ChatGPT

Data Infrastructure and Development Pipeline for a Dual-Mode Transforming Robot

Introduction

We propose a robust data infrastructure and software stack for a **transformer robot** that can operate in two modes: a quadcopter **flight mode** and a wheeled **ground mode**. This design is inspired by Caltech's Multi-Modal Mobility Morphobot (M4), which can "roll on four wheels, turn its wheels into rotors and fly" [1]. Our simplified version uses the same four rotor-wheels for both driving and flying, with a mechanism to tilt the wheel pods between horizontal (drive) and vertical (flight) positions. The system will be **manually operated via an RC controller** in its initial phase, but is built with real-time telemetry, comprehensive data logging, and hooks for future autonomous upgrades in mind.

Key Requirements:
- **Manual RC Control:** The operator uses a standard RC transmitter to drive and fly the robot. A mode-switch (e.g. an RC switch) triggers the transformation between driving and flying.
- **Two Modes of Locomotion:** In ground mode, the robot drives on four wheels like an agile RC car. In flight mode, the wheel pods rotate 90° upward to act as quadcopter rotors, allowing short-hop flight. This transformation is powered by motorized hinges with lock positions and feedback sensors for confirmation [2].
- **Real-Time Telemetry:** The robot must stream critical telemetry (e.g. attitude, altitude, speed, battery status, etc.) live to a ground station, so the operator gets immediate feedback during operation.
- **Data Logging:** All sensor readings (camera feeds, LiDAR scans, IMU, GPS, wheel encoders, hinge angles, controller states, etc.) and robot state information are recorded for post-run analysis. This is crucial for debugging and for developing future autonomous capabilities.
- **Simulation Support:** The development pipeline should include simulation tools (such as Gazebo or similar) to test the robot's behavior in both modes. Simulation will be used for software-in-the-loop (SITL) and hardware-in-the-loop (HIL) testing before full deployment.
- **Future Autonomy-Ready:** While autonomy (e.g. obstacle avoidance, SLAM, or automatic mode switching) is not required in phase 1, the architecture should accommodate adding a high-level autonomy computer (for example, an NVIDIA Jetson) and algorithms (e.g. ROS 2 navigation stack, vision processing) down the line.

In the following sections, we outline the proposed **system architecture**, including data flow between sensors, controllers, and actuators, the recommended **middleware (ROS 2)** for integration, the **simulation environment and development pipeline**, and the **logging/telemetry setup**. We also compare hardware choices – **Pixhawk**, **Teensy**, and **Jetson Nano** – for flight and ground control, discussing their roles, pros, and cons in this context.

System Architecture Overview

Hardware Components and Roles

Our design employs a modular hardware architecture with dedicated components for flight control, high-level computation, and sensor/actuator interfacing. The table below summarizes the key hardware elements:

Component	Role in System	Notes
Flight Controller (FCU) *e.g. Pixhawk 4*	Low-level **flight control unit** running autopilot firmware (PX4 or ArduPilot). Maintains stable flight (attitude control, IMU sensor fusion) and reads RC inputs. Also controls motors (rotors) and transformation servos.	Chosen for its real-time control loop and reliability. Provides onboard IMU, barometer, etc. (Pixhawk example) [3]. Receives RC receiver signals for manual control.
Companion Computer *e.g. NVIDIA Jetson Nano*	Onboard **mission computer** running a full OS (Linux) and middleware (ROS 2). Handles high-level processing: sensor data logging, real-time telemetry streaming, and future autonomy algorithms (vision, SLAM, path planning).	Not real-time, but powerful for computation. Communicates with FCU via fast link (serial UART or Ethernet) using MAVLink or ROS 2 protocols [4]. In phase 1, mainly used for data logging and telemetry; in future, can run object detection or navigation tasks [5].
Microcontroller (Optional) *e.g. Teensy 4.x*	Auxiliary **I/O controller** for custom sensors/actuators. Can interface with wheel encoders, current sensors, and control wheel motors in ground mode if the flight controller does not handle this. Publishes these sensor readings into the system (via ROS 2 or direct to companion).	Provides real-time interfacing for devices not supported by FCU. For example, can read high-rate wheel encoder ticks and motor driver commands for driving. In our design, a Teensy can report wheel motion data during testing [6].
Rotor-Wheel Motors & ESCs	Four brushless **motors with ESCs** that serve dual purpose: driving the wheels and spinning propellers for lift. Controlled via PWM outputs from the flight controller (for flight mode) or possibly the microcontroller (in drive mode).	Each motor is equipped with a wheel (tire) around a ducted propeller. In drive mode, motors spin wheels at low RPM for traction; in flight mode, motors spin props at high RPM for thrust. ESCs can be commanded by Pixhawk (in flight) or by an alternate controller for driving.

Component	Role in System	Notes
Transformation Actuators	Four servo or linear actuators (one per wheel pod) that tilt the rotor-wheel assemblies between 0° (horizontal, wheel down) and 90° (vertical, rotor up). Controlled by the flight controller or microcontroller.	These actuators enable the **mode switch**. They have **feedback sensors** (potentiometers or encoders) to verify that each pod has locked into the correct position [2] before switching modes. The Pixhawk or companion can monitor these sensors to ensure safe transformation.
Sensors – IMU & Barometer	Inertial measurement unit and altimeter for flight stabilization. Typically built into the flight controller (Pixhawk has internal IMUs, gyros, accelerometers, and barometer).	Used by autopilot for attitude estimation and altitude control. Critical for stable flight. These run at high update rates on the FCU's real-time firmware.
Sensor – GPS Receiver	Global Positioning System receiver for obtaining location and speed. Often connected to the flight controller (e.g. Pixhawk GPS+Compass module via UART/ CAN).	Provides position data for telemetry and future autonomous navigation. In manual phase, GPS can log the path and assist in stabilization (e.g. position hold if needed).
Sensor – Camera	Onboard **camera** (RGB or stereo) for vision. e.g. an HDMI or CSI camera connected to the Jetson.	In phase 1, used for recording video and possibly FPV streaming to the operator. In future, used for object detection or visual SLAM. The companion computer captures and logs image frames (via ROS 2 image topics or GStreamer).
Sensor – LiDAR	2D or 3D **LiDAR scanner** for distance sensing and mapping (e.g. a spinning 2D RPLidar or 3D Velodyne). Connected to the Jetson via serial/ USB or to microcontroller.	Provides range data of surroundings. Logged for post-analysis; later can feed into obstacle detection or mapping algorithms. Not used by the flight controller directly, but by the companion computer for situational awareness.

Component	Role in System	Notes
RC Receiver & Transmitter	Standard 2.4 GHz **RC radio system** for manual control. The receiver is connected to the flight controller (PPM/SBus input).	The operator controls throttle, steering, and mode switch via the transmitter. The FCU reads these inputs each cycle. For ground driving, either the FCU passes commands to wheel motors or a separate channel is fed to the microcontroller for throttle/steering.
Telemetry Radio (optional)	A wireless modem (e.g. 915 MHz SiK radio or Wi-Fi module) for sending telemetry data to a ground station laptop.	If using a Pixhawk, a SiK radio can plug into its TELEM port to stream MAVLink data to a PC running Ground Control Station software. Alternatively, the Jetson can use Wi-Fi to transmit telemetry and video to the ground.
Ground Station Laptop	Operator's laptop running monitoring/control software. In flight, typically runs *QGroundControl (QGC)* or similar to display telemetry. Can also run ROS 2 tools to visualize data.	Receives real-time data from the robot. Not required for basic RC control, but critical for feedback and logging. Can record incoming telemetry or connect to the robot's ROS 2 network over Wi-Fi to log data (in addition to onboard logging).

Operational Modes and Transformation: In **Ground Mode**, the wheel pods are locked in horizontal position (wheels down) at 0° and the robot drives like a four-wheel vehicle. We use differential (tank-style) steering by varying wheel speeds, enabling zero-radius turns for indoor agility [7]. Suspension on each wheel pod smooths out rough terrain. In **Flight Mode**, the pods rotate up to 90° (props up) via the hinge actuators, effectively turning the wheels into upward-facing propellers [8]. The same motors then provide lift as a quadcopter. **Mode switching** is initiated by the operator (e.g. flipping a switch on the RC transmitter). The transformation takes only a few seconds (target <10 s) [2]. During the switch, the hinge actuators move the pods and lock into place; embedded angle sensors confirm each pod's position for safety [2]. Once in the new configuration, control authority is handed off to the appropriate drive logic (ground driving or flight stabilization).

- *In ground mode:* The Pixhawk flight controller can be configured to either pass RC commands directly to the wheel motors (effectively acting as an RC motor mixer when not flying), or we can use the Teensy to control the drive motors. For simplicity, one approach is to let the Pixhawk run a **"manual pass-through"** mode for ground operation – reading the throttle/steering RC inputs and outputting corresponding motor PWM signals without trying to stabilize flight. (ArduPilot's Rover firmware could be used for driving, but switching between Rover and Copter firmware on one controller is non-trivial, so a pass-through or a second controller is preferred.) Alternatively, the RC receiver could output on additional channels directly to a motor controller (or Teensy) for wheels. In our architecture, we plan to use the **Pixhawk to read all RC inputs** and use a spare channel (tied to the mode switch) to signal the Jetson/Teensy when to take over wheel control. During ground mode, the

flight controller can be kept disarmed (so it doesn't attempt to stabilize as a quad) but still powered to relay RC signals and monitor IMU (for logging). The Teensy (or Pixhawk in pass-through) would drive the wheel motors according to the operator's commands. Wheel encoders measure distance traveled and speed, and this data is sent to the Jetson for logging and future odometry use.

- *In flight mode:* The Pixhawk (running PX4 or ArduCopter firmware) takes full control. Upon mode switch, the Pixhawk arms its motors and uses its IMU, gyro, and control loops to stabilize the drone. The operator's RC inputs now control collective throttle, pitch, roll, yaw of the quadcopter (in a stabilized flight mode). The wheel motors (now props) are driven by the Pixhawk's mixer. The hinge actuators are locked in the flight position and taken out of the control loop (they remain static at 90°). During flight, the Pixhawk uses its onboard sensors and possibly GPS to maintain stability; it can also publish its state (attitude, altitude, etc.) to the companion computer. The Jetson at this time may record camera and LiDAR data and stream essential telemetry to the ground, but it does not intervene in the control loop (manual control is fully via RC→Pixhawk).

- *Mode coordination:* To prevent any conflict, we implement a simple **mode manager** (could be a ROS 2 node on the Jetson) that listens to the RC mode switch status (via a MAVLink message from Pixhawk or a GPIO from the receiver). When ground mode is active, this manager ensures the Pixhawk remains disarmed (so it won't spin props) and instead either enables the Teensy motor controller node or configures Pixhawk to pass-through. When flight mode is active, the manager commands the hinge actuators to move to flight position (via Pixhawk servo outputs or Teensy), waits for their **"locked" confirmation,** then signals the Pixhawk to arm and take off. This coordination can initially be a manual procedure (the pilot waits a few seconds after flipping the switch to throttle up), but the infrastructure supports automating it in future.

Communication and Data Flow

The system is integrated with a robust communication network, primarily using **ROS 2 middleware** and the MAVLink protocol for command and telemetry. The data flow between sensors, processors, actuators, and logging is illustrated below:

- **Onboard Autopilot ⟷ Companion Computer:** The Pixhawk flight controller communicates with the Jetson companion using a high-speed serial link (UART) running the MAVLink protocol. MAVLink is a lightweight messaging protocol widely used in drones for telemetry and commands. In our setup, we connect Pixhawk's TELEM2 port to the Jetson's UART (or USB) port. This link allows the Pixhawk to stream its state (IMU data, attitude, GPS, battery, RC input status, etc.) to the Jetson at high rate, and allows the Jetson to send commands or receive telemetry. This architecture (flight controller + companion computer) is standard in advanced drones: the two are connected via serial or ethernet and "typically communicate using the MAVLink protocol (or uXRCE-DDS in ROS 2) [4]." The companion can also act as a routing node to pass data to ground over other links [9]. In our design, Pixhawk will send state updates over MAVLink to the Jetson continuously. We will run a **ROS 2 node on the Jetson (e.g. MAVROS2 or PX4-ROS2 bridge)** that translates MAVLink messages into ROS 2 topics for easy consumption and logging [10]. For example, Pixhawk's attitude data becomes a `/vehicle/attitude` topic, GPS data becomes `/vehicle/gps`, etc., in ROS. This makes the flight data available to any ROS 2 component in the system (and to the logging subsystem). If needed, the Jetson can also send commands back to Pixhawk (for future autonomy, e.g., offboard control commands or mode switch instructions), but in the initial manual phase, outbound commands from

Jetson will be minimal (perhaps an emergency kill switch or simple waypoint test commands). The MAVLink link also allows the Jetson to configure Pixhawk parameters or receive status text messages for debugging.

- **Microcontroller ↔ Companion:** The Teensy board, if used, connects to the Jetson via USB or a serial port. We will run a lightweight protocol or **micro-ROS** client on the Teensy to integrate it into the ROS 2 ecosystem [4]. For instance, the Teensy can publish wheel encoder counts and hinge angles as ROS topics (e.g. `/wheel_encoder_fl`, `/hinge_angle_fl` for front-left wheel, etc.). It can also subscribe to commands if we decide to offload wheel motor control to it (e.g., a `/drive_velocity` topic commanding wheel speeds). Alternatively, without ROS on Teensy, we could use a custom serial message that the Jetson interprets and republishes. Using micro-ROS is advantageous as it makes the Teensy a first-class ROS 2 node on the network, with minimal overhead. Thus, all sensor data from the microcontroller (encoders, specialized sensors) flows into the Jetson's ROS 2 network, timestamped and logged.

- **Onboard Sensors → Processing:** The Jetson directly interfaces with high-bandwidth sensors like the camera and LiDAR. For the camera, the Jetson (running Linux) can use standard drivers (V4L2 or CSI) to capture video frames. A ROS 2 camera node will publish images to a topic (e.g. `/front_camera/image_raw`). We can also compress or downsample the feed if needed for telemetry. For the LiDAR, many models have ROS drivers that publish laser scan data (e.g. a 2D lidar publishes a `/scan` topic at some frequency). The Jetson's CPU/GPU is capable of handling these data streams. During operation, the Jetson will primarily **log** this data (the operator might not need LiDAR in real-time, but we want it recorded). However, the Jetson could perform some real-time processing even in manual mode – for example, constructing a map of the environment in the background or providing an FPV video stream with overlay to the operator. We will ensure the data flows into ROS 2 so it's available for such uses.

- **Actuator Commands:** In flight mode, actuator commands are generated by the Pixhawk's flight control loops. Pixhawk outputs PWM signals to the four motor ESCs (throttle commands) and to the four hinge servos (to hold them at 90° during flight, or to deploy/stow them when commanded). In ground mode, if Pixhawk is in pass-through, it outputs PWM to the motors based on RC inputs (like a mixing for differential drive). If using the Teensy for drive, then the Teensy would output PWM signals to the motor controllers (ESCs) for wheels. In either case, the motor controllers (ESCs) ultimately receive command signals to set motor speed. All actuators (motors and servos) have their power supplied from a common battery (through appropriate regulators). The Pixhawk can control the servos for transformation (as part of a custom Pixhawk channel setup triggered by the RC switch or a companion command). The **data flow for actuators** is mostly one-way (from controllers to the physical motors/servos), but we include feedback where possible: for example, servo positions via potentiometers (to confirm hinge angle) go back into the Teensy or Pixhawk analog inputs; motor current draw can be measured and fed back (via power module to Pixhawk for overall current, and via inline sensors to Teensy for per-motor current, if desired) [11].

- **RC Control Path:** The RC receiver outputs are wired into the Pixhawk's RC input. The autopilot firmware reads these at each control loop iteration. The **RC signals** (like stick positions) are also encoded into MAVLink telemetry (so the Jetson and ground station know the user's inputs, useful for logging). In manual flight, Pixhawk uses the RC inputs to directly control the motors (in a stabilized manner). In manual driving, Pixhawk can either pass them to wheels or the Jetson/Teensy can read

them (through MAVLink or an auxiliary connection). Initially, we assume Pixhawk handles the RC mixing for simplicity. The pilot thus uses the same transmitter for both modes, making control seamless: one toggle for mode, sticks for movement. The system will ensure **mode isolation** to avoid conflicting commands – e.g., when in flight mode, wheel-drive commands are ignored except for transformation lock, and vice versa.

- **Ground Station Communication:** For real-time telemetry offboard, we have two main channels: **MAVLink Telemetry Radio** and **Wi-Fi link**. We can utilize a long-range 915 MHz MAVLink radio (which is basically a serial modem) connected to Pixhawk's TELEM1 port. This will continuously broadcast telemetry data to the ground station computer running QGroundControl (QGC). QGC will display vital info like attitude, GPS position, altitude, battery, RC signal strength, etc. This is a proven setup: Pixhawk and a ground laptop communicate via MAVLink over the telemetry link [12] [13]. In fact, during hardware-in-loop testing, "the Pixhawk and ground station laptop will communicate over MAVLink using a USB or telemetry radio" [14] [15]. We will leverage that for flight telemetry and perhaps basic drive telemetry (Pixhawk can send wheel encoder info if configured as a custom MAVLink message, or just send it as part of a ROS topic via the Jetson). Additionally, the Jetson will be equipped with a Wi-Fi module. When in range (e.g. indoors testing), the Jetson can join a network with the ground laptop and act as a ROS 2 bridge or use a **MAVLink router**. A MAVLink router on the Jetson can take the data from Pixhawk and forward it over UDP Wi-Fi to both QGC and to a ROS 2 backend on the laptop [9]. Simultaneously, large data (camera stream) can be sent over Wi-Fi (since the 915 MHz link can't handle video). The operator can then view an FPV video feed on the laptop or a heads-up display while still receiving QGC telemetry. In summary, critical numeric telemetry flows through MAVLink (robust, low-bandwidth) and high-bandwidth data (images, point clouds) flows through Wi-Fi in ROS 2 or custom channels.

- **Logging and Storage:** The Jetson, as the central computer, will log all incoming data streams (from Pixhawk, Teensy, camera, LiDAR) using ROS 2 bag files. We detail this in the **Logging** section, but note here that the data flow includes writing to storage. The Jetson has local storage (e.g. SSD or SD card) where rosbag2 will record topics. We will configure the logging to ensure minimal performance impact (e.g. using appropriate QoS and perhaps compression for high-volume topics). The Pixhawk also has an SD card where it stores its own flight logs (.ULog or .BIN files including IMU, RC, etc.). Those can serve as a backup or cross-reference.

This architecture ensures **modularity**: the flight controller handles fast control loops and critical real-time tasks, the companion handles heavy computation and communication, and the microcontroller handles any additional real-time interfacing needed. The data flow is largely mediated by ROS 2, which acts as the "central nervous system" – all sensor data and control messages are published as topics that can be logged and visualized. This ROS-based design was chosen because it "allows multiple packages to run simultaneously and communicate" in real time [16], which is ideal for integrating a complex robot. The foundation of our system thus relies on ROS, similar to how other drone research projects set up real-time data acquisition [17].

Software Stack and Middleware (ROS 2)

We recommend using **ROS 2 (Robot Operating System 2)** as the integration middleware for this project. ROS 2 provides a distributed publish/subscribe architecture with real-time capable communication (using

DDS – Data Distribution Service) and a rich ecosystem of robotics packages. This will facilitate the integration of sensors, control algorithms, and logging in a flexible way.

ROS 2 Nodes: Each major function runs in a separate ROS 2 node (or set of nodes), allowing for clear modularity:

- A **MAVLink bridge node** runs on the Jetson to interface with the Pixhawk. For example, we can use **MAVROS (or MAVROS2)**, which is a well-supported ROS package that acts as a translator between MAVLink messages and ROS topics. When connected to the Pixhawk, MAVROS will spawn topics for the vehicle's state (pose, velocity, IMU, etc.) and services/actions for sending commands. This essentially mirrors the autopilot's data into ROS. (In ROS1 this is MAVROS; in ROS2, PX4 provides a similar bridge through the micro-RTPS bridge or a community MAVROS2 package.) Using such a node, "MAVLink signals are communicated to ROS" enabling our companion computer to access flight telemetry [10]. This is crucial for logging and any future high-level control. If using the PX4 flight stack, an alternative is the **PX4-ROS2 bridge** which uses uORB messages over DDS (a more direct ROS 2 integration). Either approach gives us ROS topics like `/imu`, `/gps`, `/battery_state`, etc., published at high rates.

- **Sensor driver nodes** run for the camera and LiDAR. For the camera, we might use ROS 2 package `image_tools` or OpenCV to grab frames and publish `sensor_msgs/Image`. For LiDAR, if it's a 2D lidar like RPLIDAR, a node (e.g. `rplidar_ros2`) will publish `sensor_msgs/LaserScan`. A 3D lidar would publish point clouds (`sensor_msgs/PointCloud2`). These nodes will be configured to use appropriate QoS (e.g. best-effort for camera if some frames drop, reliable for LiDAR scans).

- **Microcontroller interface node:** If the Teensy is used without micro-ROS, we'd have a node on Jetson that reads serial data from it (e.g. encoder ticks, hinge angles) and turns them into ROS topics. Alternatively, with micro-ROS on the Teensy, the Teensy becomes a ROS node itself on the network. In that case, the microcontroller will directly publish topics (the ROS 2 DDS stack can run in a microcontroller with the micro-ROS client library, using XRCE-DDS). This would reduce custom parsing code and utilize ROS 2 messages for things like encoder counts. The PX4 documentation notes that Pixhawk and companion can even communicate by micro-ROS (uXRCE-DDS), though MAVLink is more common [4] – similarly, our Teensy could either speak ROS 2 or a custom protocol; we propose to leverage ROS 2 for consistency.

- **Telemetry/communication nodes:** We will run a **MAVLink router** or forwarding service on the Jetson. This small service (e.g. `mavlink-router` by Intel) can take MAVLink data from Pixhawk and send it to multiple endpoints – one being the local MAVROS, another being the ground station via UDP over Wi-Fi, and optionally the 915 MHz radio (if the radio is connected to Jetson rather than Pixhawk). This allows simultaneous use of QGroundControl and ROS 2 without conflict. Additionally, we might use a **video streaming node** to handle compressing and streaming the camera feed (for example, using GStreamer to stream an H.264 video from the Jetson to a ground PC or even to QGroundControl which can display video). ROS 2 nodes can also use RTPS to communicate between robot and ground if on the same network, but for long range we rely on MAVLink.

- **Future autonomy nodes:** The ROS 2 framework will make it straightforward to add autonomy. For example, adding a **SLAM node** (like RTAB-Map or Nav2's SLAM toolbox) to use LiDAR or camera data to build a map, or a **navigation stack** (ROS 2 Nav2) to plan and control autonomous movement in

ground mode. We can also integrate an **object detection node** (leveraging Jetson's GPU for running a DNN on camera images). These would subscribe to sensor topics and publish decisions (like a goal or obstacle alert). In the current phase, these nodes might run in logging or passive mode (just analyzing data without controlling the robot) so we can test algorithms using the data we collect. When we're ready, we could allow these nodes to publish commands (e.g. override RC input or provide guided mode waypoints to Pixhawk via MAVROS) to realize semi- or fully-autonomous operation.

ROS 2 vs Alternative Middleware: We choose ROS 2 because it is the **standard in robotics for integration** and it supports real-time distributed systems. ROS 2's DDS can handle our use-case of multiple producers/consumers of sensor data with low latency. It also has out-of-the-box support for our needs: logging (rosbag2), visualization (RViz, rqt), and a vast array of community drivers (for cameras, lidars, etc.). An alternative could be a custom MQTT or DDS implementation, but ROS 2 essentially wraps DDS with robotics-friendly abstractions. Another alternative is **MAVSDK** (a higher-level API for autopilot control) – we might use MAVSDK for writing simpler control logic, but it doesn't by itself handle non-flight sensors as ROS does. In fact, PX4 notes that *"MAVSDK is easier to use, while ROS provides more pre-written software for advanced cases like computer vision"* [18]. Since our project explicitly involves cameras and LiDAR (advanced perception), ROS 2 is the appropriate choice.

Real-Time Considerations: One might wonder if ROS 2 (running on Linux) can handle real-time control. We are mitigating this by letting the Pixhawk handle all **hard real-time loops** (IMU at 1 kHz, motor PWM outputs at ~400 Hz or more). The Jetson (ROS 2) operates at softer real-time (tens of Hz to a few hundred Hz for sensor reading and logging, which is fine). We will ensure the Jetson's OS is optimized (using a preempt-RT kernel if needed, and setting process priorities) for consistent timing, especially for tasks like image capture to avoid dropping frames. But a slight jitter on the companion won't affect flight stability – Pixhawk runs an RTOS for that.

Software on Pixhawk: On the flight controller, we will run a well-established autopilot firmware – either **PX4** or **ArduPilot** (ArduCopter). Both support quadcopter control and can be configured for custom servo outputs for our tilting mechanism. PX4 has a VTOL framework (for tilt-rotor planes) which might be adaptable, but an easier path is to treat the tilt as a manual servo action tied to an RC switch. We will likely use PX4 or ArduPilot in a mode that allows manual attitude control (e.g. Stabilized or AltHold modes in ArduPilot for easy manual flying). The autopilot firmware will be configured to accept our particular geometry (if needed, we can slightly modify motor mixing if the physical layout isn't symmetric – but since it's an X quad, standard mixing works). **ArduPilot** also has a "Ground Mode" concept for rovers, but switching on the fly isn't supported – so we will just define one vehicle type (quadcopter) and ensure it can handle being on the ground when not flying. There is precedent for quads driving: some drone firmwares support a "truck mode" where motors drive wheels when not flying, but if not, we implement that externally. The Pixhawk's software will also handle failsafes (loss of RC, low battery – e.g., initiate landing) to ensure safety.

Operating System and Dependencies: The Jetson Nano will run **Ubuntu 20.04 or 22.04** with ROS 2 Foxy or Humble. We'll use Ubuntu since ROS is well-supported there. The development environment will use Docker or native builds for ROS 2 packages. The Pixhawk (if it's a Pixhawk 4 or 6) runs NuttX OS internally (for PX4) or ChibiOS (for ArduPilot) – that's abstracted away from our perspective. The Teensy runs an embedded C++ program we write (with Arduino framework or straight C/C++ and the micro-ROS client as needed).

In summary, the **software stack** comprises: **Pixhawk autopilot firmware** for core control, **ROS 2 on Jetson** for integration of all sensors and future algorithms, and optional **microcontroller firmware** on Teensy for auxiliary control. This stack meets the requirement for initial manual control (Pixhawk + RC covers that) and is structured to seamlessly incorporate autonomous behaviors later by simply adding ROS 2 nodes or sending setpoints to Pixhawk. It also inherently supports simulation (since ROS 2 and PX4 both have simulation capabilities, discussed next).

Simulation Tools and Development Pipeline

To develop and test this complex system safely and efficiently, we will employ a combination of **software-in-the-loop (SITL)** simulation, **hardware-in-the-loop (HIL)** testing, and gradual integration tests. Simulation is critical for verifying the control logic and data flow before risking the real hardware, especially for flight.

Gazebo Simulation: We will use the Gazebo simulator (either classic Gazebo 11 or the newer Ignition Gazebo) as our primary physics simulation environment. Gazebo can simulate both ground vehicle dynamics and aerial dynamics with appropriate plugins. The plan is to create a URDF (Unified Robot Description Format) model of the robot with two configurations (or a movable joint for the wheel pods). The model will include the four rotor-wheel assemblies, hinge joints, and approximate mass/inertia properties. We can leverage existing Gazebo plugins for multi-rotor flight (for example, the **rotors_simulator** plugin or PX4's Gazebo plugin) to simulate the quadcopter physics. For ground mode, we'll give the wheels friction and use Gazebo's vehicle model capabilities (akin to simulating an RC car).

A simpler approach initially is to simulate **each mode separately**: We can simulate the flight mode using a standard quadcopter model (since from Pixhawk's perspective it's a quadcopter when flying). For the ground mode, we simulate a four-wheeled rover. The transformation itself (dynamic switching in simulation) can be added later; initially we might just reset the simulation with a different model to test each mode. However, an advanced approach could be to have a joint in the URDF for the pod angle and actively change it during simulation – we can script this or even have a simulated servo. This would allow testing the mode switch logic virtually (e.g., commanding the joint to rotate and verifying control handoff).

PX4 SITL: For flight control, we will harness **PX4's Software-In-The-Loop** functionality. PX4 autopilot (and ArduPilot as well) provide SITL binaries that run the autopilot logic on a PC, which can be connected to Gazebo. In practice, we will run a PX4 SITL instance on our development PC which thinks it's controlling a quadcopter. Gazebo, with a quadcopter model, will simulate the physics of that quad. This setup has been demonstrated: you launch PX4 SITL, launch Gazebo with a vehicle model, and the drone in Gazebo "flies" as if the PX4 were running on it [19]. We will integrate ROS 2 in this loop by running the same MAVROS and sensor nodes against the simulated robot. In SITL, the PX4 (or ArduPilot) sends MAVLink data to UDP, which MAVROS can listen to (no actual Pixhawk hardware needed). We can also simulate sensors: Gazebo can produce synthetic camera data (rendered scenes from the robot's perspective) and LiDAR scans (using plugins that simulate laser range data). These will feed into our ROS 2 nodes as if they were from real sensors, allowing us to test our logging pipeline and any autonomy code in a safe, repeatable environment. As part of our pipeline, *we will inject mock data using Gazebo/PX4 SITL to ensure the telemetry topics can handle the expected bandwidth* [20]. For example, we'll simulate a worst-case scenario (camera publishing at full frame rate, LiDAR at high scan rate, plus high-frequency IMU) and verify our system can transmit and log without dropping data. This addresses performance tuning early.

Hardware-in-the-Loop (HIL): After validating in pure simulation, we will progress to HIL tests. In HIL, the **actual Pixhawk hardware** is connected, but the robot is either tethered or motors are not running, and parts of the environment are simulated. One HIL approach is **Wired HIL**: connect the Pixhawk via USB to the PC, and run the simulation such that Pixhawk thinks it's flying but we intercept its motor commands. For example, ArduPilot can do HIL where Pixhawk reads simulated sensor values from the PC and sends motor outputs back to the sim. However, HIL at the physics level is complex and often SITL is sufficient. Instead, our HIL will focus on partial system tests: for instance, mounting the robot on a test stand (wheels off ground or props off for safety) and having the Pixhawk control motors at low throttle while the Jetson reads real sensors like the IMU and encoders. We can also perform a **bench test** where the Pixhawk is connected to the Jetson and a logging laptop, but instead of real flight, we feed it simulated sensor data. The plan is to **run hardware-in-loop bench tests where the Pixhawk publishes synthetic flight data while the Teensy reports wheel motion, verifying synchronization** between the two data sources before field trials [20]. In practice, this might mean using PX4's HITL mode: QGroundControl can take Pixhawk's data and forward it to MAVROS, and simultaneously we generate some wheel encoder ticks on the Teensy (perhaps spinning a wheel on a motor on the bench) to see that the timestamps line up in the logged data. This ensures that when we combine data from different sources (Pixhawk vs Teensy) in our ROS bag, they are time-aligned and complete. Our team explicitly plans to do this step to iron out any timing issues [21].

Full System Testing: Once simulation and HIL give the green light, we'll conduct controlled environment tests for each mode. Initially, tethered or low-altitude flights to ensure stability, and simple driving tests. We will use the same ROS 2 logging setup during these tests to gather data. Each test's data can be played back in ROS 2 to analyze vehicle behavior, tune control parameters, or improve our simulation model (this is an iterative process – log data can calibrate the sim).

Simulation Tools Summary:
- *Gazebo:* Physics simulation for both aerial and ground dynamics, with sensor simulation (camera, LiDAR). Essential for developing and testing control logic and sensor processing without physical risk.
- *PX4/ArduPilot SITL:* Autopilot simulation that runs on a PC and interfaces with Gazebo. It provides a realistic flight control output that we can test our companion software against.
- *RViz:* ROS 2 visualization tool to visualize the robot in simulation and sensor data (point clouds, camera images, etc.) in real-time. This helps verify that our ROS topics are publishing correctly and that sensor data aligns (e.g., seeing a LiDAR point cloud overlaid on a map).
- *QGroundControl (simulation mode):* We can connect QGC to the SITL via UDP to visualize how the autopilot "feels" during simulation (virtual HUD, etc.), just like we would in real. This ensures our telemetry link settings are correct.
- *Custom scripts:* For automated testing, we can write Python scripts (using ROS 2 Python API or drone APIs) to simulate RC inputs or autonomous missions in SITL, so we can test scenarios (like commanding a mode switch in simulation to see if the software handles it).

Development Pipeline:
1. **Simulation-First Development:** We will implement the ROS 2 nodes and test them in the SITL/Gazebo environment. For example, we verify that when the simulated Pixhawk (SITL) sends attitude data, our MAVROS node correctly publishes an `/imu` topic and our logger records it. We'll simulate driving by perhaps commanding the model's wheels in Gazebo and ensure encoder topics work. This stage helps us develop our ROS 2 packages (telemetry handling, logging, sensor processing) with zero risk.
2. **Integration Testing in Sim:** We gradually integrate more of the system – adding the camera and LiDAR into the simulation (Gazebo can use a simulated depth camera that publishes both image and depth info).

We test our entire data flow **end-to-end in sim**: sensor → ROS 2 → logging/telemetry. If bandwidth issues or software bugs appear, we fix them now. For instance, if we find the rosbag can't keep up with a 30 FPS video feed, we might reduce the rate or resolution. We ensure that in simulation "telemetry topics saturate at expected bandwidth" without dropping data [20].
3. **Static Bench Tests (HIL):** Next, we run the hardware on the bench. The Pixhawk is connected to actual sensors (IMU, GPS antenna – though indoors GPS might not lock, we can see if it at least communicates). The Teensy is connected to one wheel assembly or to a signal generator (to mimic encoder pulses). The Jetson is in the loop running ROS 2. We then **compare the timing**: does a Pixhawk IMU reading (coming via MAVROS) line up with a Teensy encoder tick count at the same time? We might spin a wheel by hand and see if both Pixhawk (detecting maybe slight movement in IMU) and Teensy (counting rotations) show data for that event. This ensures our system's parts are synchronized and integrated.
4. **Incremental Movement Tests:** We then test each mode individually in a controlled manner. For ground mode, we might put the robot on blocks (wheels off ground) and use the RC to spin the wheels, verifying our control signals and that our ROS logging picks up wheel speeds, etc. For flight mode, we do a tethered motor spin test (props off or on a test stand) to ensure the Pixhawk responds to RC and that our telemetry shows the correct attitude changes when we tilt the vehicle by hand.
5. **First Mobility Tests:** Once satisfied, we perform a simple driving test in ground mode (in a lab or hallway) under RC control, log all data. Then a hover test or short controlled flight in an open area, again logging all data. We use the ground station for real-time monitoring to ensure everything is nominal (for example, checking that vibration levels from logs are acceptable, battery voltage under load is okay, etc.).
6. **Data Review and Iteration:** After each test, we analyze the rosbag data. We can plot wheel encoder counts vs. Pixhawk-reported velocity, or compare IMU readings to expected values, etc. Issues discovered (like a sensor saturating, or a delay in telemetry) feed back into improving the system. We also update simulation parameters to better reflect reality (e.g., the actual mass or motor constants measured). This iterative loop continues until the robot performs reliably in both modes under manual control.
7. **Autonomy Feature Integration (Future):** With a solid manual-control foundation, the pipeline extends to developing autonomy features in parallel. For instance, we can start running SLAM algorithms on the logged data to build maps, then test them live in simulation, then live on the robot (with the operator still there as backup). Because our data infrastructure logs everything, we can use real data to develop these algorithms effectively (post-processing logs to test mapping, etc.).

By using this comprehensive simulation and testing pipeline, we reduce risk and ensure that when the robot is actually deployed, its software has been vetted both in virtual scenarios and with real hardware in safe conditions. This pipeline mirrors approaches used in academia and industry where SITL precedes outdoor flight tests [22] [23], giving confidence in software before full deployment.

Telemetry and Data Logging

Real-Time Telemetry and Control Feedback

Real-time telemetry is vital for the operator to make informed decisions and maintain safety. We implement a dual telemetry system: one through the autopilot's MAVLink for critical flight data, and another through ROS 2 (Wi-Fi) for advanced sensor data and debugging.

- **MAVLink Telemetry to GCS:** The Pixhawk flight controller will transmit telemetry via MAVLink to a **Ground Control Station (GCS)** software. Typically, we will use *QGroundControl (QGC)* on a laptop or tablet. QGC provides an HUD, maps, and vehicle status in real-time. Using a 915 MHz telemetry radio

(SiK radio pair) is a common method: one radio on the robot (Pixhawk TELEM1 port) and one on the ground (USB to laptop). They form a transparent serial link. As mentioned, the base design is that "there exists a radio on the drone and a ground station computer... the drone radio connects to the FC via telem1 port using MAVLink... the ground radio and laptop communicate via MAVLink" [12]. We will configure the telemetry stream to include key data at high rates: attitude (e.g. 50 Hz), IMU (maybe 10 Hz for vibration check), position (10 Hz), battery status (1 Hz), RC input and mode status, etc. This will let the operator see things like: roll/pitch angles, altitude, GPS coordinates, airspeed (if flying), and system warnings (e.g. low battery). In ground mode, some of these are less relevant (altitude is 0, etc.), but the telemetry still provides orientation (e.g. if the robot tilts on a slope) and other info. We can also send a custom MAVLink message for wheel speeds or just reuse existing fields (e.g. Pixhawk's "ground speed" might reflect wheel movement if GPS or wheel odometry is fused). The operator's RC transmitter typically also has its own telemetry (like RSSI, or if using something like ArduPilot with a Yaapu telemetry to the transmitter LCD) – but we will rely on the laptop as the main dashboard.

- **ROS 2 Telemetry over Wi-Fi:** The Jetson companion opens up richer telemetry possibilities, especially for future use. In phase 1, we primarily use it for logging, but we can also stream selected data live. For example, we can have a ROS 2 topic for **vehicle pose** that the ground laptop subscribes to, providing a live 3D view in RViz. Or stream the camera feed to the ground for FPV driving (possibly with an overlay of LiDAR points for situational awareness). We will set up a Wi-Fi access point on the ground (or use robot as hotspot) such that the Jetson and the ground PC are on the same network. Then, using ROS 2's DDS discovery, the ground station can directly subscribe to any ROS topic the robot publishes. This effectively mirrors what rosbag would record, but in real-time. We must be mindful of bandwidth: high-rate raw data (like uncompressed video or point clouds) may not reliably transmit over wireless, especially at distance. So we will likely compress video (e.g. H264 at 720p) and perhaps downsample LiDAR (or just not live-stream every point). For critical telemetry though (the kind that MAVLink already covers), we stick to MAVLink since it's very efficient and designed for low bandwidth.

- **User Interface:** On the ground side, the operator will have multiple feedback displays: QGroundControl gives a structured flight data view and mission planning interface (even if missions are not autonomous, QGC is useful for setting waypoints or monitoring sensors like compass health, GPS signal, etc.). Additionally, for ROS 2 data, we can use **rqt** (a GUI tool in ROS for plotting and monitoring topics) or **RViz** for visual sensor data. For instance, we might run an *RViz configuration that shows the camera feed* and a live 2D map of LiDAR points. During early testing, this is helpful for debugging sensors (e.g. making sure the LiDAR is oriented correctly and detecting obstacles). In the field, the operator might not actively monitor RViz due to bandwidth limits, but the capability is there especially when using the robot in Wi-Fi range. Another telemetry option is to use a lightweight custom dashboard (e.g. a Python script or web interface) that subscribes to ROS 2 and displays key numeric readings (similar to QGC but for any custom data). For example, wheel speeds, hinge angle status, or a notification if a sensor goes offline. This could be part of our base station software suite.

- **Telemetry for Debugging and Tuning:** Real-time feedback is not just for the pilot's situational awareness, but also for the developers. For example, viewing real-time plots of IMU readings can help detect vibration issues early. Or monitoring CPU usage and temperature on the Jetson via telemetry ensures our computer is not overheating or overloaded. We can publish a `/diagnostics` topic from the Jetson (ROS has diagnostic_updater libraries) and see this info

remotely. The manual operation phase will likely involve a lot of tuning (PID gains for flight, alignment of wheels, etc.), and having telemetry like motor outputs or attitude error plotted in real-time (via something like *rqt_plot* or QGC's tuning graphs) is invaluable. We will use those tools to fine-tune the controller. For instance, ArduPilot has a telemetry message for attitude control tuning which QGC can graph; we will enable that.

- **Fail-safes & Remote Control:** The RC controller remains the primary control input, and it also serves as an immediate failsafe mechanism (the pilot can cut throttle, etc.). The Pixhawk will be configured with standard failsafes: if RC signal is lost, it will either hold, land, or in ground mode probably come to a stop; if battery is critically low, it might warn or initiate a land sequence. All these events are telemetered to the GCS (and can be logged in ROS too). We will test failsafe behavior in simulation and on the ground to ensure the robot doesn't do something unexpected (like try to RTL when it's not appropriate – since our use-case might not involve a home position for ground driving, we might disable GPS-required failsafes). Instead, a critical battery failsafe could simply stop the robot (in ground mode) or land it immediately (in flight mode). The telemetry would alert the operator in such cases (QGC flashes warnings).

Data Logging and Post-Analysis

Comprehensive data logging is a cornerstone of our project. We will log **all relevant sensor and state data** during each test run so that we can perform detailed post-mortem analysis, validate performance, and build datasets for future autonomy (e.g. for training perception algorithms or mapping environments).

Our logging strategy includes multiple layers:

- **ROS 2 Rosbag2 Logging:** On the Jetson companion, we will run the ROS 2 logging utility `rosbag2` to record topics. We can configure rosbag2 via a YAML file or command-line to select which topics to record and whether to use compression. Given the large amount of data (especially from camera and LiDAR), we will likely record everything to an onboard storage (like a high-endurance SD card or SSD on the Jetson). The topics to log include: camera images, LiDAR scans, wheel encoder counts, hinge angles, IMU data, attitude (quaternion or Euler angles), position (GPS or odometry), RC inputs, and any other telemetry (voltage, current, etc.). Essentially if it's available in ROS, we log it. The concept proposal explicitly mentions **"a PostgreSQL-backed rosbag2 pipeline will archive every trial"** [24]. This suggests that we intend not only to record data in bag files but also to possibly pipe the data into a database for structured storage. Rosbag2 by default uses SQLite as the underlying storage for ROS 2 data. However, there are ways to integrate other storage backends or to post-process bag files into SQL databases. One approach is to use rosbag2's *storage plugin* interface to write directly to a database. Another approach is after the run, convert the bag to CSV or use a script to insert records into a PostgreSQL database (especially for numeric data). We might use a tool or develop a pipeline where after each test run, the bag file is uploaded to a logging server where a PostgreSQL database stores key data (like time-series of vehicle pose, etc.). This would allow running queries across multiple runs (for example, to compare performance metrics or do statistic analysis). Regardless of whether it's real-time to Postgres or a post-process, the data will end up in long-term storage on a PC.

- **Flight Controller Logs:** The Pixhawk itself logs high-rate binary logs on its SD card (often at 100-400 Hz for IMU). These logs (e.g. ArduPilot's .bin or PX4's .ulg files) can be retrieved after flights. They

contain high-resolution data for the flight controller's internals (such as PID outputs, vibration levels, EKF states). We will use these primarily for diagnosing flight performance issues – for example, if the drone is oscillating, the Pixhawk log would show the PID response. While our ROS logs will have attitude at maybe ~50 Hz, the Pixhawk internal log might have sensor data at 1 kHz which can reveal subtle issues. We'll sync these with ROS logs via timestamp (both use system time or GPS time). The Pixhawk logs can be analyzed with tools like *Flight Review* or *Mission Planner's log analyzer*. They complement the ROS logs, which contain more of the "external" data (like video).

- **Data Logging Infrastructure:** To handle the potentially large data volume, we'll ensure the Jetson has sufficient storage and write speed. Raw video and LiDAR can fill memory quickly, so we may use compression in rosbag2. ROS2 supports compressed image messages (compressed JPEG or PNG). We might opt to log images in compressed form to reduce size (with a small CPU overhead to compress). LiDAR point clouds might be compressed using the Zstd compression plugin in rosbag2. We also set rosbag2 to use asynchronous writing to not block sensor threads. The *PostgreSQL pipeline* could mean using a separate thread or node that takes certain topics (like wheel encoder and pose data) and inserts into a database in real-time. There are ROS 2 to database bridges out there (some use timeseries DBs like InfluxDB, or Postgres for relational storage). If time permits, we'll incorporate a direct database logging for key metrics (so we can quickly query them without parsing bags). For example, storing "distance traveled in each mode" or "battery consumption" per run in a database would be useful to accumulate experience over many tests.

- **Time Synchronization:** An important aspect of data logging is time sync. We will synchronize clocks between the Pixhawk, Jetson, and any other devices. The Pixhawk uses its own time (which can be synced to GPS time when locked). ROS 2 uses system time (Jetson's Linux clock). We will run an NTP service or ROS 2 time sync to ensure the Jetson's clock is stable (since Pixhawk logs with its time, but when we get data via MAVROS, it timestamped by Jetson on receipt – however MAVROS also provides the autopilot timestamp, which we can use). If needed, we can use the **timestamp from the Pixhawk** for those data to align with the Pixhawk log. Similarly, the Teensy data can include a timestamp (Teensy could sync its clock via NTP or we reset it at start). In practice, for short runs, a slight offset is manageable, but we aim for millisecond-level sync if possible, especially to compare events like the exact moment of liftoff in all sensors. One method is to use a common time reference like GPS: if the GPS provides a pulse-per-second, Pixhawk and Jetson could both use it to discipline their clocks. We may not need that complexity if we can simply calibrate offsets.

- **Data Analysis:** Once data is logged, we will use tools like **Jupyter notebooks, MATLAB, or Python scripts** to analyze it. For example, plotting the trajectory of the robot in 3D (using wheel odometry for ground and GPS for global reference), analyzing how smooth the mode transition was (did orientation spike?), measuring latency (we can compare a known event across sensors to measure delays). If using a PostgreSQL database, we can run SQL queries to aggregate results of many runs – e.g., average time it takes to transform, or correlation between battery voltage drop and flight time. This helps in system characterization (like knowing how many "hops" you can do on a full charge, etc.). We can also replay the rosbag through ROS 2 to, say, generate a visualization or feed it into an offline SLAM algorithm to see how well it maps.

- **Logging during Simulation:** Our simulation runs will also be logged with rosbag2. This is useful because we can generate "ground truth" data easily in simulation (Gazebo knows the true pose of the robot). By logging sim data, we have a baseline to compare against real logs. For example, we

can log the simulated robot's odometry and the simulated sensor outputs, then later see how the real sensor data differed. It also allows us to develop analysis code even before the real robot is running, by using the simulated data as if it were experimental data.

- **Safety Logs:** We will ensure that critical information is redundantly logged. For example, battery voltage and current will be logged by Pixhawk at low-rate and also by Jetson (perhaps via Pixhawk MAVLink or an ADC on Teensy). If an anomaly occurs (like a sudden reset), having both logs increases the chance we have the data leading up to it. Our logging laptop can also serve as a backup – we might configure the ROS 2 network so that the laptop also subscribes and records important data live (in case the robot crashes and the onboard log is lost). This might be done selectively due to bandwidth (e.g., laptop subscribes only to low-rate status info and not heavy image data).

Overall, the logging and telemetry setup ensures **no data is lost** and that the operator is never "flying blind." In fact, our plan foresees a continuous improvement loop: *log everything, analyze offline, improve or tune, simulate if needed, then test again*. This approach is directly supported by our infrastructure: *"ROS 2 infrastructure will stream … data over Wi-Fi to a logging laptop. A PostgreSQL-backed rosbag2 pipeline will archive every trial."* We will use **Gazebo/PX4 SITL to inject mock data** and ensure the telemetry system can handle it, then do HIL tests (Pixhawk + Teensy) to verify synchronization, *before field trials* [25]. This end-to-end strategy from streaming to archiving gives us high confidence that by the time we operate the real hybrid robot, we'll be prepared to capture and utilize all the information it generates.

Hardware Options for Control: Pixhawk vs. Teensy vs. Jetson Nano

The control system of this robot can be implemented with different hardware configurations. We evaluate three key components – **Pixhawk flight controllers**, **Teensy microcontrollers**, and **Jetson Nano companion computers** – in terms of their roles, advantages, and drawbacks for this project. Rather than mutually exclusive choices, these components are often complementary, but understanding their capabilities helps in assigning tasks optimally.

Pixhawk (Autopilot FCU) *Example: Pixhawk 4 or Cube*	A dedicated flight control unit with onboard IMU, running autopilot firmware (PX4/ ArduPilot). It stabilizes the vehicle in flight and interfaces with RC, motors, and certain sensors (GPS, etc.).	• **Real-time, reliable control:** Designed for fast control loops (up to 1kHz) and sensor fusion, which is essential for stable multirotor flight. • **Built-in sensors:** High-quality IMUs, gyros, accelerometer, compass, barometer on board, plus support for GPS, airspeed, etc. [26] . Reduces need for external sensor boards. • **Mature autopilot software:** PX4 and ArduPilot provide field-tested flight control algorithms, failsafes, and mode management (loiter, alt-hold, etc.) out-of-the-box. We can leverage these rather than writing our own flight controller. [3] • **RC and PWM interfaces:** Pixhawk has ports for RC receivers and many PWM outputs for motors/servos, simplifying wiring to actuators. It can	• **Limited processing for high-level tasks:** Microcontroller (usually STM32) on Pixhawk is excellent for control, but not suitable for heavy computation (image processing, large map building, etc.). It typically runs an RTOS and autopilot firmware, without room for custom user programs beyond small scripts. • **Not easily extensible for new sensors:** Pixhawk can read basic sensors (analog, I2C, CAN), but integrating a high-bandwidth camera or 3D LiDAR directly is not feasible. Those need a companion computer. • **Mode switching complexity:** Autopilot firmware is not inherently designed for a morphing vehicle with dual modes (though VTOL support exists). We might need to customize or at least carefully configure it. It may not control	**Central to flight control and stabilization.** We will use a Pixhawk as the **primary flight controller** to ensure stable quadcopter flight. It will also handle RC input and could manage the transformation servos. In ground mode, Pixhawk either passively monitors or passes through commands. Its autopilot firmware (PX4/ PX4) gives us a strong starting point for flight without developing control from scratch [3] .

Hardware	Description & Role	Pros	Cons	Use in Our Design
		directly drive our 4 motors + 4 servos. • **Community and documentation:** Widely used in drones, so lots of support resources and tools (QGC, Mission Planner, logs analyzers).	ground driving optimally (ArduPilot has separate firmware for Rover vs Copter). • **Size and power:** Pixhawk boards add weight and require a steady power supply. However, newer Pixhawks are quite small and light; power usage is modest (~2–3W).	

PJRC Teensy (Microcontroller) *Example: Teensy 4.1 (600 MHz ARM)*	A high-speed Arduino-compatible microcontroller board. Used for custom sensing and control tasks that require precise timing or direct hardware interfacing. In this project, it can serve as a motor controller for wheels and a data acquisition unit for encoders, current sensors, etc.	• **Real-time control capability:** Can execute loop code at tens of kHz reliably for tasks like reading encoders or generating PWM signals, with jitter in the microsecond range – good for precise motor control (though Pixhawk can also do PWM, Teensy offers more custom control logic freedom). • **Flexible I/O:** Lots of GPIO, analog inputs (for current or potentiometer sensors), hardware timers, and communication interfaces (UART, SPI, I2C). Easy to interface additional sensors (e.g. wheel encoders, distance sensors) that Pixhawk might not support directly or might lack channels for. • **Ease of programming:** Arduino environment or C++ makes it quick to develop firmware. And a Teensy 4.x is	• **Not an autopilot:** Lacks the sophisticated state estimation and control firmware that Pixhawk has. We would have to program any control logic from scratch (for example, if we tried to use Teensy *instead of* Pixhawk for flight, it would be enormous effort to achieve similar stability – not recommended). • **No OS / limited concurrency:** Typically single-thread loop (though interrupts available). Managing many tasks (reading multiple sensors, controlling motors, communication) requires careful programming. We must ensure the Teensy code is robust; otherwise, a bug could hang the microcontroller. • **Memory and compute constraints:** While fast in MHz, it is still an embedded device with limited	**Supports specialized tasks alongside the Pixhawk.** We plan to use a Teensy for **wheel control/ odometry** in ground mode and to interface additional sensors. For example, Teensy will read the wheel **encoder ticks** and **wheel motor ESC signals** to report actual wheel rotation speed (important for logging and future traction control). It might also control the wheel motors when Pixhawk is not actively doing so (acting as an electronic speed controller interface commanded by Jetson or directly by RC in ground mode). During HIL testing, the Teensy provides wheel data to compare with Pixhawk's data [6]. Basically, the Teensy is our custom "glue" for anything the Pixhawk and

Hardware	Description & Role	Pros	Cons	Use in Our Design
		powerful enough to do floating-point math, filtering, etc., if we wanted to implement a custom controller. • **Small and low-cost:** Tiny form factor and lightweight. Consumes very little power. Good to sprinkle on the robot for distributed tasks if needed.	RAM/flash. Complex algorithms (like SLAM or image processing) are out of scope – but that's what the Jetson is for. • **Integration overhead:** Using a Teensy adds another integration step – we need to establish a protocol between Teensy and Jetson (or Pixhawk). Debugging across these can be tricky. However, micro-ROS can mitigate integration issues by standardizing comms.	Jetson can't do alone.

NVIDIA Jetson Nano (Companion Computer) *(4x ARM cores, 128 CUDA cores, runs Linux)*	A single-board computer with GPU acceleration, running a full OS. Used for high-level computation, running ROS 2, and heavy algorithms (vision, mapping). Also serves as the communications hub (Wi-Fi, etc.) and logging unit.	• **High computational power:** Capable of running Linux applications, neural networks (using CUDA), and handling multiple sensor data streams at once. Essential for tasks like image processing from the camera (object detection, classification) or 3D SLAM with LiDAR – tasks far beyond microcontrollers. • **Runs ROS 2 and rich OS services:** Having Ubuntu with ROS means we can tap into a vast array of existing software. For example, integrating OpenCV, TensorRT, or ROS Nav2 stack for path planning. It's the platform for **future autonomy upgrades**, as envisioned [3] . • **Network and interface support:** Has built-in Wi-Fi or can use a dongle for wireless comms, enabling the telemetry to ground over IP. Also supports	• **Not real-time / Non-deterministic:** Runs a non-real-time OS (unless specially configured). This means it cannot be solely trusted for immediate control loops like stabilizing a drone (scheduling delays in Linux could be tens of milliseconds, which is too slow for that). It also means sensor timestamping might be slightly delayed. We mitigate this by offloading real-time tasks to Pixhawk/Teensy. • **Higher power consumption:** A Jetson Nano uses significantly more power (~5–10W when active with peripherals) compared to a Pixhawk or Teensy. This will reduce battery life. Also, it may require a proper power supply (5V 2A typical) and generates heat (needs cooling considerations). • **Larger and heavier:** Although	**Brain of the system for data and upgrades.** In phase 1, the Jetson is used for **real-time telemetry relay and data logging** (rosbagging all the things) [11] . It will run the ROS 2 nodes integrating Pixhawk & Teensy data, and push telemetry to the ground. It may also handle an FPV camera stream. We are effectively using it as a **telemetry and logging server on the robot**, as well as a platform to run simulations (SITL) and tests. Looking forward, the Jetson will enable **autonomous behavior**: e.g., running obstacle avoidance algorithms (leveraging the LiDAR to detect obstacles and plan paths) [27] , or even doing onboard visual-inertial

			Ethernet, USB 3.0 (for fast sensors or data offload), HDMI (for debugging with a monitor). It can log data to large storage (SD/SSD). • **Flexibility in programming:** We can use Python, C++, or any language/ SDK. This makes development faster for high-level logic versus writing embedded C. It's essentially a computer on the robot.	small (credit-card size), once you add a carrier board, heat sink, and peripherals, it's a few tens of grams and occupies space. We have to mount it carefully on the robot with vibration isolation (to protect it and also to not disturb sensors). • **Boot time and reliability:** Unlike microcontrollers which start almost instantly, the Jetson may take ~30 seconds to boot Linux. If the robot is power-cycled, there's a delay before high-level functions come online. We need to ensure the Pixhawk can operate independently during that time if needed (for example, if we needed to take off within seconds of power on – but typically not required). We also should use a UPS or safe shutdown routine to avoid SD card corruption on sudden power loss.	odometry and mapping for navigation. Its GPU can accelerate such tasks (for instance, running a SLAM algorithm or a neural network for object recognition). Thus, Jetson is crucial for meeting the "future autonomous upgrades" requirement. The Jetson communicates with Pixhawk via MAVLink (serial) and with the world via Wi-Fi, acting as the robot's high-level decision unit [28] .

Summary of Comparison: The Pixhawk and Jetson Nano are not either-or; they are best used in tandem, capitalizing on Pixhawk's strength in *real-time control* and Jetson's strength in *high-level computing*. The Teensy is an optional enhancer for low-level interfacing that neither of the others handle well (e.g., reading a wheel encoder with microsecond accuracy or controlling a custom hardware).

Pros of Pixhawk: Reliable flight control, existing stabilized modes (which is a huge plus – as Caltech's M4 researchers would attest, developing a multi-modal robot is much easier if the flying part is handled by a proven controller). Pixhawk essentially ensures our quadcopter mode will work, letting us focus on integration. It also gives us telemetry and logging from the autopilot side (with community support for analysis). It directly supports manual RC control, which is exactly our initial requirement.

Cons of Pixhawk: It can't do everything – notably, it won't process camera or LiDAR data, so without a Jetson we'd have no advanced sensing. Also, customizing it for the transformable aspect might be challenging (the autopilot might "think" it's always a quad; it won't, for example, know about wheel odometry unless we feed it as if it were a sensor like optical flow). We might not rely on autopilot for ground navigation logic at all, using a simpler separate approach for that.

Pros of Teensy: Provides a straightforward way to implement any custom control or sensor handling we need without disturbing the autopilot or the companion computer. For example, if we want to ensure the four wheels maintain the same speed in ground mode (like a simple traction control), a Teensy could read all four encoders and adjust PWM to each motor within a few milliseconds loop – something that might be harder to achieve in Jetson (due to Linux jitter) or in Pixhawk (due to not having encoder inputs). Also, Teensy can serve as a safety fallback – e.g., we could program it that if it detects a certain condition (no heartbeat from Jetson, perhaps), it could stop the motors. But primarily, it's for data collection (since an RC car style odometry isn't built into Pixhawk's EKF, we gather that separately).

Cons of Teensy: It adds development overhead. We need people comfortable with embedded coding and debugging on microcontrollers. Also, it's another point of failure – we must ensure the Teensy's code is robust (watchdog timers, etc.). If it crashes, we might lose some sensor inputs (though the robot could still fly/drive via Pixhawk & RC alone, which is good redundancy). We mitigate risk by keeping the Teensy's role mostly observational/logging and non-critical actuation. For instance, if the Teensy were actively controlling wheel motors and it died, that would be bad (wheels might lock or go uncontrolled). We could instead route RC wheel control through Pixhawk or directly to ESC as a backup path so that even if the Teensy is down, the operator can still drive.

Pros of Jetson: Essential for the "data" side of the data infrastructure. Without it, we wouldn't be able to log the camera or run ROS 2. It's effectively our gateway to all advanced functionality (like how in the UT Austin Drone project they added a Jetson Nano to use depth camera, optical flow, LiDAR for advanced processing [28]). The Jetson also allows offloading some computations from Pixhawk – e.g., state estimation can be augmented by vision (VIO) on Jetson if needed, feeding back into Pixhawk via MAVLink (some projects do that). In future, an autonomous mission manager would run on Jetson and simply send velocity setpoints to Pixhawk. All that is feasible because of a companion computer.

Cons of Jetson: We have to be mindful of power and ensure the battery can support it along with four motors. Also, environmental protection – Jetson is a mini computer, we need to keep it cool and maybe shielded from dust if going outdoors. Boot time we discussed; a solution is to keep the system powered between deployments or accept a waiting period.

Hardware Options Alternative: One might consider an **all-in-one solution** such as the new boards that combine a flight controller and companion (e.g., Holybro Pixhawk CM4 Baseboard which has a Pixhawk and Raspberry Pi CM4 on one PCB [29]). These can simplify wiring and reduce latency between FCU and companion. For our project, using separate modules is fine, but it's good to note integrated options exist. Also, if not using Pixhawk, one could attempt to use just the Jetson + a DAQ board (like an Arduino or a Navio2 HAT) to control the motors directly – however, that's generally not recommended for a high-performance quadcopter because of Linux non-real-time issues. Pixhawk is essentially doing what something like Navio (Pixhawk-on-RPi) would do, but more reliably. So our chosen mix – Pixhawk + Jetson (+ Teensy) – aligns with best practices in UAV design [30], and specifically addresses the unique needs of a hybrid vehicle.

Conclusion and System Integration Diagram

Bringing it all together, our proposed system architecture meets the project requirements by combining the strengths of each component and technology:

- **Manual Control via RC:** The Pixhawk autopilot reads RC inputs to stabilize flight, giving a novice-friendly control in the air. On the ground, manual control is direct to the wheels (through either Pixhawk or Teensy), enabling responsive driving. The operator decides when to switch modes; the system executes the transformation quickly and reliably with feedback.
- **Real-Time Telemetry:** Critical status info (attitude, position, battery) is relayed live to a ground station (QGC) over a MAVLink telemetry link [12], keeping the operator informed. Meanwhile, the companion computer streams richer data over Wi-Fi when available, including video and custom diagnostics, for enhanced situational awareness and debugging.
- **Data Logging:** Every run is recorded using ROS 2 rosbag2 on the Jetson, capturing a synchronized timeline of all sensor readings and state information [31]. This data is stored and can be offloaded to a PostgreSQL database for further analysis or visualization. The thorough logging enables post-mission analysis (e.g., verifying that during a certain maneuver the system behaved as expected) and accelerates development of autonomy features by providing real datasets.
- **Simulation and Testing Pipeline:** Before any risky maneuvers, the team can simulate them in Gazebo with PX4 SITL, refining control parameters in a safe virtual environment [19]. The pipeline progresses through HIL tests (ensuring Pixhawk-Teensy coordination [20]) to real-world trials, thereby minimizing surprises. This approach de-risks the project and ensures that each subsystem (flight control, driving control, sensor integration) is validated step-by-step.
- **Future-Proof Design:** By using ROS 2 and a companion computer, the system is ready for future upgrades such as autonomy. When the time comes to add autonomous navigation, the Jetson can host ROS 2 packages for SLAM, path planning, and computer vision that subscribe to the already-existing sensor topics. The Jetson can then send high-level commands to Pixhawk (e.g. "fly to XYZ" or "drive to waypoint") using MAVROS/MAVLink, effectively adding brains to the brawn. The hardware choices (Pixhawk + Jetson) are commonly used for research platforms aiming for autonomy [28] [5], so our robot will be in an excellent position to leverage community software and algorithms.
- **Reliability and Maintainability:** Each component in the architecture has a well-defined role, which simplifies troubleshooting. For instance, if flight is unstable, we inspect Pixhawk and its PID tuning. If a sensor isn't logging, we check the ROS node or microcontroller. The use of standardized protocols (MAVLink, DDS) and modular software (ROS 2 nodes) makes the system easier to extend or modify (e.g., swap in a better LiDAR or upgrade to a Jetson Orin in the future with minimal software changes).

Finally, we present a high-level architecture diagram for clarity (simplified for overview):

Figure: High-level system architecture with flight controller (Pixhawk) and companion computer (Jetson) [4] [28]. *Sensors (camera, LiDAR, GPS, IMU) feed into the Jetson via ROS 2, while the Pixhawk autopilot handles motor control and stabilization. The Pixhawk and Jetson communicate using MAVLink, and the Jetson relays telemetry to the ground station over Wi-Fi or telemetry radio. Logged data (rosbag2) can be offloaded for analysis.*

In summary, the proposed data infrastructure and development pipeline ensure that our transformable robot will be **operational and testable in the short term (with manual control and robust data capture)**, and **extensible in the long term (for autonomous behavior and complex analysis)**. By combining the real-time control capabilities of a Pixhawk flight controller with the computational muscle of a Jetson running ROS 2, and by validating everything through simulation and logging, we set a strong foundation for the project's success – from driving down hallways to taking off and flying over obstacles, all while recording every bit of the journey for insight and improvement.

Sources:

- Sabree et al., *Nature Communications* (2023) – Multi-Modal Mobility Morphobot (M4) concept [1] [32].
- Concept Proposal – Project description and requirements [8] [2] [3] [25].
- PX4/ArduPilot Documentation – System architecture and companion computer integration [4] [5] [28].
- Drone Estimation Lab (UT Austin) – Real-time data acquisition setup with Pixhawk, Jetson Nano, ROS, and Gazebo [17] [10] [12].
- Project-specific analysis – Hardware trade-offs for Pixhawk vs Teensy vs Jetson in hybrid control.

---

[1] [32] New Bioinspired Robot Flies, Rolls, Walks, and More - www.caltech.edu
https://www.caltech.edu/about/news/new-bioinspired-robot-flies-rolls-walks-and-more

[2] [3] [6] [7] [8] [11] [20] [21] [24] [25] [31] concept_proposal.pdf
file://file_00000000d4a871fdb1b15e69d1cce8b8

[4] [5] [9] [18] [29] Companion Computers | PX4 User Guide (v1.14)
https://docs.px4.io/v1.14/en/companion_computer/

[10] [12] [13] [14] [15] [16] [17] [19] [22] [23] [27] [28] [30] Real-Time Data Acquisition – Drone Estimation Lab
https://sites.utexas.edu/del/real-time-data-acquisition/

[26] PX4 System Architecture | PX4 Guide (main)
https://docs.px4.io/main/en/concept/px4_systems_architecture