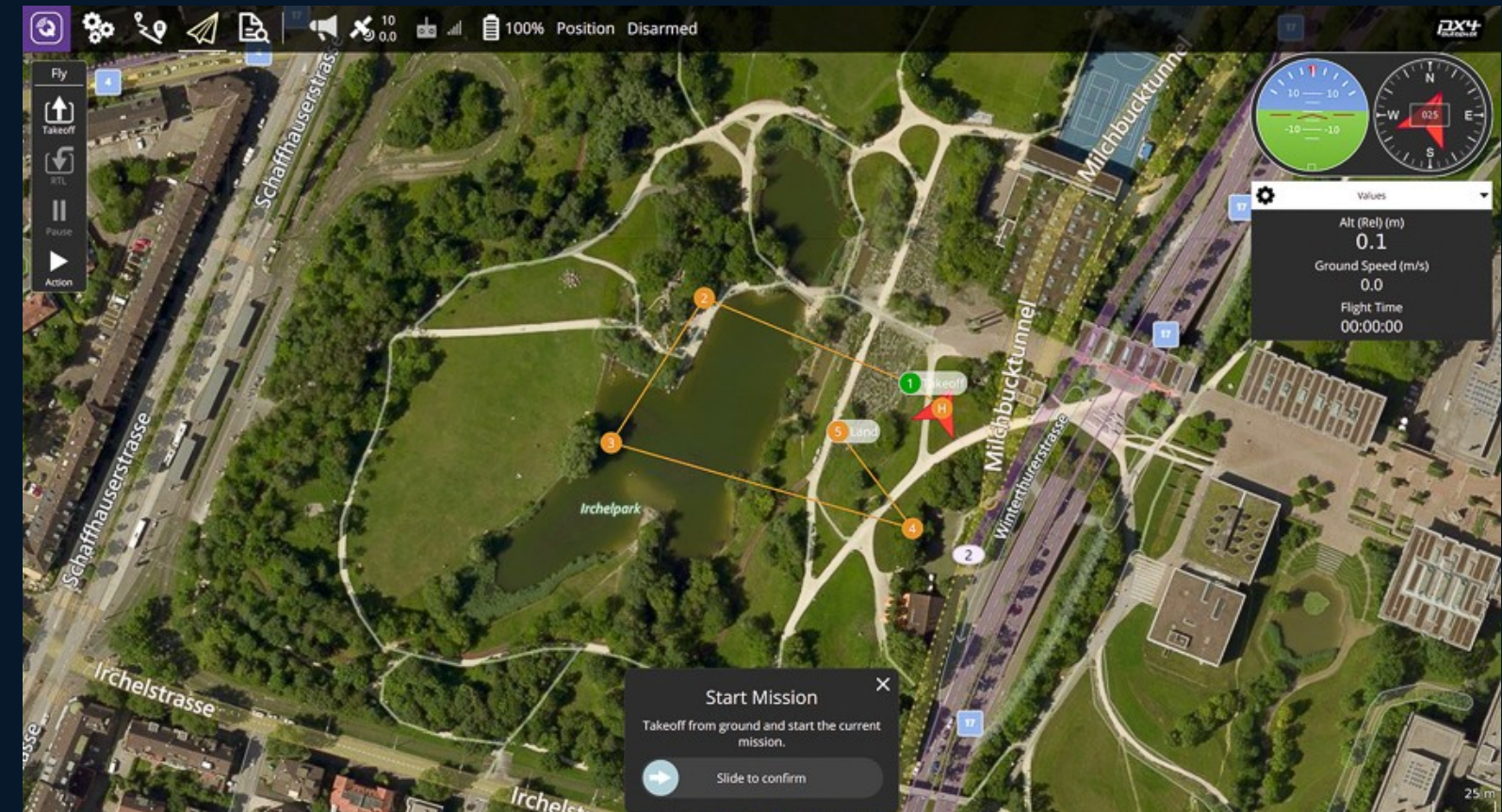
PID - Tuning

- Sono tre parametri che stabilizzano il velivolo: *proporzionale (P)*, *integrale (I)*, *derivata (D)*
- Permette al flight controller di “filtrare” gli input e di regolare l’asset anche in caso di sollecitazioni esterne (es. vento)
- Non sono uguali per tutti i velivoli, dipendono dal peso, dalla batteria e potenza dei motori
- Non è facile calcolarli, per settarli bisogna partire da valori pre-impostati e fare tuning ad ogni tentativo

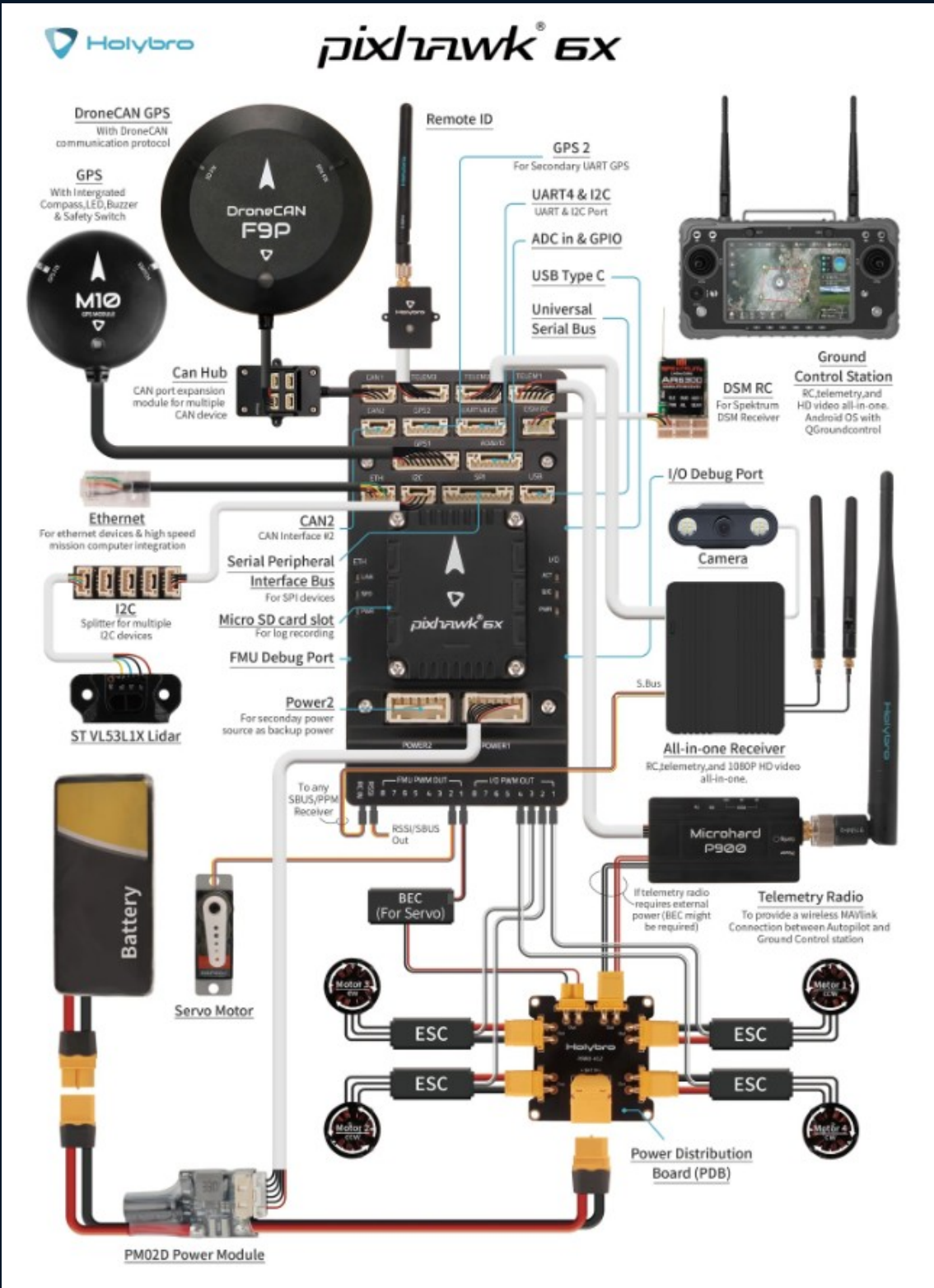
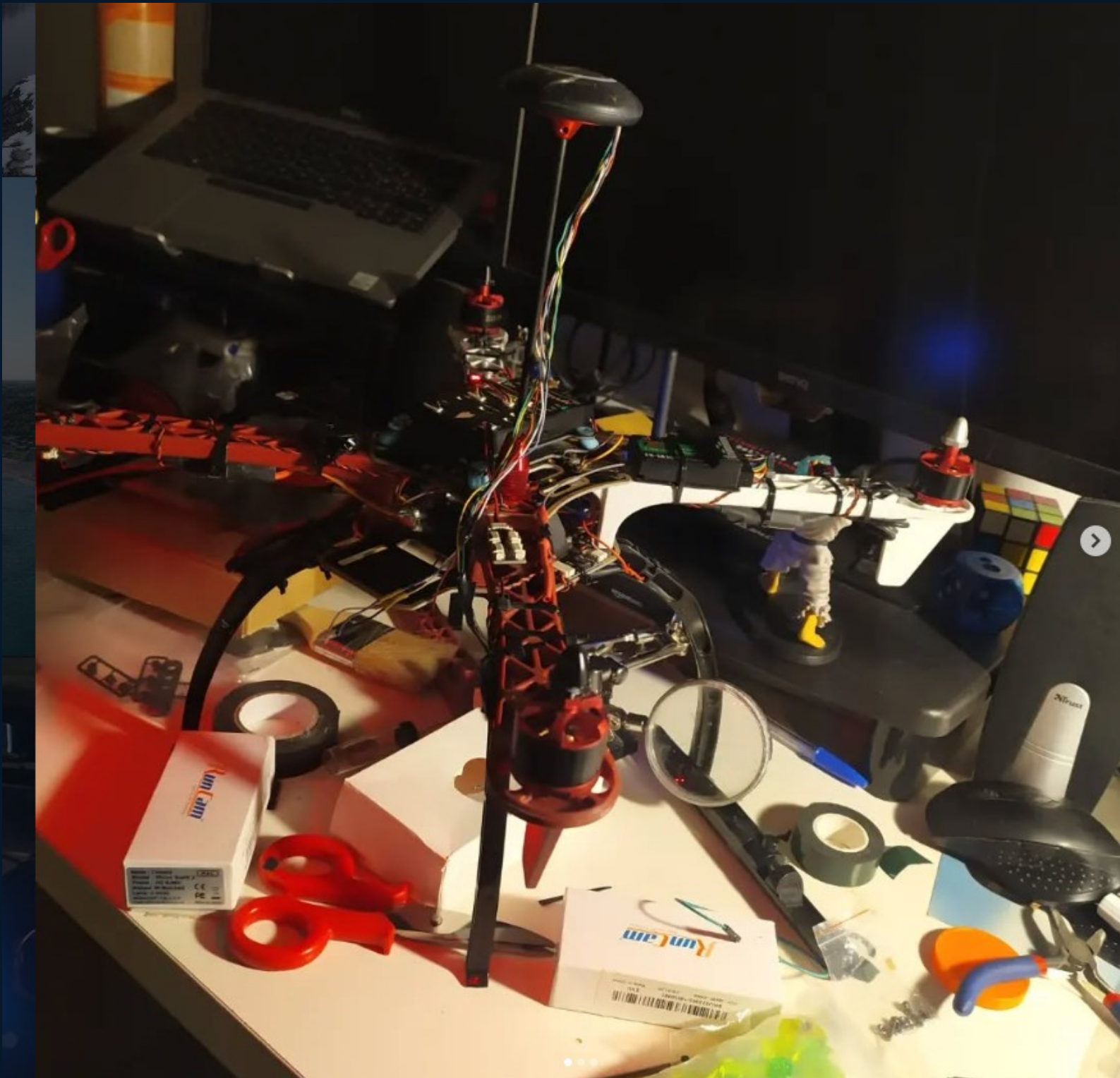


Droni autocostruiti / enterprise

- Utilizzo: riprese aeree professionali e ricerca
- Modelli noti: flight controller basati su PX4, Ardupilot, Inav, Betaflight
- Caratteristiche: sono dotati di stazioni da terra per pianificare missioni di volo, possono essere programmati sia da GUI che con appositi framework e linguaggi
- Ideali per ricerca, missioni, puro divertimento



Droni autocostruiti / enterprise

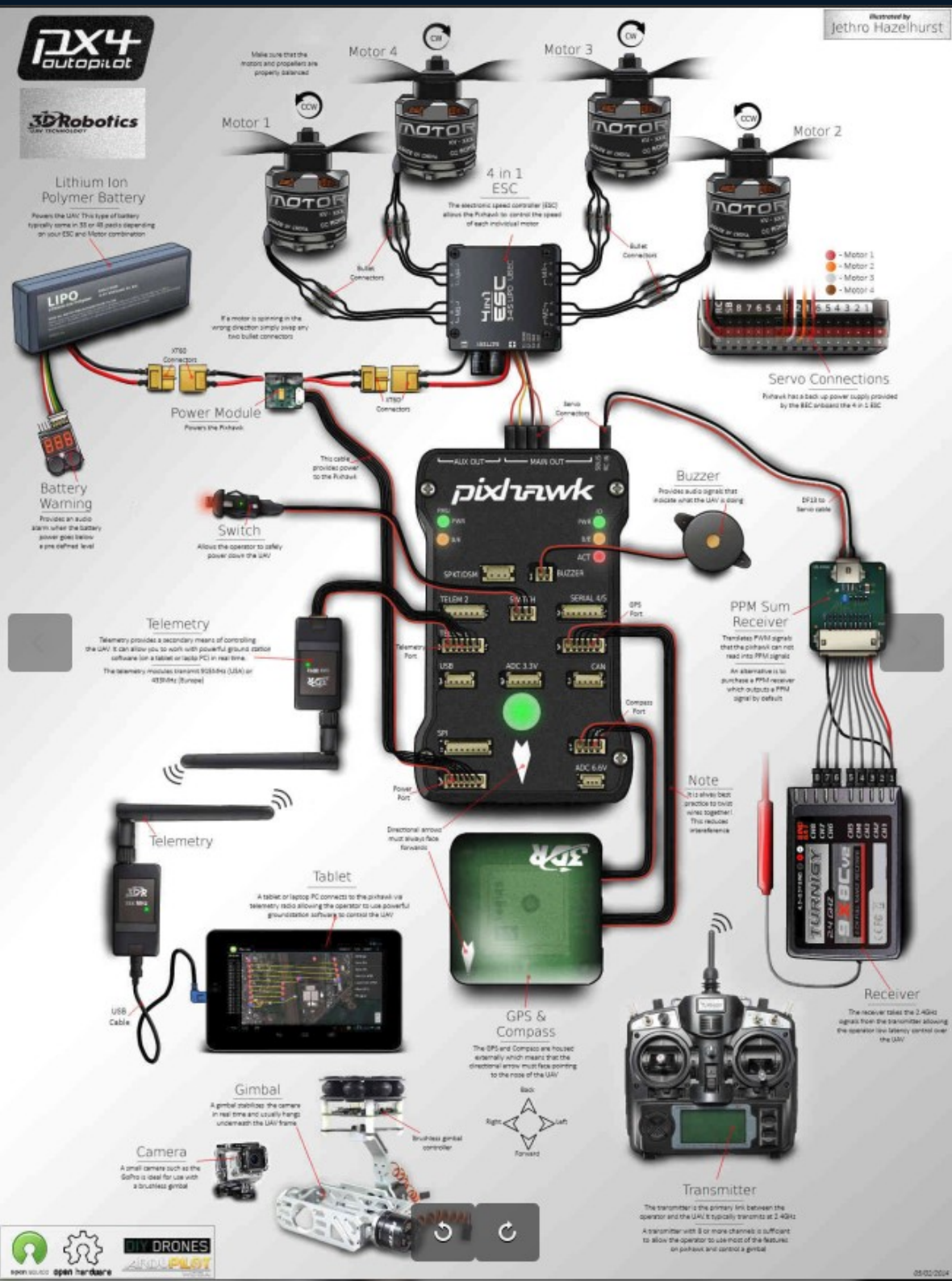
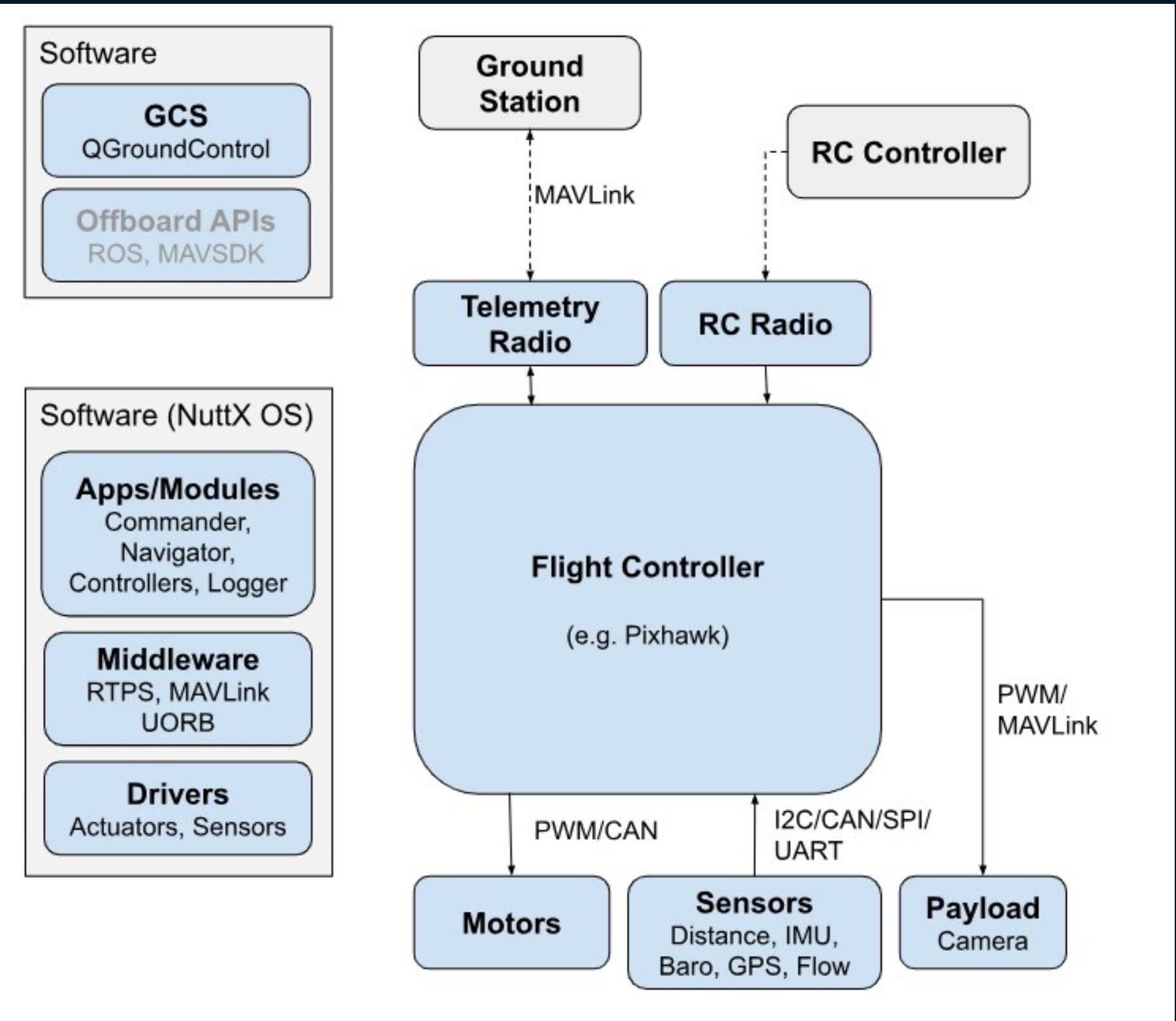


Software per Flight Controller e SDK

- Droni FPV: hanno firmware open come Betaflight, Inav e derivati, dispongono solo di interfaccia di configurazione, solitamente senza sdk
- Droni programmabili tramite sdk: hanno firmware open come PX4, Ardupilot o alcuni modelli Dji con sdk proprietario
- Dronecode Foundation: sottogruppo della Linux Foundation, fondazione indipendente per progetti open source sui droni, collabora con sviluppatori, utenti finali ed aziende per lo sviluppo libero.
- PX4 ed Ardupilot implementano Mavlink usato per la comunicazione tra Flight controller e GCS.
- PX4 è basato su Licenza BSD mentre Ardupilot su GPL3 (dal 2016 non fa parte della Dronecode Foundation)
- In questa sede approfondiremo PX4, tenendo conto che Ardupilot può essere un'ottima alternativa libera



Architettura PX4



Architettura PX4

PX4 è uno dei sistemi operativi per droni (autopiloti) più avanzati e diffusi al mondo. La sua forza risiede in un'architettura modulare e stratificata, progettata per essere robusta, flessibile e basata su

- 1. Message Broker uORB**
- 2. Hardware Abstraction Layer (HAL) con i controller Position, Velocity e Attitude**
- 3. Gestori del volo: Navigator e Commander**
- 4. MAVLink per la comunicazione esterna**

1. Il Middleware: uORB (Micro Object Request Broker)

- Mavlink è perfetto per scambiare informazioni su missioni e di telemetria con sistemi esterni al FC
- Internamente viene utilizzato uORB, un message broker ad altissime prestazioni che permette lo scambio di messaggi in modo efficiente da parte dei processi interni (aspetto cruciale in una centralina di volo!)
- Vengono utilizzati appositi topic per salvare dati grezzi provenienti dai sensori ed informazioni elaborate (a seconda del topic), ogni subscriber può registrarsi ad un topic per leggere il dato aggiornato
- uORB è un sistema "Last-is-Best": se un Publisher invia 10 aggiornamenti e il Subscriber ne legge solo uno, riceverà sempre e solo l'ultimo valore
- Per fare un esempio: viene letta sempre l'ultima posizione del GPS dal topic *vehicle_gps_position*.

Questo rende il sistema estremamente asincrono e isolato: se un modulo fallisce, non trascina necessariamente con sé l'intero sistema.

2. Hardware Abstraction Layer (HAL)

- Il Codice di volo legge dati standardizzati fornito dal driver, senza conoscere la natura del dispositivo
- Dietro le quinte vi è NuttX (sistema realtime) che garantisce determinismo e bassa latenza.
- Sono presenti degli estimators (come il Filtro Esteso di Kallman) che leggono i dati grezzi dai topic di IMU, GPS, Barometro e calcolano lo stato del drone, contenente informazioni di posizione, velocità e orientamento.

Calcolato tale stato, vengono coinvolti (ad ogni ciclo)

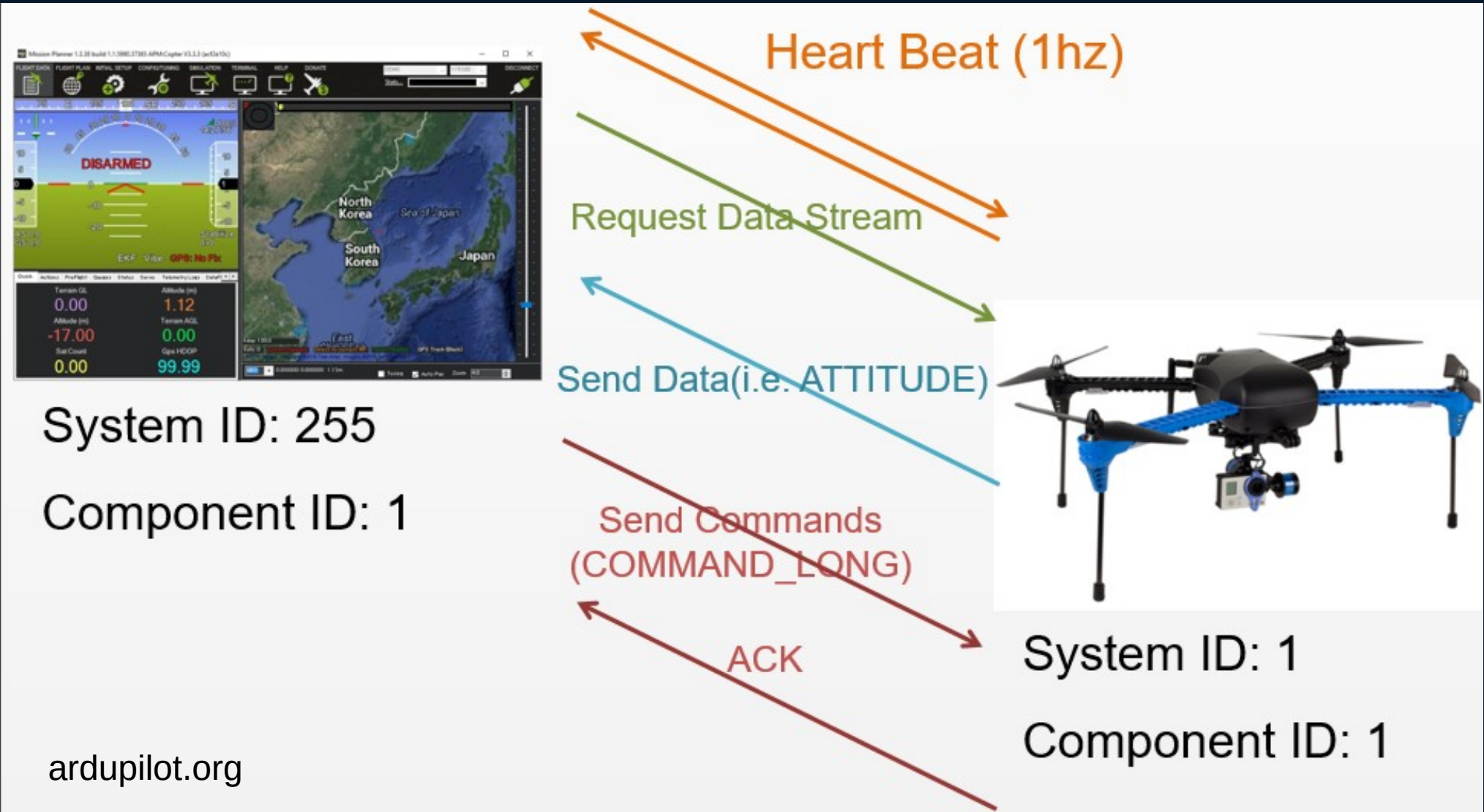
- Position Controller: per il calcolo della velocità necessaria per raggiungere una posizione.
- Velocity Controller: per il calcolo dell'assetto (pitch/roll) necessario.
- Attitude Controller: per il calcolo della velocità di rotazione dei motori (Rate Control).

3. Gestione del Volo: Navigator e Commander

Mentre i controller gestiscono la fisica del drone, I seguenti moduli gestiscono la logica:

- **Commander:** gestisce i cambi di stato (Arming/Disarming), i controlli di sicurezza (Failsafe) e i passaggi tra le modalità (Manual, Mission, Hold).
- **Navigator:** gestisce le missioni autonome (ricevute dalla GCS), i waypoint e i comandi di ritorno a casa (RTL)

4. Mavlink: un protocollo per lo scambio dati



MAVLink Frame – 8-263 bytes

ardupilot.org

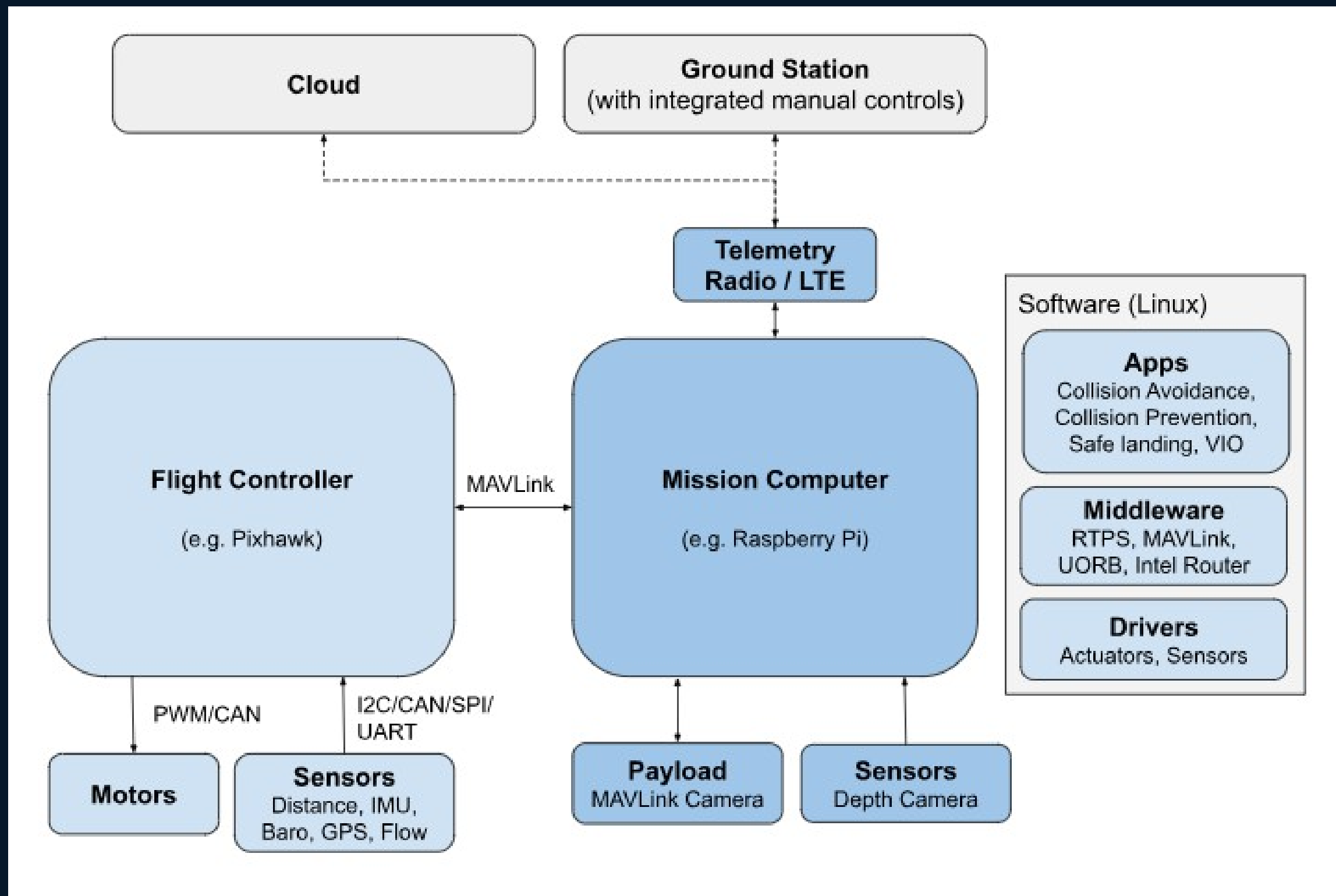
Byte Index	Content	Value	Explanation
0	Packet start sign	v1.0: 0xFE (v0.9: 0x55)	Indicates the start of a new packet.
1	Payload length	0 - 255	Indicates length of the following payload.
2	Packet sequence	0 - 255	Each component counts up his send sequence. Allows to detect packet loss
3	System ID	1 - 255	ID of the SENDING system. Allows to differentiate different MAVs on the same network.
4	Component ID	0 - 255	ID of the SENDING component. Allows to differentiate different components of the same system, e.g. the IMU and the autopilot.
5	Message ID	0 - 255	ID of the message - the id defines what the payload "means" and how it should be correctly decoded.
6 to (n+6)	Data	(0 - 255) bytes	Data of the message, depends on the message id.
(n+7) to (n+8)	Checksum (low byte, high byte)	ITU X.25/SAE AS-4 hash, excluding packet start sign, so bytes 1..(n+6) Note: The checksum also includes MAVLINK_CRC_EXTRA (Number computed from message fields. Protects the packet from decoding a different version of the same packet but with different variables).	

Mavlink: un protocollo per lo scambio dati

- È un protocollo applicativo, collocabile a livello 7 nel modello ISO/OSI, permettendo lo scambio di informazioni tra la Ground Control Station (GCS) ed il flight controller
- Può appoggiarsi su protocolli come UDP oppure sfruttare un collegamento seriale UART (cablato o radio)
- MavSDK è implementato in diversi linguaggi: ciò permette di implementare applicativi sia lato edge (script in esecuzione su un secondo computer di bordo) che applicazioni desktop, mobile o cloud per la gestione del drone

- MAVSDK-C++ (2016): Used in production.
- MAVSDK-Swift (2018): Used in production.
- MAVSDK-Python (2019): Used in production.
- MAVSDK-Java (2019): Used in production.
- MAVSDK-Go (2020): Proof of concept.
- MAVSDK-JavaScript (2019): Proof of concept.
- MAVSDK-CSharp (2019). Proof of concept.
- MAVSDK-Rust (2019): Proof of concept.

Mission Computer e Flight Controller



Mission Computer e Flight Controller

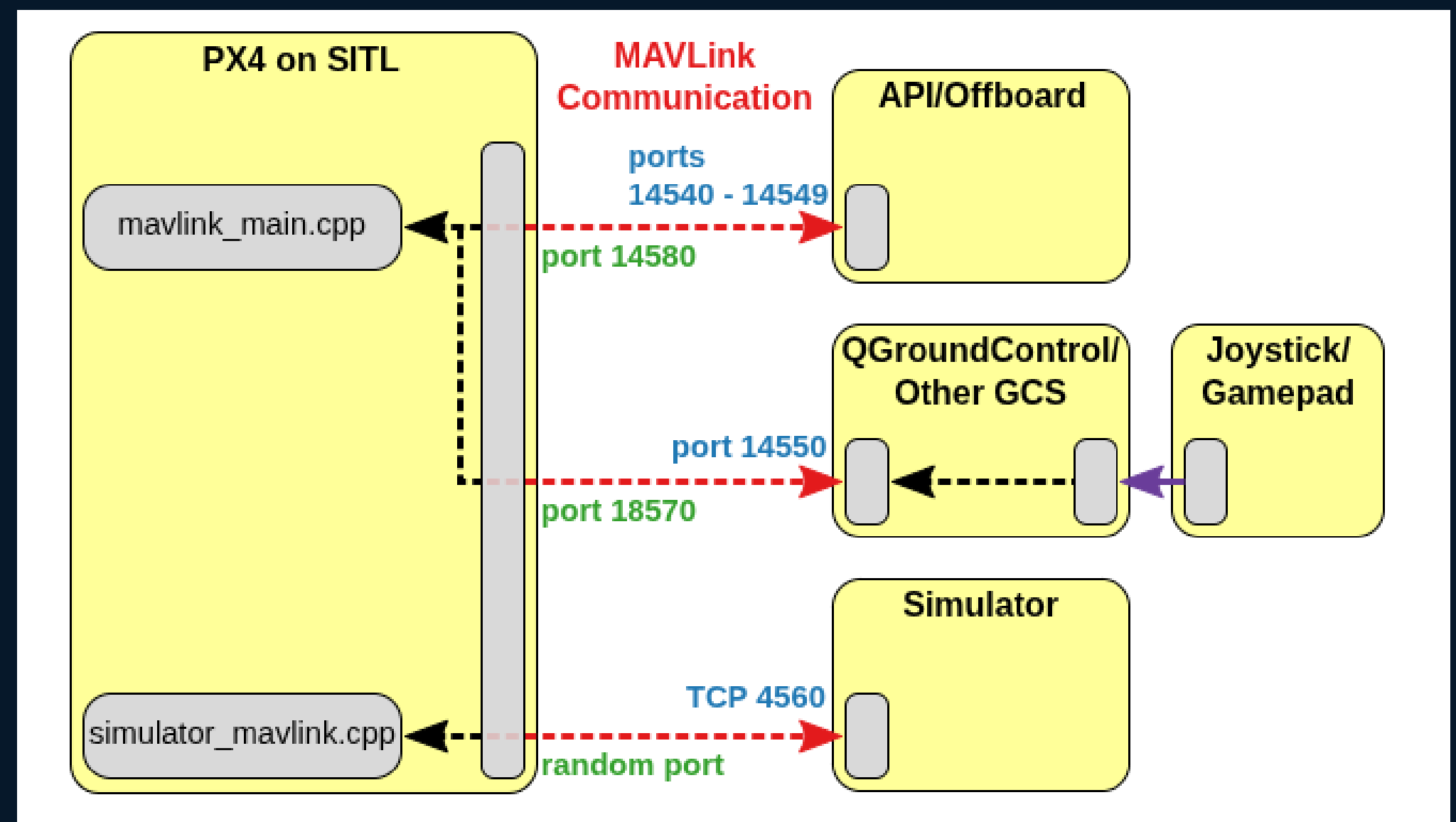
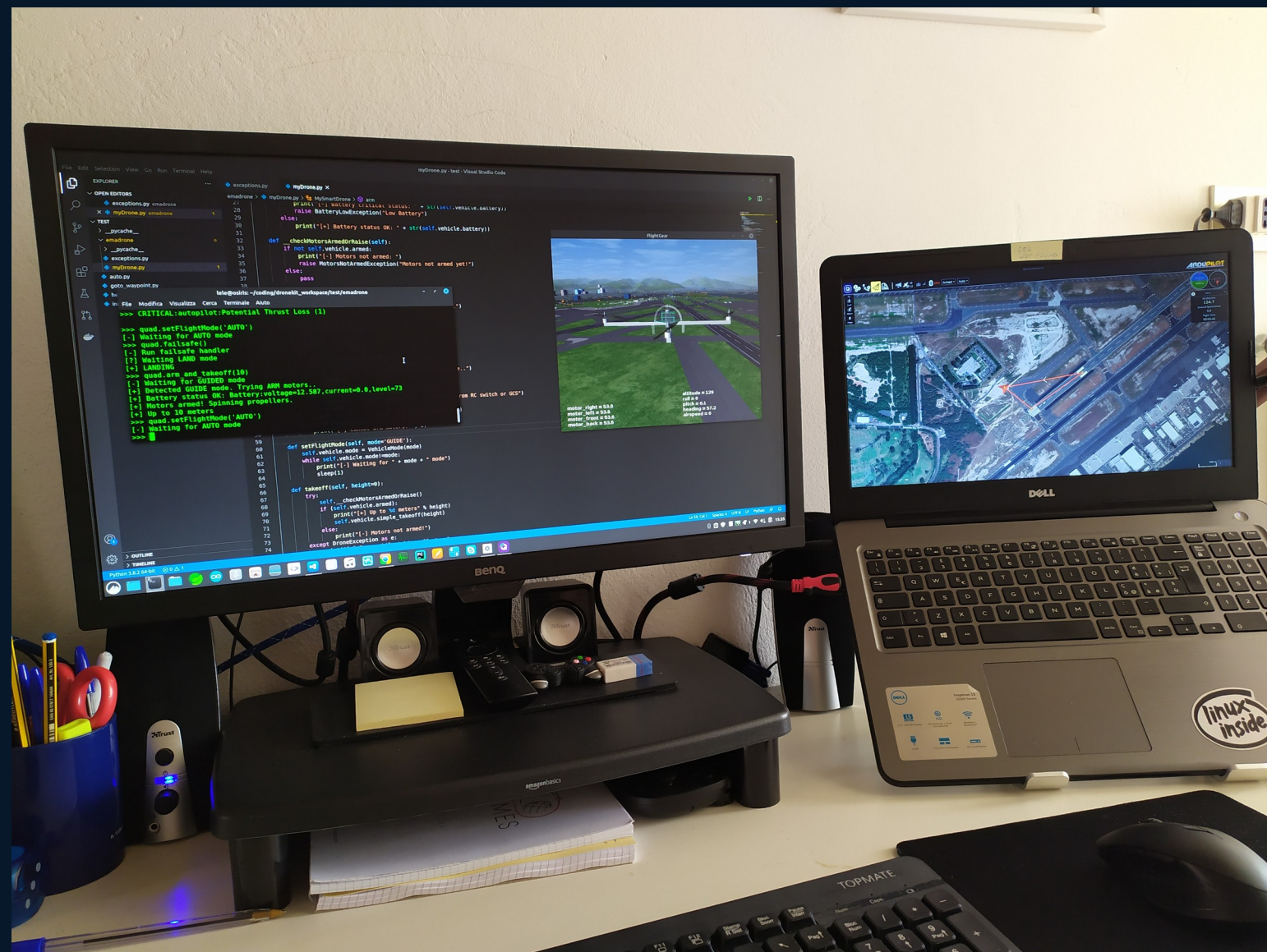
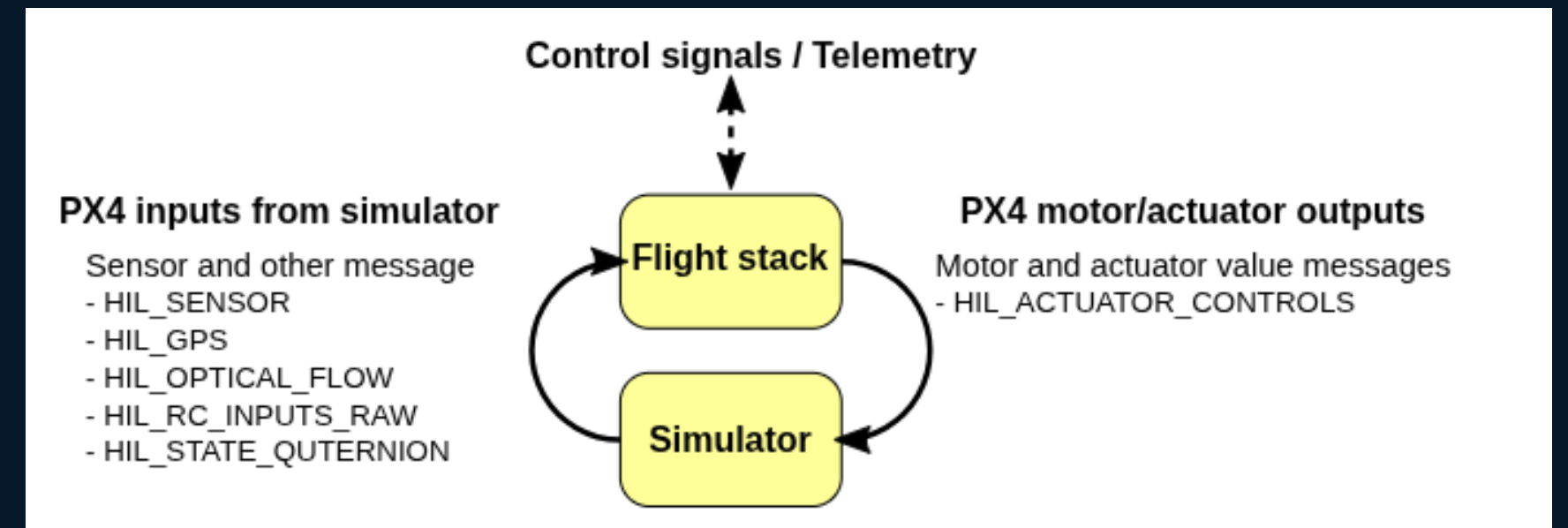
- *Flight Controller* è la centralina su cui gira PX4, può essere definito il sistema nervoso del drone, in quanto gestisce le operazioni di basso livello come il movimento dei motori, il ritorno a casa automatico, etc
- *Mission Computer (o Companion Computer)* è il cervello pensante del drone, su cui vengono pianificate le missioni

Per fare degli esempi concreti:

- Flight Controller può una centralina Pixhawk a cui sono collegati ESC, ricevente radio, GPS, etc
- Mission Computer può essere un Raspberry Pi a bordo del drone, con con modulo *NPU* e uno script di *Computer Vision* per motion tracking di un oggetto in movimento
- NB: un Mission Computer può anche essere un device remoto (esempio uno smartphone o tablet con software per *Ground Station*)

Simulazione e demo

- SITL: firmware e simulatore emulati sul computer
- HITL: firmware sulla centralina, simulatore sul computer
- Sensori e attuatori “virtualizzati” in entrambi i casi
- Noi utilizzeremo SITL



DISCLAIMER: Occhio a quello che fate!

A scopo didattico, ci si limita ad eseguire alcuni script in un ambiente virtuale e sicuro. Con alcuni settaggi, tali script funzionano anche su HARDWARE REALE, ma con tutta una serie di RISCHI e PERICOLI REALI (tra cui la perdita del controllo del drone, con esiti imprevedibili).

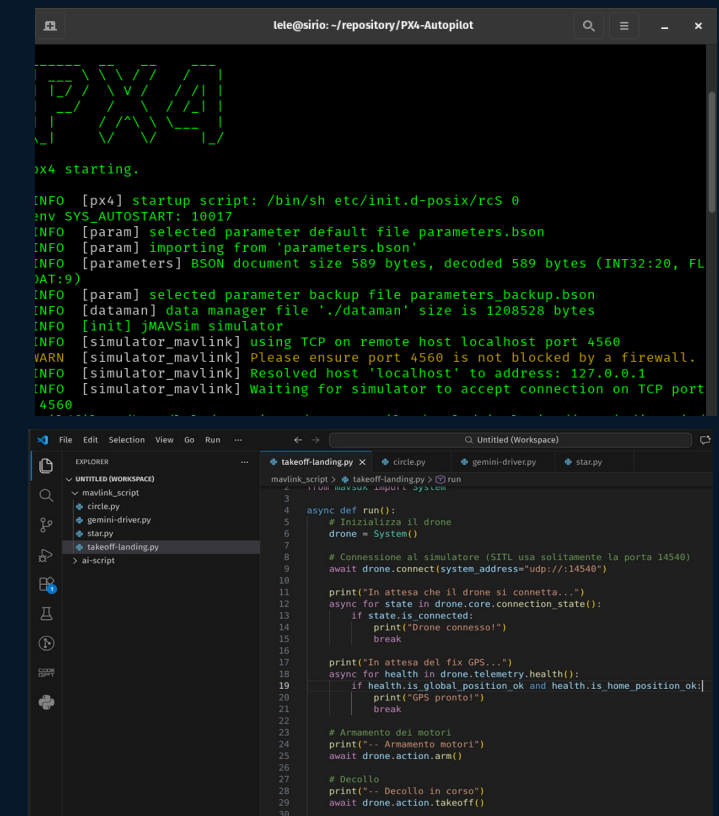
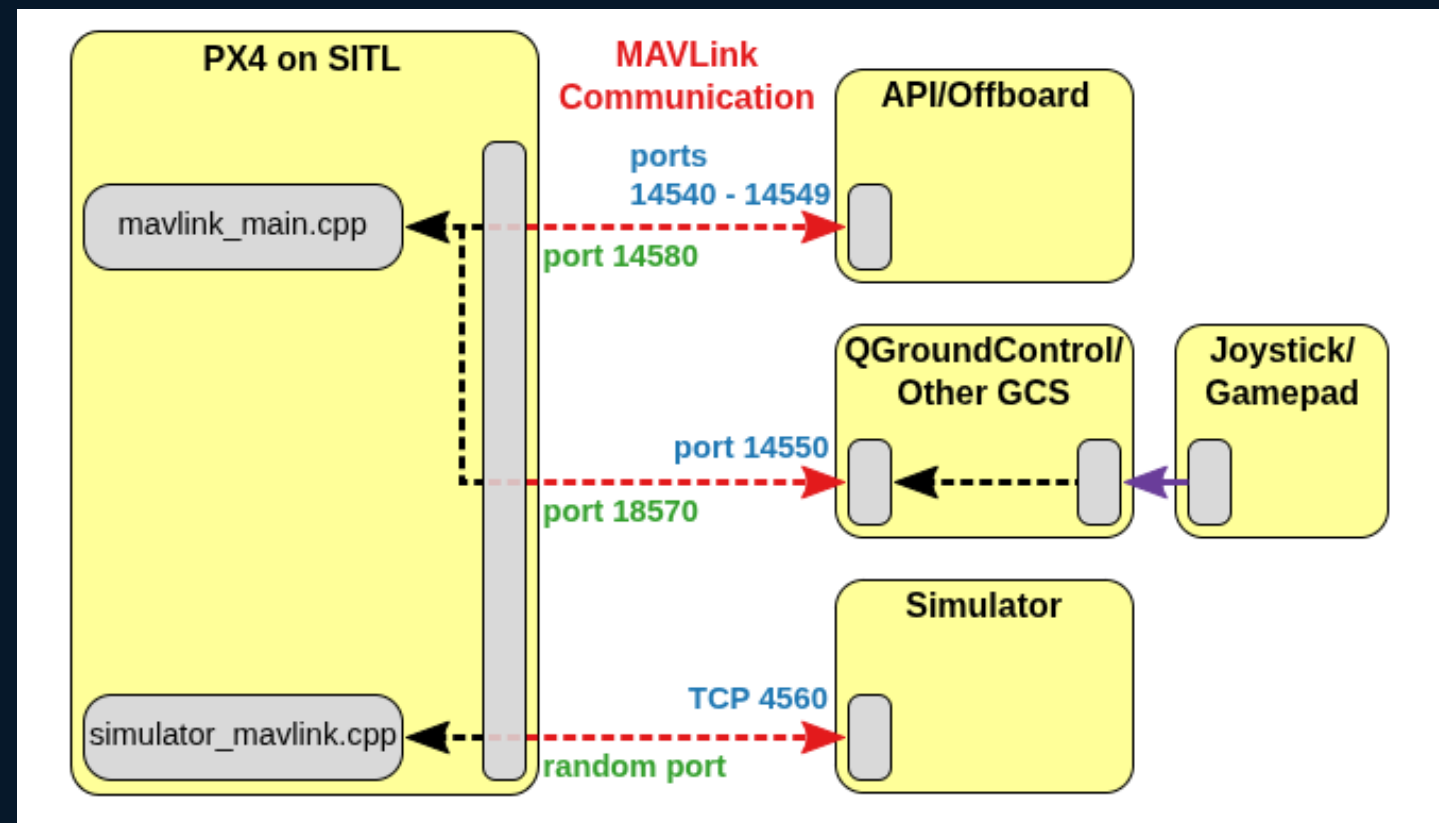
PRESTATE ATTENZIONE AI REGOLAMENTI: IN ALCUNI PAESI IL VOLO AUTONOMO È VIETATO O PERMESSO SOLO TRAMITE AUTORIZZAZIONI RILASCIATE DALLE AUTORITÀ COMPETENTI.

L'AUTORE NON SI RITIENE RESPONSABILE DI EVENTUALI ABUSI DOVUTI ALLE INFORMAZIONI PRESENTI IN QUESTA SEZIONE.

Simulazione e demo (SITL)

Per la nostra demo utilizzeremo la modalità SITL e la nostra macchina locale:

- **PX4 SITL è un'istanza di PX4 virtualizzata con il comando “*make px4_sitl jmavsim*”**
- **Come GCS abbiamo utilizzato QgroundControl per riprendere il controllo del drone e gestire le missioni**
- **JmavSim come *simulatore* (lanciato con il comando *make* di px4)**
- **Come *Offboard* usiamo la nostra macchina, come *API* abbiamo la libreria *mavsdk-python***
- **Un IDE come *VSCode* e un terminale per lanciare gli script python**



Una missione “Hello Drone”: decollo ed atterraggio

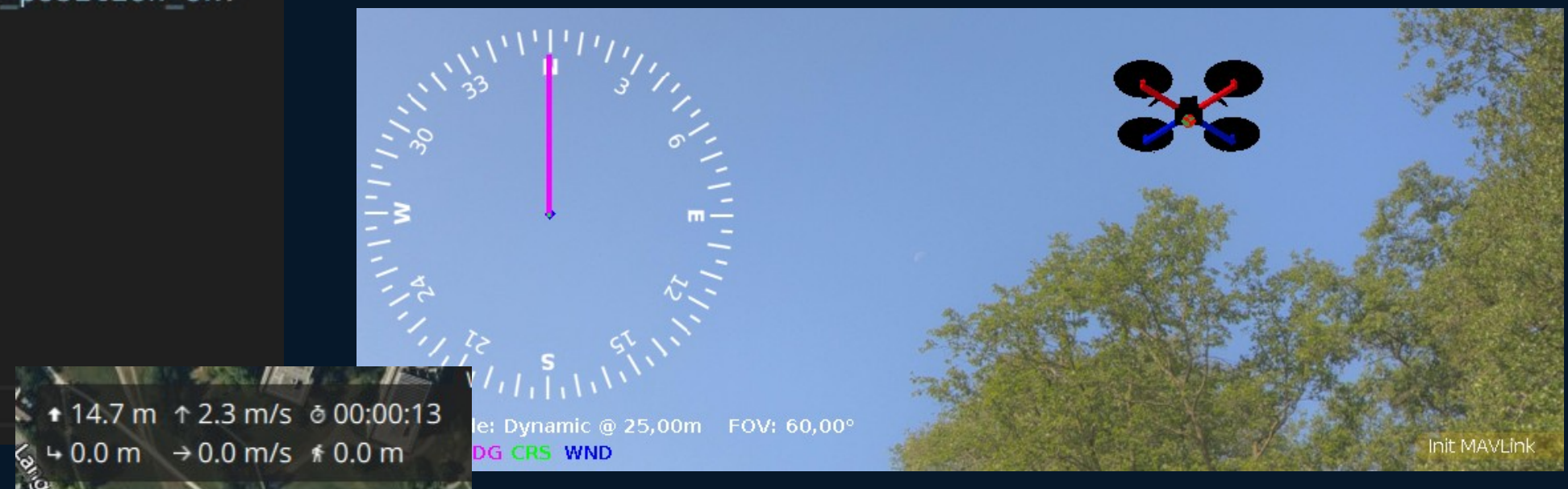
Per far decollare il drone, il nostro script dovrà:

- 1. Connettersi all’istanza di PX4 (inizializzazione)**
- 2. Attendere il segnale GPS ed il numero minimo di satelliti (gps fix)**
- 3. Attivare i motori (arm)**
- 4. Decollare ad altezza pre-impostata (takeoff)**
- 5. Attendere 20 secondi**
- 6. Atterrare (landing)**
- 7. Disattivare i motori (disarm)**

Missione 1: inizializzazione, avvio motori e decollo

```
1 import asyncio
2 from mavsdk import System
3
4 async def run():
5     # Initializing drone object
6     drone = System()
7
8     # Connecting to PX4 instance (sitl uses port UDP 14540 by default)
9     await drone.connect(system_address="udp://:14540")
10
11     print("Waiting for drone connection ready...")
12     async for state in drone.core.connection_state():
13         if state.is_connected:
14             print("Drone connected!")
15             break
16
17     print("Waiting for GPS fix and satellites..")
18     async for health in drone.telemetry.health():
19         if health.is_global_position_ok and health.is_home_position_ok:
20             print("GPS ready!")
21             break
22
23     # Arming motors
24     print("-- Arming motors")
25     await drone.action.arm()
26
27     # Taking off the drones
28     print("-- Taking off")
29     await drone.action.takeoff()
```

```
tele@sirio: ~/repository/PX4-Autopilot
tele@sirio: ~/repository/PX4-Autopilot x tele@sirio: ~/repository/mavlink_script x
INFO [mavlink] mode: Onboard, data rate: 4000 B/s on udp port 14280 remote port 14030
INFO [mavlink] partner IP: 127.0.0.1
INFO [mavlink] mode: Gimbal, data rate: 400000 B/s on udp port 13030 remote port 13280
INFO [logger] logger started (mode=all)
INFO [logger] Start file log (type: full)
INFO [logger] [logger] ./log/2026-01-04/19_46_40.ulg
INFO [logger] Opened full log file: ./log/2026-01-04/19_46_40.ulg
INFO [mavlink] MAVLink only on localhost (set param MAV_{i}_BROADCAST = 1 to enable network)
INFO [mavlink] MAVLink only on localhost (set param MAV_{i}_BROADCAST = 1 to enable network)
INFO [px4] Startup script returned successfully
pxh> WARN [health_and_arming_checks] Preflight: GPS fix too low
INFO [tone_alarm] home set
WARN [health_and_arming_checks] Preflight: GPS fix too low
INFO [commander] Ready for takeoff!
INFO [commander] Armed by external command
INFO [tone_alarm] arming warning
INFO [navigator] Using default takeoff altitude: 20.0 m
INFO [commander] Takeoff detected
```



Missione 1: atterraggio e disattivazione motori

```
31     # Wait for 20 seconds
32     await asyncio.sleep(20)
33
34     # Landing
35     print("-- Landing")
36     await drone.action.land()
37
38     # Monitor status until the drone is disarmed (on the ground)
39     async for is_armed in drone.telemetry.armed():
40         if not is_armed:
41             print("-- Drone landed and disarmed.")
42             break
```

...prima di procedere con missioni più avanzate, introduciamo una modalità di PX4..

Modalità OFFBOARD

1. È la modalità di PX4 con cui il controllo viene delegato allo script in esecuzione sul Mission Computer;
2. in questo modo passa dallo stato di “autonomia” (Position o Altitude) a quello di “esecuzione” (Offboard);
3. PX4 si limita a regolare i motori per raggiungere l’ultima posizione geografica ricevuta come Setpoint del tipo (X,Y, Z)
4. Lo script entra in questa modalità con un metodo apposito di mavsdk e deve inviare segnali costanti entro un intervallo di tempo stabilito (meno di un secondo)
5. In caso di timeout o instabilità, PX4 riprende il controllo del drone tramite Failsafe
6. Failsafe: il drone rimane sospeso in aria (Hold) o, in base alla carica della batteria, avvia la fase di atterraggio (Landing)

Modalità OFFBOARD

Per entrare in tale modalità, possiamo alzarci a 5 metri (altitudine di crociera nel simulatore) e richiamare il metodo *start()* sull'oggetto *Offboard*. In caso di errore, possiamo (ad esempio) avviare la fase di atterraggio in sicurezza.

```
await drone.offboard.set_position_ned(PositionNedYaw(0.0, 0.0, -5.0, 0.0))

try:
    await drone.offboard.start()
except OffboardError as error:
    print(f"Errore Offboard: {error._result.result}")
    await drone.action.land()
    return
```

Missione2: disegnare una stella con rotazione prua

Per disegnare una stella in cielo, il nostro script dovrà:

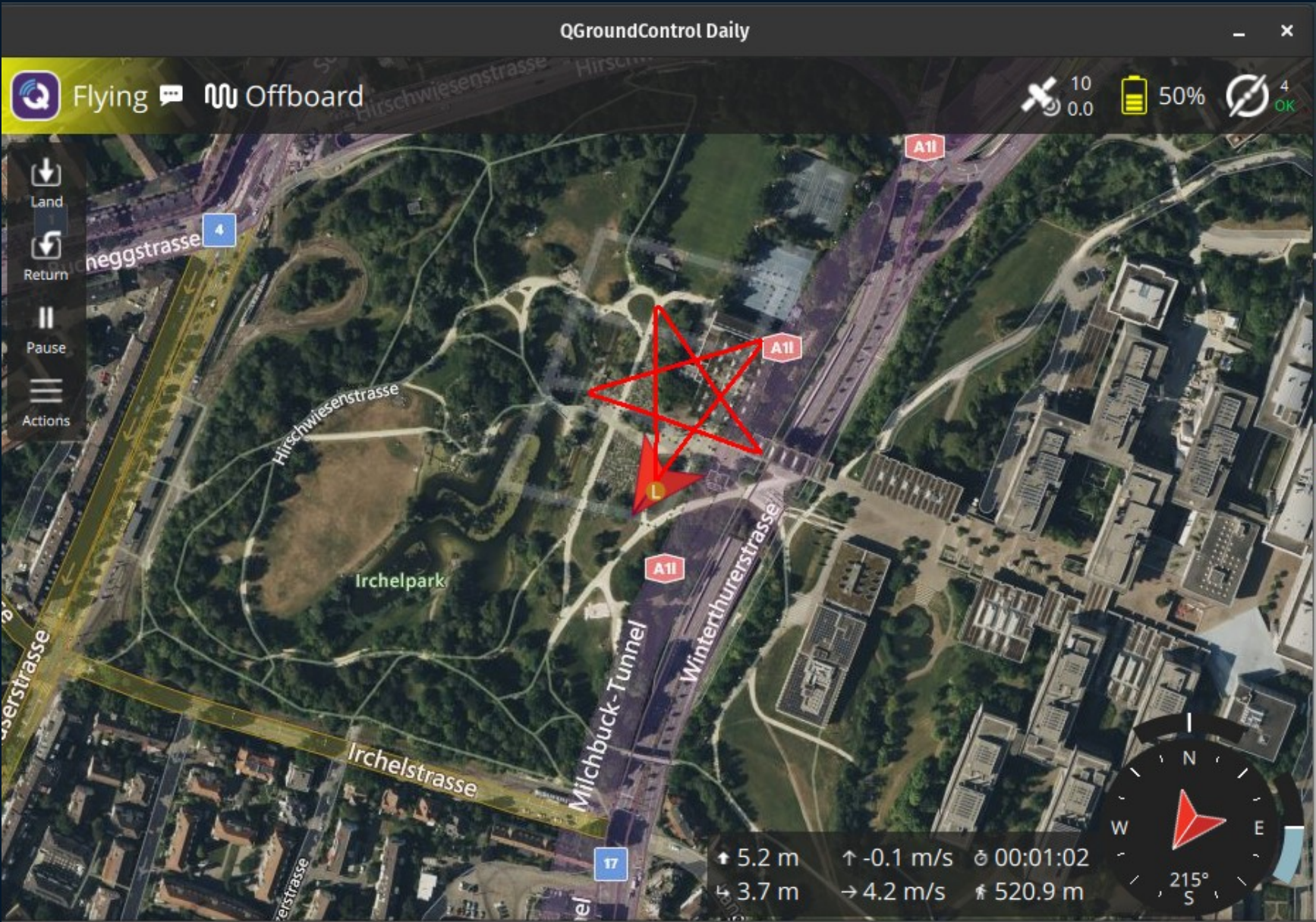
- 1. Connettersi all'istanza di PX4 (inizializzazione)**
- 2. Attendere il segnale GPS ed il numero minimo di satelliti (gps fix)**
- 3. Attivare i motori (arm)**
- 4. Decollare ad altezza pre-impostata (takeoff)**
- 5. Entrare in modalità offboard**
- 6. Determinare i 5 vertici del grafo, e in un ciclo, percorri gli archi che li connette, ruotando la prua ad ogni iterazione (uso di funzioni trigonometriche)**
- 7. Ritornate al punto di partenza**
- 8. Uscire dalla modalità offboard**
- 9. Atterrare (landing)**
- 10. Disattivare i motori (disarm)**

Disegnare una stella con rotazione prua

```
1 import asyncio
2 import math
3 from mavsdk import System
4 from mavsdk.offboard import OffboardError, PositionNedYaw
5
6 async def run():
7     drone = System()
8     await drone.connect(system_address="udp://:14540")
9
10    print("Waiting for drone...")
11    async for state in drone.core.connection_state():
12        if state.is_connected:
13            break
14
15    # 1. Arming and Takeoff
16    print("-- Arming and Takeoff")
17    await drone.action.arm()
18    await drone.action.takeoff()
19    await asyncio.sleep(8)
20
21    # 2. Initial setpoint to enter Offboard mode
22    # (Cruise altitude: 5 meters for safety with long segments)
23
24    await drone.offboard.set_position_ned(PositionNedYaw(0.0, 0.0, -5.0, 0.0))
25
26    try:
27        await drone.offboard.start()
28    except OffboardError as error:
29        print(f"Offboard Error: {error._result.result}")
30        await drone.action.land()
31        return
32
33    # 3. Move 5m forward with rotation towards North (0°)
34    print("-- Moving 5m forward (Looking ahead)")
35    await drone.offboard.set_position_ned(PositionNedYaw(5.0, 0.0, -5.0, 0.0))
36    await asyncio.sleep(6)
37
38    # 4. Drawing the Star with Yaw calculation
39    side = 100.0
40    star_angle = 0
41    curr_n, curr_e = 5.0, 0.0 # Current position after the first step
42
```

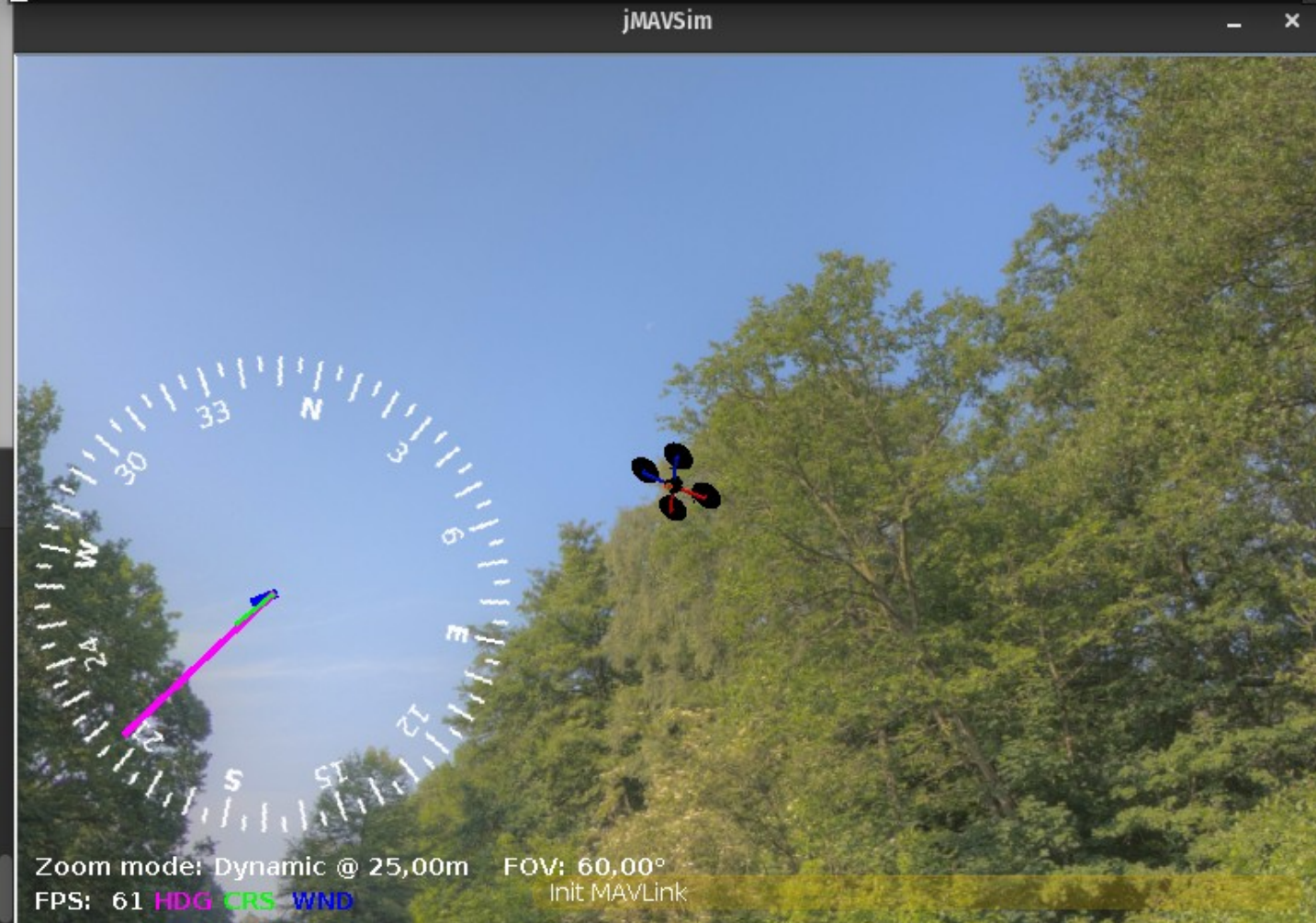
```
43    print("-- Starting star pattern with heading rotation")
44    for i in range(5):
45        # Calculate next position
46        rad = math.radians(star_angle)
47        next_n = curr_n + side * math.cos(rad)
48        next_e = curr_e + side * math.sin(rad)
49
50        # Calculate Yaw to look at the target point
51        # atan2(y, x) -> atan2(delta_east, delta_north)
52        yaw_deg = math.degrees(math.atan2(next_e - curr_e, next_n - curr_n))
53
54        print(f"Segment {i+1}: Towards N:{next_n:.1f} E:{next_e:.1f} | Yaw: {yaw_deg:.1f}°")
55
56        # Send movement and rotation command
57        await drone.offboard.set_position_ned(PositionNedYaw(next_n, next_e, -5.0, yaw_deg))
58
59        # Wait to reach the point (estimated time for 100m)
60        await asyncio.sleep(10)
61
62        # Update current position for the next calculation
63        curr_n, curr_e = next_n, next_e
64        star_angle += 144
65
66    # 5. Return to starting point (0,0) looking towards "home"
67    print("-- Returning to base")
68    yaw_return = math.degrees(math.atan2(0 - curr_e, 0 - curr_n))
69    await drone.offboard.set_position_ned(PositionNedYaw(0.0, 0.0, -5.0, yaw_return))
70    await asyncio.sleep(10)
71
72    # 6. Landing
73    print("-- Landing")
74    await drone.offboard.stop()
75    await drone.action.land()
76
77 if __name__ == "__main__":
78     asyncio.run(run())
```

Disegnare una stella con rotazione prua



```
tele@sirio: ~/repository/mavlink_script
python3 star.py
Connection using udp:// is deprecated, please use udpin:// or udpout:// (cli_arg.cpp:28)
Vehicle type changed (new type: 2, old type: 0) (system_impl.cpp:323)
Waiting for drone...
-- Arming and Takeoff
-- Moving 5m forward (Looking ahead)
Received ack for not-existing command: 176! Ignoring... (mavlink_command_sender.cpp:304)
-- Starting star pattern with heading rotation
Segment 1: Towards N:105.0 E:0.0 | Yaw: 0.0°
Segment 2: Towards N:24.1 E:58.8 | Yaw: 144.0°
Segment 3: Towards N:55.0 E:-36.3 | Yaw: -72.0°
Segment 4: Towards N:85.9 E:58.8 | Yaw: 72.0°
Segment 5: Towards N:5.0 E:0.0 | Yaw: -144.0°
-- Returning to base
```

```
tele@sirio: ~/repository/PX4-Autopilot
INFO [commander] Disarmed by landing
INFO [tone_alarm] notify neutral
INFO [logger] closed logfile, bytes written: 51045849
INFO [commander] Armed by external command
INFO [tone_alarm] arming warning
INFO [logger] Start file log (type: full)
INFO [logger] [logger] ./log/2026-01-07/18_11_10.ulg
INFO [logger] Opened full log file: ./log/2026-01-07/18_11_10.ulg
INFO [navigator] Using default takeoff altitude: 20.0 m
INFO [commander] Takeoff detected
]
```



Generare missioni con Intelligenza Artificiale

Possiamo utilizzare il seguente prompt per generare gli script con Gemini a partire da un testo in linguaggio naturale

You are an expert in MAVSDK Python (version 1.0+).
Generate ONLY an 'async def run_mission(drone):' function.
STRICT RULES:

1. Use 'from mavsdk.offboard import PositionNedYaw, OffboardError'.
2. Always move with: 'await drone.offboard.set_position_ned(PositionNedYaw(N, E, D, YAW))'.
3. Remember that in NED, altitude (D) is NEGATIVE to go up (e.g. -10.0 for 10 meters).
4. Before using set_position_ned, ALWAYS send an initial setpoint and then do 'await drone.offboard.start()'.
5. Never use 'drone.action.goto_location' unless global GPS is required.

CORRECT CODE EXAMPLE:

```
async def run_mission(drone):  
    await drone.action.arm()  
    await drone.action.takeoff()  
    await asyncio.sleep(5)  
    # Initial setpoint required for Offboard  
    await drone.offboard.set_position_ned(PositionNedYaw(0.0, 0.0, -5.0, 0.0))  
    await drone.offboard.start()  
    # Movement  
    await drone.offboard.set_position_ned(PositionNedYaw(5.0, 0.0, -5.0, 90.0))  
    await asyncio.sleep(5)  
    await drone.offboard.stop()  
    await drone.action.land()
```



Generare missioni con Intelligenza Artificiale

....e il seguente script per interfacciare PX4 con Gemini. Tuttavia la latenza elevata è dovuta alla generazione del codice

```
1 import asyncio
2 import os
3 import re
4 import google.generativeai as genai
5 from mavsdk import System
6
7 # Configure your API KEY here or as an environment variable
8 GOOGLE_API_KEY = "APY_KEY"
9
10 # ROBUST PROMPT: This prevents outdated code
11 SYSTEM_PROMPT = "YOUR_PROMPT"
12
13 genai.configure(api_key=GOOGLE_API_KEY)
14 model = genai.GenerativeModel('gemini-2.5-flash')
15
16 def extract_python_code(text):
17     """Extracts code from markdown blocks ```python ```"""
18     pattern = r"```python\s+(.*?)\s+```"
19     match = re.search(pattern, text, re.DOTALL)
20     if match:
21         return match.group(1)
22     return text
23
24 async def main():
25     # Initialize the drone
26     drone = System()
27     print("Connecting to PX4 simulator...")
28     await drone.connect(system_address="udp://:14540")
29
30     user_prompt = input("\nWhat should the drone do today? > ")
31
32     full_prompt = f"{SYSTEM_PROMPT}\n\nUser Request: {user_prompt}"
33
34     print("Gemini is processing the mission...")
35     response = model.generate_content(full_prompt)
36     raw_code = response.text
37
38     clean_code = extract_python_code(raw_code)
39
40     print("--- GENERATED CODE ---")
41     print(clean_code)
42     print("-----")
43
```

```
44 confirm = input("Do you want to execute this mission on the simulator? (y/n): ")
45 if confirm.lower() == 'y':
46     try:
47         # Create a local environment for executing the generated code
48         # Pass the already connected 'drone' object
49         exec_scope = {"drone": drone, "asyncio": asyncio, "math": __import__('math')}
50         from mavsdk.offboard import PositionNedYaw, OffboardError
51         exec_scope["PositionNedYaw"] = PositionNedYaw
52         exec_scope["OffboardError"] = OffboardError
53
54         exec(clean_code, exec_scope)
55
56         print("Executing mission...")
57         await exec_scope['run_mission'](drone)
58         print("Mission completed successfully.")
59
60     except Exception as e:
61         print(f"Error during execution: {e}")
62 else:
63     print("Mission canceled.")
64
65 if __name__ == "__main__":
66     asyncio.run(main())
67
```



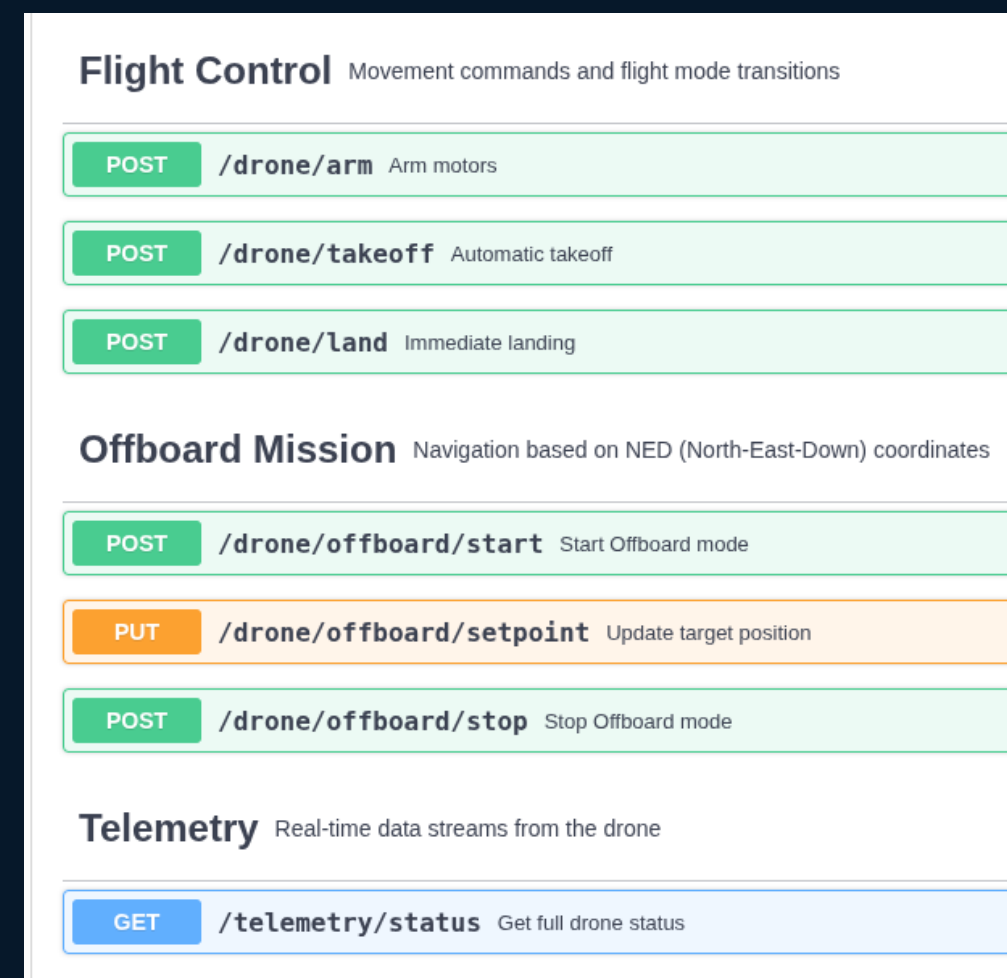
Sostituendo APY_KEY con la Google key e YOUR_PROMPT con il prompt visto prima, otteniamo un prompt in cui scrivere al drone cosa fare (in linguaggio naturale!)

Integrazione con LLM: Agenti AI e REST (PoC)

Un'integrazione ottimale con l'AI si otterrebbe interfacciando un Agente AI con delle *API REST*, in modo tale che un *LLM* acceda anche ai dati di telemetria: si avrebbe minore latenza e una maggiore visibilità dell'AI sullo stato del drone.

Idea:

- 1) Potremmo implementare le api rest da usare come interfaccia tra l'Agente e *Mavlink*
- 2) Scrivendo delle specifiche su *Swagger/YAML*, con uno script python e *Langchain* potremmo implementare un agente che guidi il drone tramite le API implementate al punto precedente
- 3) In un contesto di produzione (fuori dal simulatore) dovremmo prendere delle precauzioni: sarebbero opportuni dei controlli e limiti di sicurezza e non lasciare pieno controllo all'agente



```
from langchain_openai import ChatOpenAI
from langchain_community.agent_toolkits.openapi import azure_openai_utils # Or
from langchain_community.agent_toolkits.openapi.base import create_openapi_agent
from langchain_community.utilities.openapi import OpenAPISpec

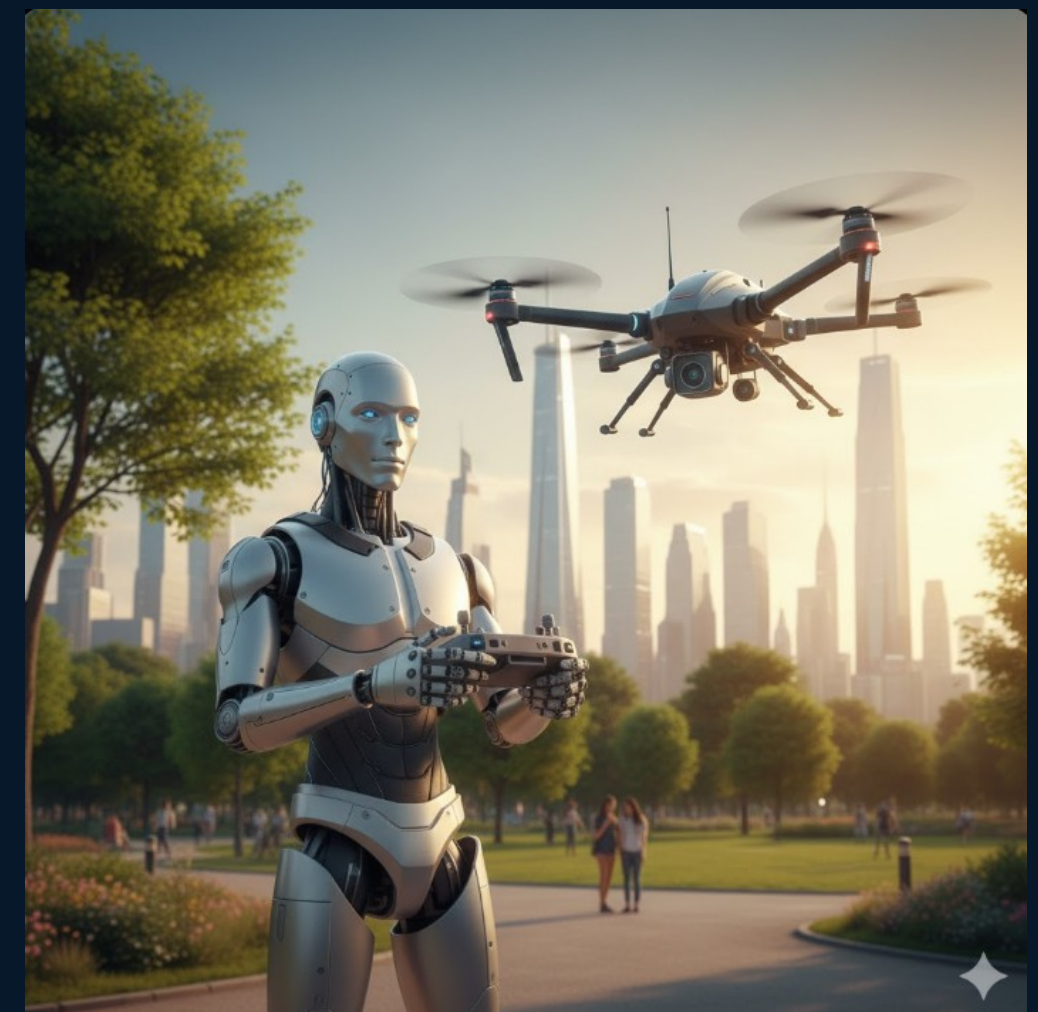
# 1. Load your OpenAPI schema (the .yaml file you provided)
spec = OpenAPISpec.from_file("drone_openapi.yaml")

# 2. Initialize the model (e.g., GPT-4 or Gemini via LangChain)
llm = ChatOpenAI(model="gpt-4", temperature=0)

# 3. Create the agent dedicated to drone control
# The agent will "figure out" on its own how to use /drone/arm, /drone/takeoff,
drone_agent = create_openapi_agent(spec, llm, allow_dangerous_requests=True)

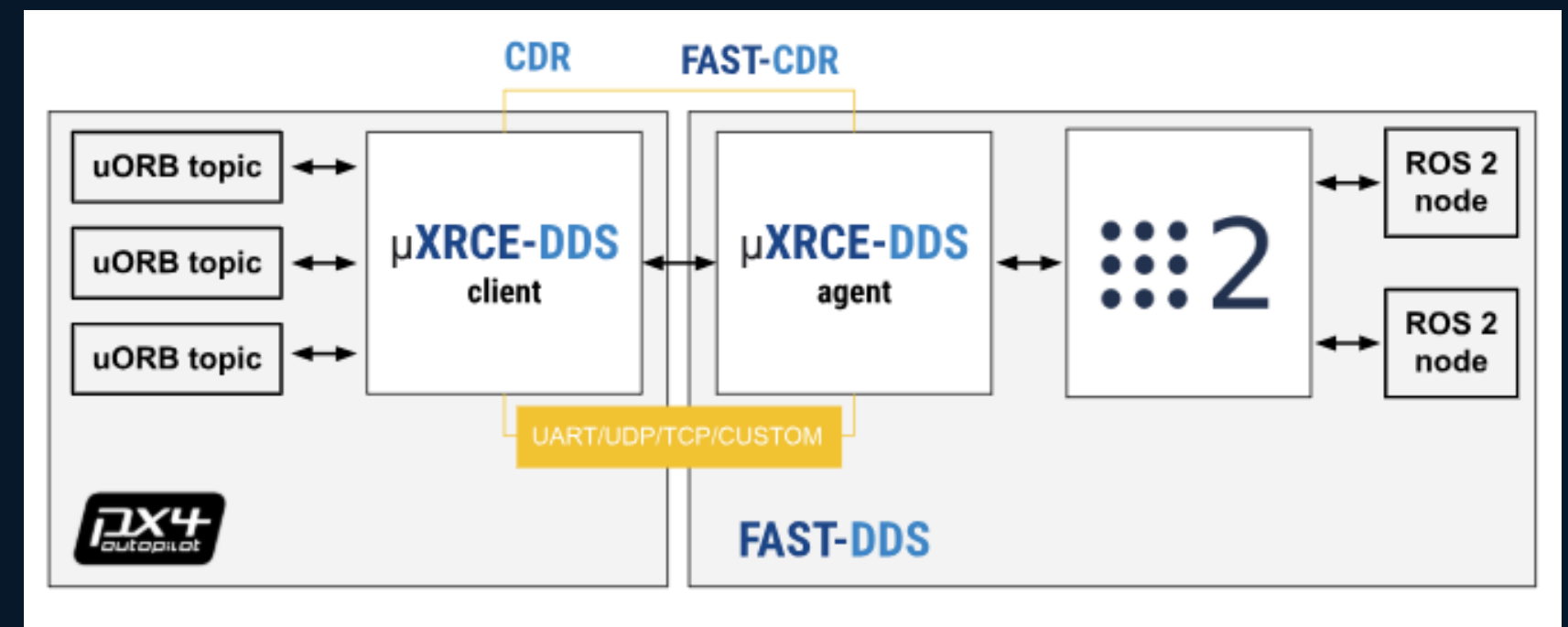
# 4. Example of a complex command
user_command = "Check if the sensors are okay. If they are, take off and climb"

# In modern LangChain, use .invoke() instead of .run()
drone_agent.invoke({"input": user_command})
```



Integrazione con ROS 2 (Robotics Operating System)

- ROS è un framework per lo sviluppo di applicazioni robotiche, permette di gestire più robot contemporaneamente e permette di interfacciarli allo stesso modo in cui si potrebbero interfacciare più microservizi
- Mavlink permette di integrare PX4 con ROS 1 (ormai obsoleto), rendendo possibile la gestione del volo ad alto livello (C++/Python) tramite un sistema di nodi, service, topic e messaggi sincroni / asincroni
- Di recente, l'integrazione con ROS 2 avviene tramite *CDR* e *Fast-CDR* che si occupano di convertire i messaggi uORB in messaggi ROS, permettendo ai rispettivi nodi ros di leggerli
- Con più istanze di PX4 / ROS è possibile programmare uno sciame di droni e gestirli come se fossero thread e sfruttando vari moduli di robotica...

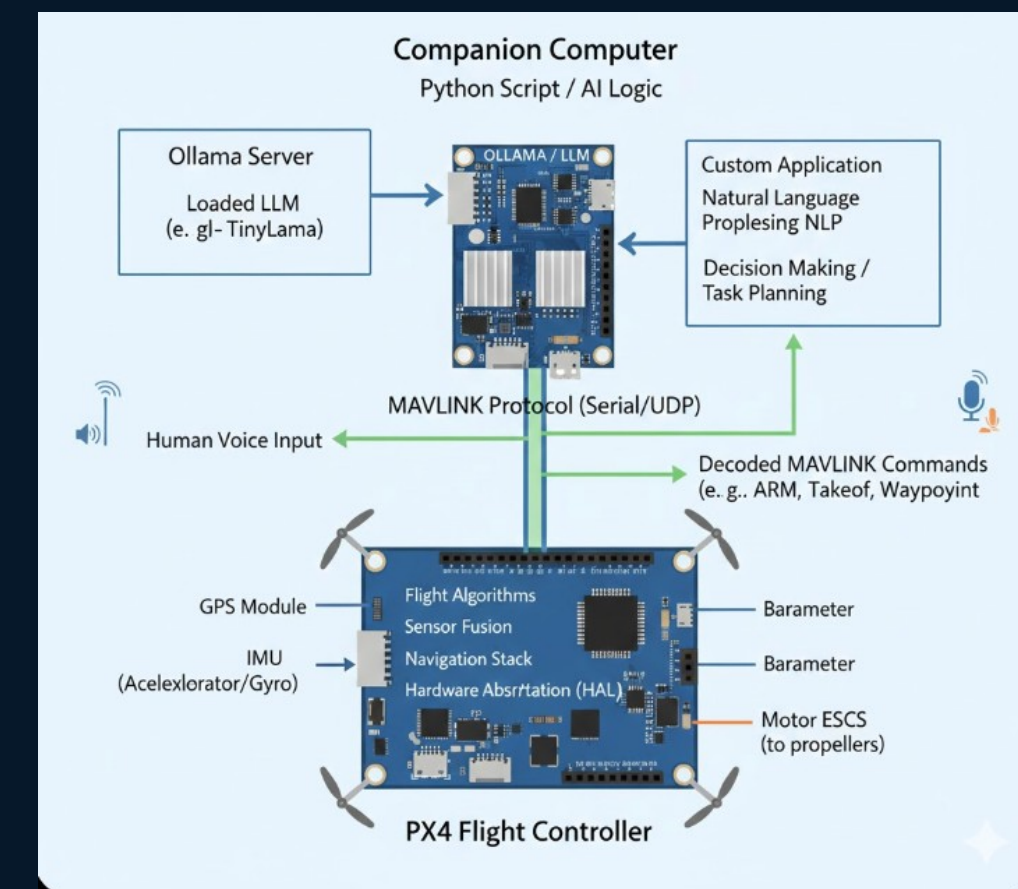


Rendendo possibili scenari fantascientifici (ma già noti)



Possibili idee future

1. Tramite *Reactylon* (reactylon.com) implementare soluzioni di realtà aumentata sul video ricevuto dal drone, stampando dati di telemetria su occhiali XREAL
2. Installare Ollama e un piccolo LLM su Companion Computer per dotare il drone di un potere decisionale su piccole missioni, senza usare Gemini



“Una volta che avrai volato, camminerai sulla terra con lo sguardo rivolto verso il cielo perchè è là che vorrai tornare.”

Leonardo Da Vinci