ECE 445 Team 75 RailRider

Final Report

James Recera (jrecera2)

Varun Sharma (varuns10)

Zhanshuo Zhang (zz128)

ECE 445, Senior Design Laboratory

University of Illinois Urbana-Champaign

Spring 2026

Abstract

RailRider is a compact single-wheel inspection robot intended for narrow structural paths where conventional ground robots or aerial robots are difficult to use. The system uses one drive wheel for forward motion and pitch control, two reaction flywheels for roll stabilization and yaw control, an ESP32-S3-based embedded controller for real-time sensing and actuation, and a Raspberry Pi interface for higher-level sensing and command generation. The final design direction moved from an earlier single-flywheel concept to a dual-flywheel configuration because the two-flywheel layout provides better roll authority and steering flexibility for a narrow-path inspection platform. This report summarizes the project motivation, system design, modeling and control approach, electronics and printed circuit board design, verification structure, cost sections, and final project considerations.

Contents

1. Introduction

1.1 Problem and Motivation

Many inspection environments are still hard to access with standard robotic platforms, especially when the workspace is narrow, cluttered, or partially confined. In the built environment, routine inspection is needed for structures such as buildings, tunnels, pipelines, and other infrastructure, but traditional inspection often still depends on people physically entering the site, which is slow and increases labor and access difficulty [1]. Aerial robots are useful in many inspection tasks because they can reach locations that are hard for humans to access, but they also face practical limits in indoor or confined spaces, where localization, obstacle avoidance, and operating time become more difficult [1, 2]. Recent work on confined-space and GNSS-denied inspection also shows that drones often need extra sensing, tethering, or support systems just to maintain reliable operation in these environments, which adds complexity to the platform [2, 3]. Because of that, narrow structural areas such as cable trays, beam edges, pipe-rack paths, and other thin inspection routes still motivate a compact ground robot that can move through tight spaces while staying stable on a small contact footprint.

RailRider is our response to this problem: a compact single-wheel inspection robot that can balance on a narrow contact area while carrying sensing hardware for remote inspection. The design began as a simpler single-flywheel concept, but it was later changed to a dual-flywheel system because turning and side-to-side stabilization needed more control authority. In the final concept, the drive wheel handles pitch balance and forward motion, while the two angled flywheels support roll stabilization and yaw control.

1.2 System Function and Block-Level Organization

At a high level, RailRider is divided into mechanical, electronics, embedded control, and sensing/vision subsystems. The mechanical subsystem provides the single-wheel chassis, flywheel mounts, wheel support, and protective structure. The electronics subsystem distributes power and connects the ESP32-S3 controller to the IMU, motor drivers, encoders, Raspberry Pi, and battery-monitoring circuits. The embedded control subsystem reads IMU and encoder feedback, runs the pitch, roll, and yaw control loops, and sends commands to the drive motor and flywheel motors. The Raspberry Pi and sensing subsystem supports higher-level communication, event logging, and future inspection-related vision functions.

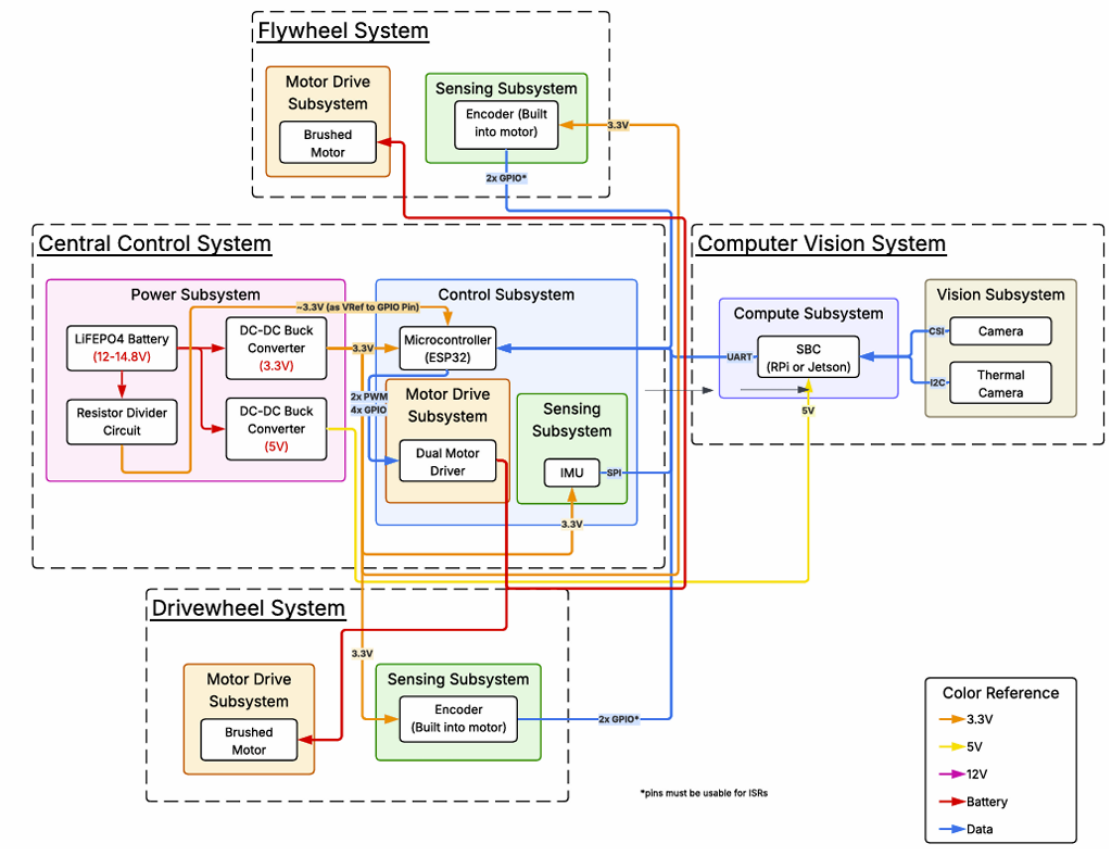

Figure 1: High-level block diagram of each system within RailRider

The main function of the robot is to stay balanced while moving along a narrow path and collecting inspection-related information. The drive wheel gives the robot forward and backward motion, but it also acts as the pitch-control actuator. The two flywheels provide reaction torque for roll and yaw, which is important because the robot has a very small ground contact area. The ESP32-S3 is responsible for the time-sensitive balance loop, while the Raspberry Pi is separated out for higher-level tasks that do not need the same real-time response.

1.3 Performance Requirements

The final system requirements are centered on balance, motion, and inspection behavior. The robot should maintain self-balance for at least 60 s during a stationary balance test, traverse a 2 m narrow path under controlled conditions, stop within 20 cm after a safety command or unsafe attitude trigger, and generate a structured event log containing at least three event types from the sensing or command system. These requirements define the main verification targets for the final prototype.

2. Design

2.1 Design Overview

The RailRider design combines a compact mechanical frame, one driven wheel, two reaction flywheels, an ESP32-S3 control printed circuit board (PCB), an inertial measurement unit, motor interfaces, encoder feedback, and a Raspberry Pi communication interface. The drive wheel is assigned to pitch balance and forward motion. The two flywheels are assigned to lateral balance and steering, using differential flywheel action for roll and common-mode flywheel action for yaw. This architecture keeps the single-wheel footprint needed for narrow inspection paths while adding enough control authority for balance and turning.

The largest block-level change during the semester was the shift from a one-flywheel concept to a two-flywheel concept. The one-flywheel layout was simpler, but it limited steering authority and made lateral stabilization harder. The two-flywheel layout increases the coupling between hardware, control, and mechanical design, but it also gives a clearer way to separate roll stabilization from yaw steering through differential and common-mode flywheel commands. This change affected the physical model, embedded control structure, PCB connector plan, and final integration strategy.

2.2 Mechanical Design

The overall mechanical design and assembly for RailRider had a few key design consideration during the modeling phase of the project. These included ease of manufacturing and assembly, layout of internal motors, distribution of mass, and the specific mechanical interfaces being used to drive each wheel. Version one focused primarily on keeping the mass as low and centered as physically possible while keeping rotating mechanisms as simple as possible. This reduced the chances of potential failure between parts, while keeping the CAD as streamlined as possible to ensure ease of printing. See Appendix **??**, Fig. 5, for a visual reference of the model.

As discussed before, version one of the robot had critical flaws, including a drive system incapable of during in-place due to the wheel footprint, as well as an undersized flywheel. This led to a general redesign to a trapezoidal chassis that instead utilized two flywheels and a rounded drivewheel in order to increase the maneuverability of the robot. However the change in chassis also led to a fundamental change in the layout of components, mechanical systems and safety considerations. Safety guards were added around the flywheels to guarantee the safety of individuals in proximity to the robot in case of total mechanical further. Additionally, the overall size of the robot was shrunk down in order to reduce the total amount of mass and height it would have, thus reducing the minimum requirements needed for each motor to keep the system in equilibrium. This resulted in a might tighter package while still keeping the philosophy of a modular, easy to manufacture chassis in mind for fast iteration should components or parts change within our system.

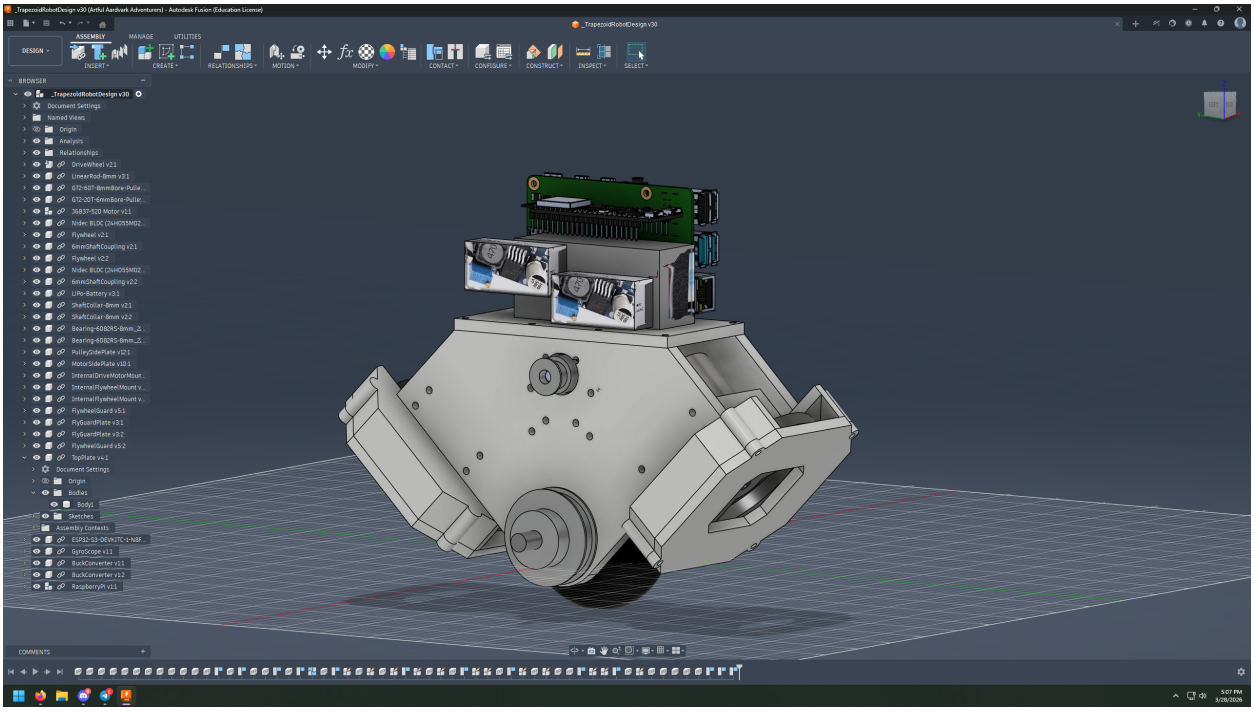

Figure 2: Version two of RailRider's CAD.

By the second iteration, as show above in figure 2 the general packaging used as much volume internally as possible to house each motor and wheel, making the most out of the available space and reducing the size of our robot significantly, while not compromising on modularity, ease of assembly and allowing for rapid manufacturing. It ultimately made for a platform that was easy to iterate on, repair, and swap individual components on if necessary, while being structurally sound.

2.3 System and Physical Modeling

The RailRider platform is modeled as a single-wheel underactuated robot with one drive wheel and two reaction flywheels. The drive wheel mainly affects pitch balance and forward motion, while the two flywheels are used for roll stabilization and yaw steering. Instead of building a full multibody model, the system is separated into pitch, roll, and yaw channels. This reduced model is more useful for the prototype because the actual robot also has friction, backlash, motor saturation, sensor noise, and some mechanical asymmetry.

The main coordinates are pitch angle α, roll angle β, and yaw angle γ. The drive wheel radius is R, and the distance from the wheel axle to the body-flywheel center of mass is r. The body mass is m_b, each flywheel mass is m_f, and the drive wheel mass is m_w. The body and two flywheels are

grouped as

$$M = m_b + 2m_f \tag{1}$$

with body inertias I_{bx}, I_{by}, and I_{bz}, and wheel inertias I_{wx}, I_{wy}, and I_{wz}. The model assumes rigid-body motion, no slip in the main drive direction, and symmetric flywheel placement.

2.3.1 Reduced Pitch, Roll, and Yaw Model

Because the two flywheels are mounted symmetrically at about 45°, their torques can be separated into differential and common-mode components:

$$\tau_{fd} = \frac{\tau_0 - \tau_1}{2}, \qquad \tau_{fc} = \frac{\tau_0 + \tau_1}{2} \tag{2}$$

where τ_0 and τ_1 are the two flywheel reaction torques. The differential component mainly creates roll torque, while the common-mode component mainly creates yaw torque:

$$\tau_{roll} = -\sqrt{2}\tau_{fd}, \qquad \tau_{yaw} = -\sqrt{2}\tau_{fc} \tag{3}$$

This is the main modeling reason for using two flywheels: opposite flywheel action can stabilize roll, while same-direction flywheel action can help steer the robot.

For pitch, the drive wheel is treated as a wheel-driven inverted pendulum. With no slip,

$$x = R\phi, \qquad \dot{x} = R\omega \tag{4}$$

where x is wheel axle displacement, ϕ is wheel angle, and ω is wheel angular speed. The body-flywheel center of mass is located at

$$x_c = x + r\sin\alpha, \qquad z_c = R + r\cos\alpha \tag{5}$$

so drive-wheel torque affects both wheel speed and body pitch. A compact form of the pitch dynamics is

$$\ddot{\alpha} = \frac{-\tau_w - Mgr\sin\alpha - Mr^2\sin\alpha\cos\alpha\,\dot{\alpha}^2}{I_{bx} + Mr^2\sin^2\alpha} \tag{6}$$

$$\dot{\omega} = \frac{\tau_w - RMr\left(\ddot{\alpha}\cos\alpha - \dot{\alpha}^2\sin\alpha\right)}{I_{wx} + m_wR^2 + MR^2} \tag{7}$$

where τ_w is the drive-wheel torque. These equations show the underactuated behavior of the robot, since the same wheel must move the robot and also recover the body pitch angle.

For roll, the differential flywheel torque must oppose the gravity-generated roll moment. The effective roll inertia is

$$J_{\text{roll}} = I_{by} + M(r+R)^2 + I_{wy} + m_wR^2 \tag{8}$$

and the reduced roll dynamics are

$$\ddot{\beta} = \frac{-\sqrt{2}\tau_{fd} + g \sin\beta \left[M(r+R) + m_w R\right]}{J_{\text{roll}}} \tag{9}$$

This shows that roll stability depends on the balance between flywheel correction torque and the gravitational tipping moment.

For yaw, the common-mode flywheel torque is the main steering input. If l_{cf} is the distance from each flywheel center of mass to the combined body-flywheel center of mass, then

$$J_{\text{yaw}} = I_{bz} + I_{wz} + 2m_f l_{cf}^2 \tag{10}$$

and

$$\ddot{\gamma} = \frac{-\sqrt{2}\tau_{fc} + \tau_\mu}{J_{\text{yaw}}} \tag{11}$$

where τ_μ represents ground interaction and yaw resistance.

2.3.2 Control-Related Model Terms

For control, raw wheel speed is not enough because the body also rotates while the robot is balancing. The center-of-mass forward velocity is estimated as

$$V_{\text{COM}} = \omega' R + \dot{\alpha}(R + r\cos\alpha) \tag{12}$$

where ω' is the wheel speed measured relative to the body. This term is used because it combines wheel motion and body pitch motion, which makes it more useful for the pitch-speed control loop than encoder speed alone.

During turning, the robot also needs a lean command so that lateral acceleration is balanced. For desired yaw rate ω_{zp}, the feedforward lean angle is written as

$$\theta_l = k_l \arctan\left(\frac{\omega_{zp} V_{\text{COM}}}{g}\right) \tag{13}$$

where k_l is tuned experimentally. In the roll direction, a gravity-compensation term is also used:

$$\tau_G = k_g g \sin\beta' \frac{M(r+R) + m_w R}{\sqrt{2}} \tag{14}$$

where k_g is a tuning coefficient and β' is the roll error used by the controller.

Overall, this model is not intended to capture every physical detail of the robot. Effects such as wheel slip, motor delay, drivetrain backlash, frame asymmetry, and sensor noise still need experimental tuning. However, the model captures the important structure of RailRider: the drive wheel

controls pitch and forward motion, differential flywheel action controls roll, and common-mode flywheel action supports yaw steering.

2.4 Control System Design

The RailRider controller follows the pitch, roll, and yaw separation used in the physical model. The drive wheel is responsible for pitch balance and forward motion, while the two flywheels provide body reaction torque for roll stabilization and yaw steering. Differential flywheel action is used for roll control, and common-mode flywheel action is used for yaw control [4, 5].

The control system was divided into two stages. Task 1 focuses on stand-still self-balancing, where the robot stabilizes near the upright position before any motion command is added. Task 2 extends the same idea to forward motion, turning, lean-angle control, and flywheel command allocation. The detailed Task 1 and Task 2 control diagrams are included in Appendix B.1, and the expanded Task 2 equations are included in Appendix B.2.

For Task 1, the pitch loop is controlled by the drive wheel. Let α be the pitch angle and let the stand-still reference be $\alpha^* = 0$. A practical pitch control law is

$$u_p = K_{p,\alpha}(\alpha^* - \alpha) - K_{d,\alpha}\dot{\alpha} \tag{15}$$

where u_p is the drive-wheel command. The proportional term gives restoring action, while the derivative term adds damping from the IMU angular-rate measurement.

The roll loop is controlled by the differential flywheel command. Since roll must also resist the gravity moment, the controller combines PD feedback with gravity compensation:

$$\tau_{fd} = K_{p,\beta}(\beta^* - \beta) - K_{d,\beta}\dot{\beta} + \tau_G \tag{16}$$

where $\beta^* = 0$ during stand-still balancing. A scaled compensation term is

$$\tau_G = k_g g \sin\beta' \frac{(m_b + 2m_f)(r + R) + m_w R}{\sqrt{2}} \tag{17}$$

where k_g is tuned experimentally and β' is the roll error used by the controller. This term gives the flywheels extra support against gravity, but the feedback loop still handles the main correction [4].

For Task 2, the same inner balance loops are kept, but outer loops are added for center-of-mass velocity and turning. The pitch loop receives a velocity-based reference, the roll loop tracks a lean-angle command during turns, and the yaw loop uses common-mode flywheel action. Since the flywheels are shared between roll and yaw, the final command allocation gives roll stabilization higher priority when the motors approach saturation.

2.5 Electronics and PCB Design

2.5.1 Overall Electronics Architecture

The RailRider electronics are organized around an ESP32-S3 controller, a Raspberry Pi, an IMU, encoder feedback, and three motor interfaces. The ESP32-S3 runs the real-time balance and motion control loop, while the Raspberry Pi supports higher-level vision and command generation. The IMU provides body attitude feedback, and the encoders provide motor speed feedback so the ESP32 can estimate robot motion and send control commands to the actuators.

The robot uses two 12 V Nidec brushless motors with built-in driver boards as the left and right reaction flywheels for roll stabilization and yaw control. The drive wheel uses a JGB37-520 Hall encoder DC geared motor driven through an external L298N H-bridge module, giving the robot both forward/backward motion and pitch balance authority [8, 9, 10]. Overall, the electronics are divided into sensing, computation, communication, and actuation, matching the final dual-flywheel robot architecture.

2.5.2 PCB Revisions and Final Design

Before the final two-flywheel PCB, an earlier board was designed for the original one-flywheel version of RailRider. This board was mainly a connector and routing board built around the ESP32-S3-DevKitC-1. It brought out IMU, encoder, UART, battery sensing, and motor-driver signals for cleaner early testing. Although it was not used as the final hardware direction, it helped check connector organization, pin routing, and early subsystem integration. The old CAD and previous one-flywheel PCB schematic are included in Appendix A.

The final PCB was redesigned for the two-flywheel system and uses the ESP32-S3-WROOM-1 directly on the board instead of the development board. This revision integrates the MCU, flywheel connectors, drive motor interface, IMU connector, Raspberry Pi UART connector, USB programming port, battery-monitoring inputs, and power circuitry into one compact board. The ESP32 module was placed near the board edge for antenna clearance, and the connectors were arranged around it so each subsystem could route to nearby pins with less trace crossover.

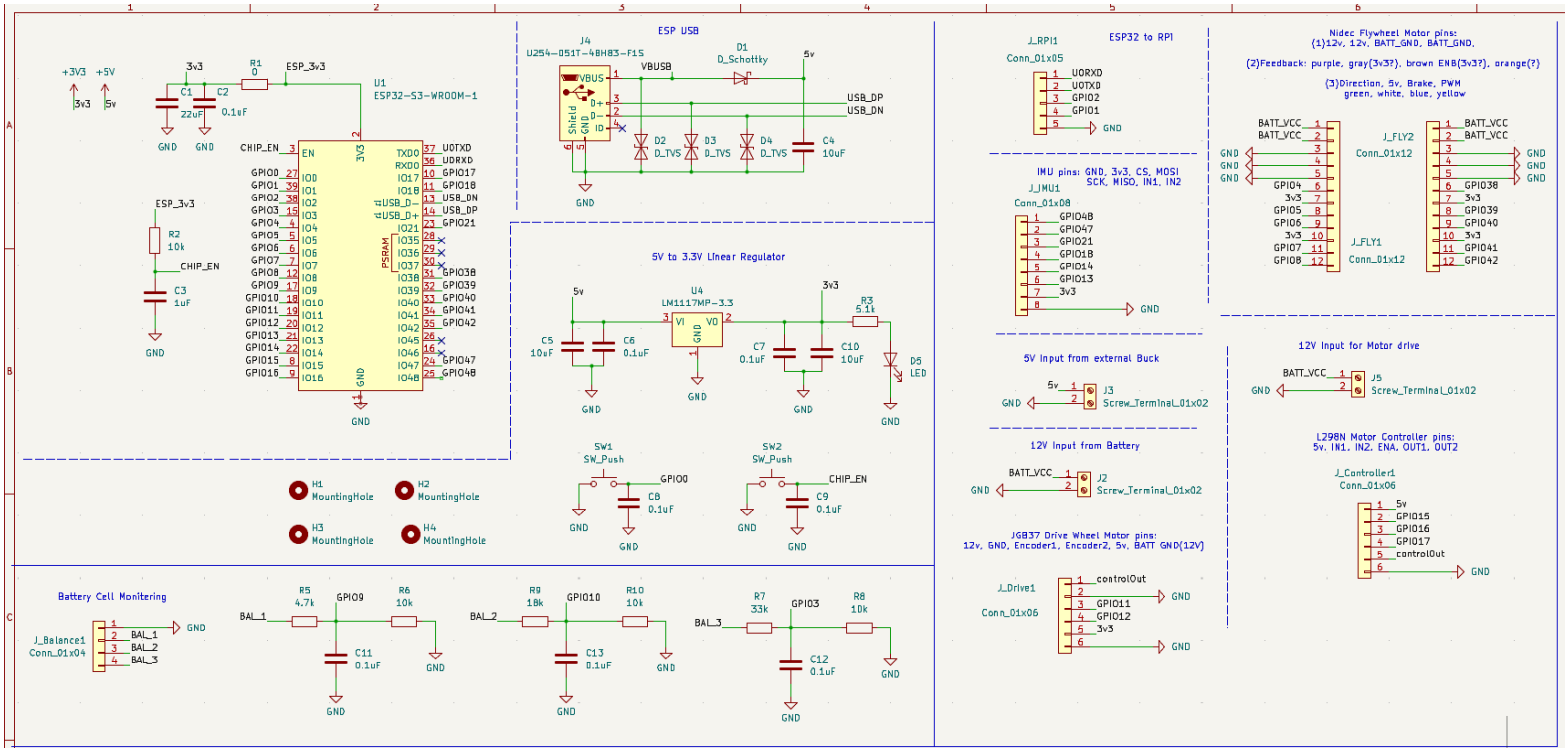

Figure 3: Final two-flywheel PCB schematic.

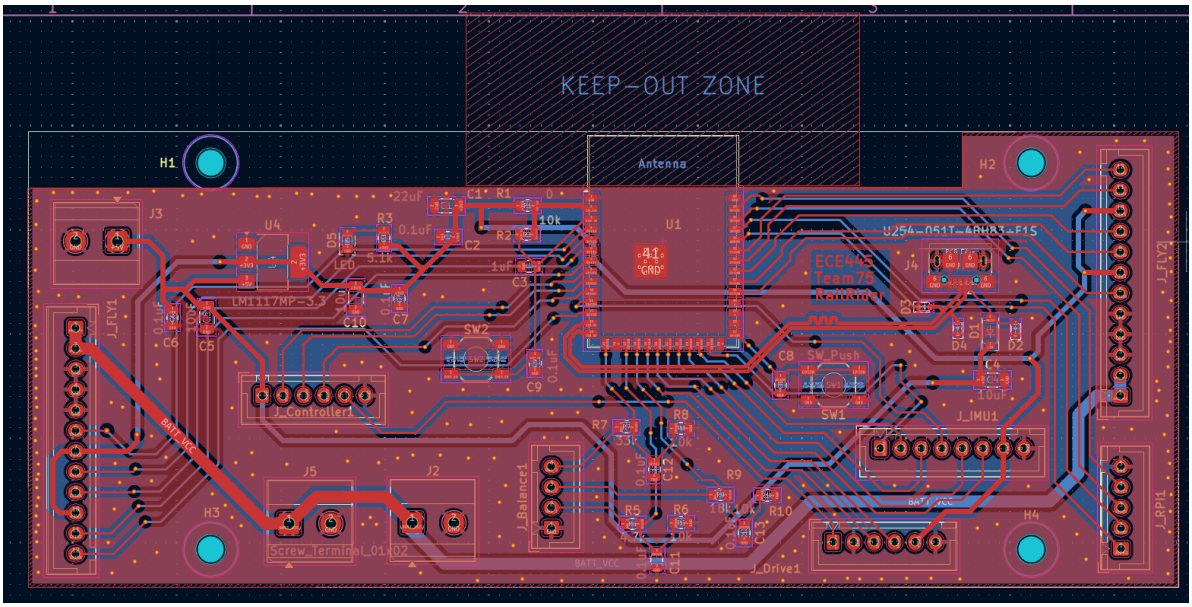

Figure 4: Final front-side PCB layout for the two-flywheel RailRider board. The ESP32-S3-WROOM-1 is placed near the board edge for antenna clearance, with subsystem connectors arranged around it for cleaner routing.

The layout separated high-current power routing from lower-current signal routing. The 12 V battery and motor traces were made wider, about 60 mil, while the 5 V and 3.3 V rails used about 25 mil traces and normal signal traces used about 10 mil. The USB D+ and D− lines were routed as a differential pair to the ESP32 native USB pins. The board also includes a 5 V to 3.3 V regulator, local ESP32 decoupling, BOOT and RESET circuitry, battery-sense resistor dividers, and top and bottom ground pours with stitching vias. Some protection footprints were kept in the design but left as DNP parts for the prototype.

2.6 Embedded Software Design

Software for RailRider is broken up into two main components, the control loop and the vision payload. The control loop is handled by the ESP32 for fast, precise control over our required communication interfaces, and keeps software design simpler by having one microprocessor handle a specific critical robot task. Details regarding the internal mathematical calculations done to determine output motor control speeds can be found in section 2.4, but generally speaking, input sensor readings from the IMU are sent over SPI to the MCU to be processed, after which the resulting next-timestep actions for the motors are sent out over PFM and PWM signals to their respective motors.

2.7 Raspberry Pi and Vision Interface

The Raspberry Pi 4 runs a Flask and Flask-SocketIO webserver that serves a real-time browser dashboard with live IMU telemetry, an MJPEG camera stream, WASD drive controls, obstacle and drop-off alerts, and a timestamped event log. It communicates with the ESP32-S3 over a 3.3V UART link at 115200 baud, forwarding user commands as ASCII strings (`D,vx,vz` for drive, `P,axis,kp,ki,kd` for PID updates) and pushing received sensor telemetry to the browser via WebSocket. The vision system runs as a background thread processing 640×480 frames from the Arducam 12MP camera, executing two detection pipelines per frame: ArUco tag detection using the `DICT_4X4_50` dictionary with bounding-box area-ratio proximity estimation (CLOSE above 10%, MEDIUM above 3%, FAR below 3%), and contour-based edge detection that crops the bottom 30% of each frame as a region of interest, applies Canny edge detection and declares a drop-off when a contour spans more than 30% of the frame width. The contour method replaced an earlier row-density approach after live testing revealed false positives on textured surfaces, where scattered edge pixels satisfied the density threshold without forming a coherent horizontal line. All detection results are annotated directly onto the camera frames before streaming, and emitted to the dashboard as WebSocket events with timestamps.

2.8 Interface Between Subsystems

The mechanical, electronics, and embedded software subsystems are tightly coupled in RailRider. The control system depends on the physical layout and mass distribution of the robot, so the flywheel placement, wheel radius, center-of-mass height, and frame stiffness directly affect the tuning range of the pitch and roll loops. The PCB and electronics subsystem provides the signal and power interface between the ESP32-S3, IMU, motor drivers, encoders, Raspberry Pi, and actuator wiring. The embedded software subsystem then uses those interfaces to read sensor feedback, compute the control commands, and drive the three motors. Because of this structure, the electronics and control design form the bridge between the physical robot and its real-time balancing behavior.

3. Verification

3.1 Verification Overview

In order to guarantee full functionality between all subsystems and components for the RailRider robot, each system much be tested individually and retested throughout the integration process to ensure everything works as intended. The testing process for each part of the robot will be discussed here, noting which tests were able to be verified correctly, as well as where the robot design faltered and did not meet our specific verification requirements.

3.2 Mechanical Verification

A majority of the mechanical components were verified qualitatively, being physically assembled and tested for basic functionality to ensure all parts that are expected to be in motion were moving as expected. The main mechanical quality tested and observed from the

3.3 Electronics and PCB Verification

PCB continuity was tested between corresponding pins using a multimeter to ensure all common traces were connected in their expected locations. This guaranteed that all mapped GPIO pins were at their expected locations for our breakout connections to the motors, as well as ensured that all power supply and ground pins were in their expected locations. Once all traces were verified to be continuous and connected in the proper locations, a series of voltage readings were taken from the output of our buck converter, stepping down 5V to 3.3V. Measurement with a multimeter over a period of 60 seconds showed a stable output of 3.31V, adequate for powering all sensors and the MCU on the PCB, guaranteeing full functionality of our board prior to final assembly.

Motors were verified by stepping their input controls linearly from our MCU and observing the resulting behavior. We expect that for each constant step in input control over PWM/PFM, there should be an equally consistent change in the speed at which the motor is spinning at. Generally speaking, this held true for the drive motor after observing RPM rates via the built in drive motor encoders. However, observation of the flywheel motors revealed that behavior was not as we had expected, sometimes failing to start spinning, rapidly oscillating back and forth, or spinning at a much slower speed than expected for a linear step in the input signal. This revealed to the team that additional troubleshooting and research was required for the Nidec motors we had selected for the project, which ultimately proved to be very problematic after not following the behavior we had desired.

Other electronics were verified qualitatively once wired correctly to the MCU. Functionality of the IMU was verified by decoding all data sent over SPI and verifying it matched any tilting and

rotation of the sensor in the real world. Communication from ESP32 to Raspberry Pi was tested via writing and reading strings at a common baud rate over the wire, and ensuring each device was successful at decoding each others messages. Through unit testing all components were shown to be working, allowing for proper communication and signaling to all motors, sensors and co-processors.

3.4 Control Verification

Control system verification was done through tilt-correction behavior observation. For testing, we would tilt our robot to a specific angle of $\pm 5^{\circ}$, measured by the IMU, either on the roll or pitch of the robot, and monitor the resulting action of the robot to see if it corrected itself back to the upright position. This helped to verify basic behavioral checks to ensure that all reactions aided in bringing the robot back upright, rather than further away from equilibrium. After initial testing, the verification process would have been to log current IMU data for the roll, pitch, and yaw of the data, and after execution of the motor reactions to the robot's tilt, observe how close the robot is to equilibrium, but due to complications with the motors, such observation was unable to be performed. If successful, however, successful verification would have shown that after a period of time, the robot would stabilize around equilibrium with the measured IMU data showing that the robot has successfully balanced itself.

3.5 Vision Verification

In order to test the vision system and verify its overall performance, we use a library of 4x4 50 millimeter ArUco markers in order to determine how accurately detection is taking place. Tags are measured and placed at a distance of 3 feet away from the robot's camera, and the accuracy is determined by the number of total markers used for testing divided by the number of accurately classified markers. All 10 markers were successfully classified at distance, so basic vision is determined to be functional and verified. Edge detection is also verified but qualitatively, only basic preliminary testing being done with table ledges to determine whether or not the vision system was functional under lab conditions.

3.6 Requirement and Verification Summary

Verification of all systems proved to be time consuming but guaranteed that specific functionality was as expected, and working for other subsystems that depended on systems behaving correctly. Ultimately, basic verification of functionality was possible for each system that was present on the robot, but in-depth verification, specifically for the control system, was made impossible due to unexpected behavior observed during verification of the motors. As motors did not behave as expected during the verification process, it led to verification being effectively impossible for the control system, acting as one of many bottlenecks in our testing and design process.

4. Costs

The estimated physical parts cost for the RailRider prototype is about $400 if PCB fabrication and common electronic parts are provided by the ECE department. If not covered, the cost increases by about $50 for JLCPCB ordering and about $50 for Digi-Key parts, giving an estimated parts total of about $500.

Part	**Qty.**	**Cost**
ESP32-S3-WROOM-1	1	$5
LSM6DSO IMU	1	$8
Raspberry Pi 4	1	$139
Arducam 12MP camera	1	$30
Nidec flywheel motors	2	$60
Flywheels	2	$40
JGB37-520 drive motor	1	$17
L298N motor driver	1	$7
LiPo 3S battery	1	$34
Buck converters	2	$10
Miscellaneous mechanical parts	–	$50
Parts subtotal	–	**$400**
Optional PCB and Digi-Key parts	–	$100
Estimated parts total	–	**$400–$500**

Table 1: Estimated parts cost for the RailRider prototype.

Labor is estimated as three team members working about 8 hours per week for a 14 week semester. At $25 per hour with a 2.5 overhead multiplier, the labor cost is about $21,000. Including parts, the total estimated project cost is about $21,400 to $21,500.

5. Schedule

Week of 2/2: Zhanshuo presents initial project idea for approval, team is formed.	**Week of 3/23:** All parts for revised robot designed are ordered by James and Zhanshuo; in-depth research begins on the control system by Zhanshuo and specific work assignments are given to all members for remainder of semester.
Week of 2/9: Initial research is done on other single-wheeled robots. All members start looking into the different requirements our robot will have as well as how we can constrain our design. James starts working through an initial design with parts and initial research for the proposal. Initial high level requirements and subsystems are made as a team.	**Week of 3/30:** James finishes 3D printing of physical robot chassis, begins test fitting components for parts that have arrived and reiterates models to better fit parts as they arrive. As parts arrive, wiring and assembly of breadboard robot is done by all team members.
Week of 2/16: Zhanshuo creates GitLab repository for project and starts initial work on PCB. All members do research on how best to design robot and specific constraints are added to our robot design. Breakout PCB board design is finalized by James and initial parts for doing in-lab testing are also ordered.	**Week of 4/6:** Final PCB design work begins from Zhanshuo, all members are in lab doing testing of now arrived hardware and initial testing of code. One motor is declared defective, while the others are noted for having weird behavior.
Week of 2/23: James works on full design of robot, including BOM and CAD models for printing.	**Week of 4/13:** James troubleshoots flywheel motors independently to try and reverse engineer functionality to work for the robot.

Week of 3/2: All team members work on design document that is presented later that week, which receives feedback from Professor Kim that our wheel design is flawed. Zhanshuo talks with the machine shop regarding manufacturing of some parts for the robot. All team members begin to work on the breadboard demo.	**Week of 4/20:** PCB Ordered, further troubleshooting of motors done in lab done by James and work on control system code done by Zhanshuo and Varun.
Week of 3/9: Work is done for the breadboard demo and all members contribute to creating a rudimentary car setup capable of detecting ArUco tags and driving towards them. All members do initial research on how to revise the robot design to better suit our application and cost constraints.	**Week of 4/27:** PCB arrives, final PCB assembly and wiring done by Zhanshuo. Final integration tasks done with all team members.
Week of 3/16: Zhanshuo works on research and redesign of robot capable of holonomic drive and lower in cost, with parts and subsystems being verified by James. Entire initial design is re-conceptualized into its new trapezoid packaging.	**Week of 5/4:** All team members work on the final report for submission.

6. Conclusions

6.1 Accomplishments

RailRider changed from the original single-flywheel concept into a smaller dual-flywheel single-wheel robot. The final design uses the drive wheel for pitch balance and forward/backward motion, while the two Nidec flywheels are used for roll stabilization and yaw control. The mechanical design also improved from the first larger chassis to a tighter body with better internal packaging and flywheel guards.

The strongest working stage was the breadboard setup with the ESP32-S3 development board. In that version, the IMU, drive motor, flywheel control lines, and Raspberry Pi UART communication were all tested separately. The control code also had both stages built in: Task 1 for standing balance, and Task 2 for drive and turn commands. Before moving to the PCB, Task 1 was close to a usable tuning point, and the Raspberry Pi was able to send commands to the ESP32.

The custom PCB did not give the same result as the breadboard, but it was still useful because it showed where the real integration problems were. Once the system moved to the PCB and JST connectors, the motor behavior became much harder to debug. This made it clear that the next version needs cleaner harness labeling, more test points, and motor feedback that can be verified before the robot is fully assembled.

6.2 Uncertainties

The largest uncertainty was the Nidec flywheel motor interface. The motors could spin, but the documentation was limited, and the middle four pins did not give usable feedback in our tests. Since the flywheel encoder or feedback pins could not be trusted, the controller had to treat the flywheels mostly as command-only actuators. That made roll tuning much harder because we could not directly confirm flywheel speed or acceleration during recovery.

This uncertainty showed up in the balance behavior. During tuning, the robot still struggled to recover from roll errors of about 7 degrees, even after increasing flywheel command authority. Sometimes the motors responded weakly or unevenly instead of producing a stronger correction. The drive motor also had noticeable delay in pitch recovery, so the robot could already be falling farther before the wheel response caught up.

The final prototype did not fully meet the quantitative requirements. The stationary balance target was 60 s, but the PCB-integrated robot could not repeatably hold balance for that long. The 2 m traversal test was not completed because Task 2 needed stable Task 1 balance first. The 20 cm emergency stop requirement was not fully verified because repeatable driving was not reached. The event-log requirement of at least three event types also could not be finished after the Raspberry

Pi failed during final integration.

6.3 Safety and Ethical Considerations

The main ethical issue for this project is safe and honest reporting of the robot's performance. Because RailRider is intended to balance and move on narrow structures, unsafe operation could damage the prototype, the surrounding test setup, or nearby users. Testing should therefore use clear safety boundaries, conservative speed limits, and manual shutdown capability. Results should also be reported according to the actual verified behavior of the prototype rather than the intended behavior of the design. This follows the IEEE Code of Ethics by prioritizing public safety and truthful communication of engineering results.

6.4 Future Work

The biggest thing we would change first is the flywheel motor choice and testing. The Nidec motors were usable for spinning the flywheels, but they were hard to trust for closed-loop control because the documentation was limited and the middle feedback pins did not give useful speed signals in our setup. Before using them again, we would test one motor completely off the robot with an oscilloscope, external tachometer, and simple command sweep. If the feedback still cannot be recovered, it would be better to switch to a motor and driver with a documented encoder or tachometer output.

The next fix is the PCB bring-up process. Our breadboard setup was easier to debug because every signal could be checked by hand, but the final JST harness made wiring mistakes much harder to find. A better process would be to bring up the board in small steps: power first, then ESP32 programming, IMU, one flywheel, the other flywheel, drive motor, encoders, and finally Raspberry Pi UART. Each connector should be labeled and checked with a multimeter before the robot is assembled.

After the hardware is trustworthy again, the control work should return to Task 1 only. The goal should be to make the robot pass the 60 s standing balance test several times in a row before adding any driving command. Task 2 should then be added slowly, starting with low-speed straight driving, then turning, then lean feedforward, and finally Raspberry Pi commands.

Appendix

A. Old Design References

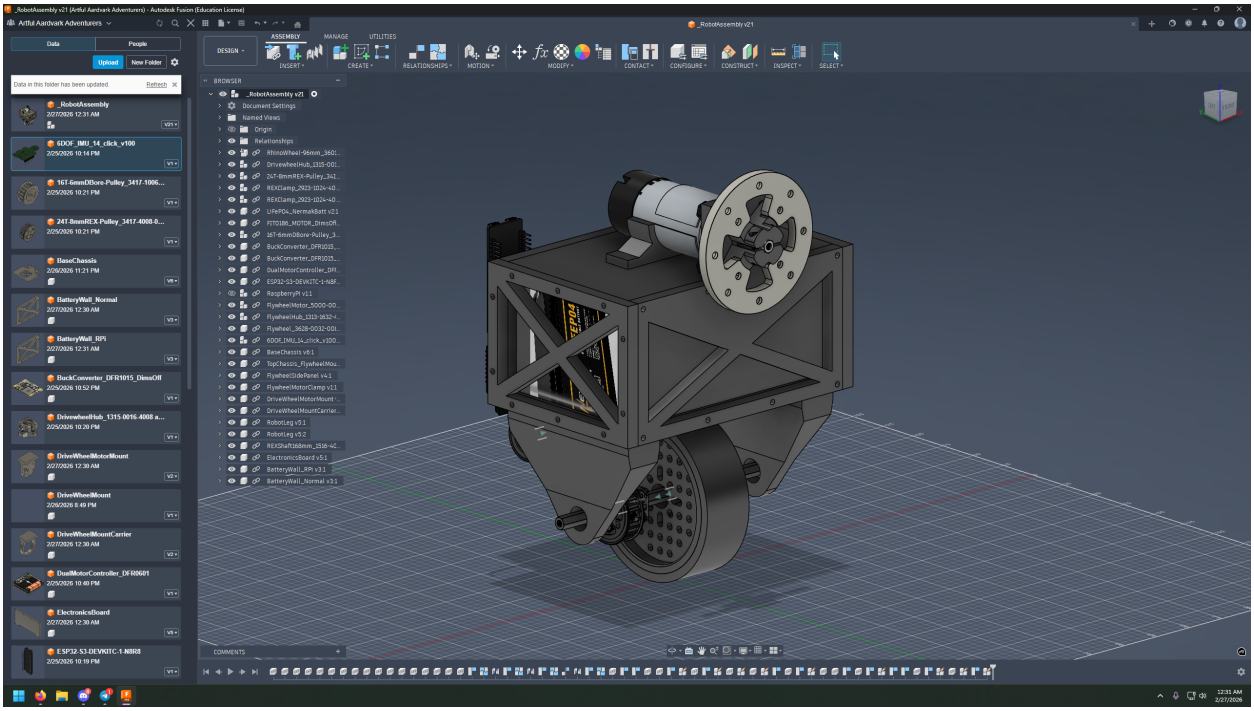

Figure 5: Version one CAD model for the original RailRider design.

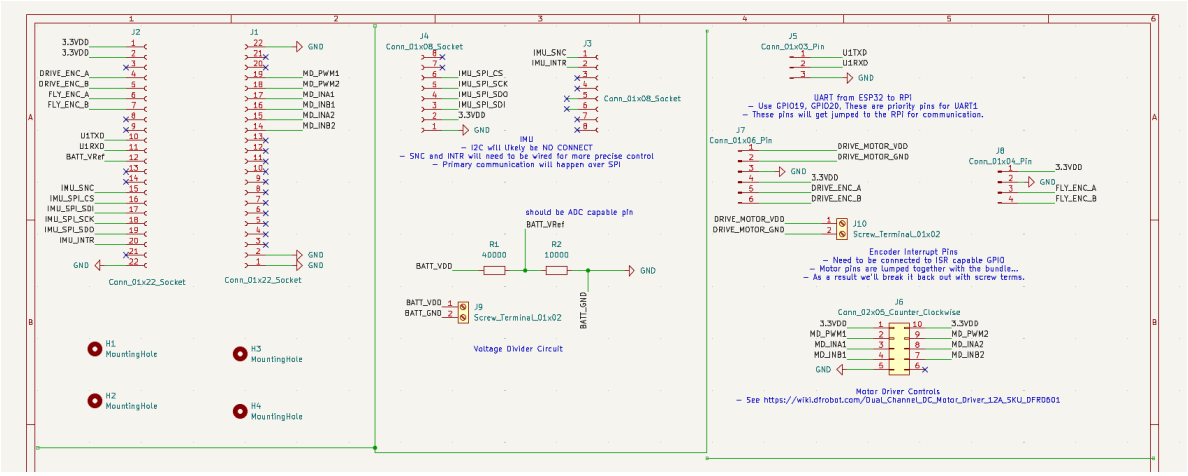

Figure 6: Previous one-flywheel PCB schematic used for early wiring and connector organization.

B. Control Architecture

B.1 Control System Diagrams for our Design

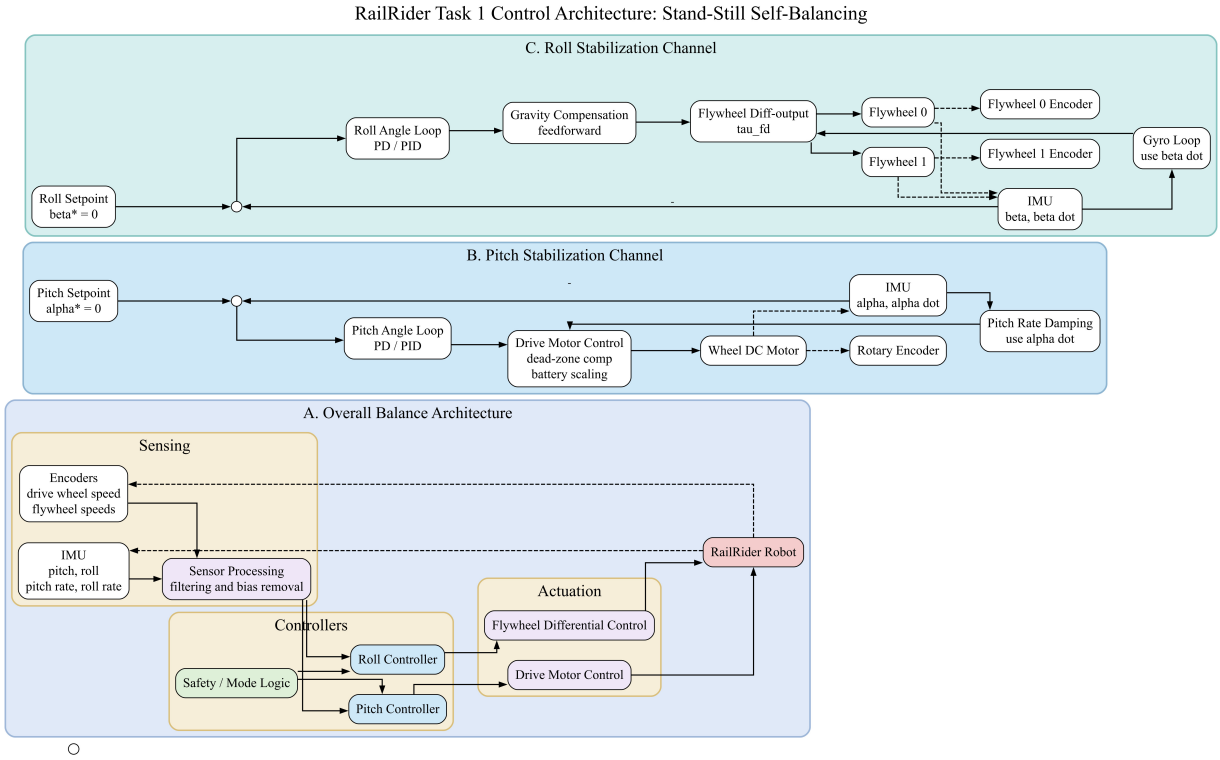

Figure 7: Task 1 control architecture for stand-still self-balancing. The drive wheel regulates pitch, while the flywheel differential command regulates roll.

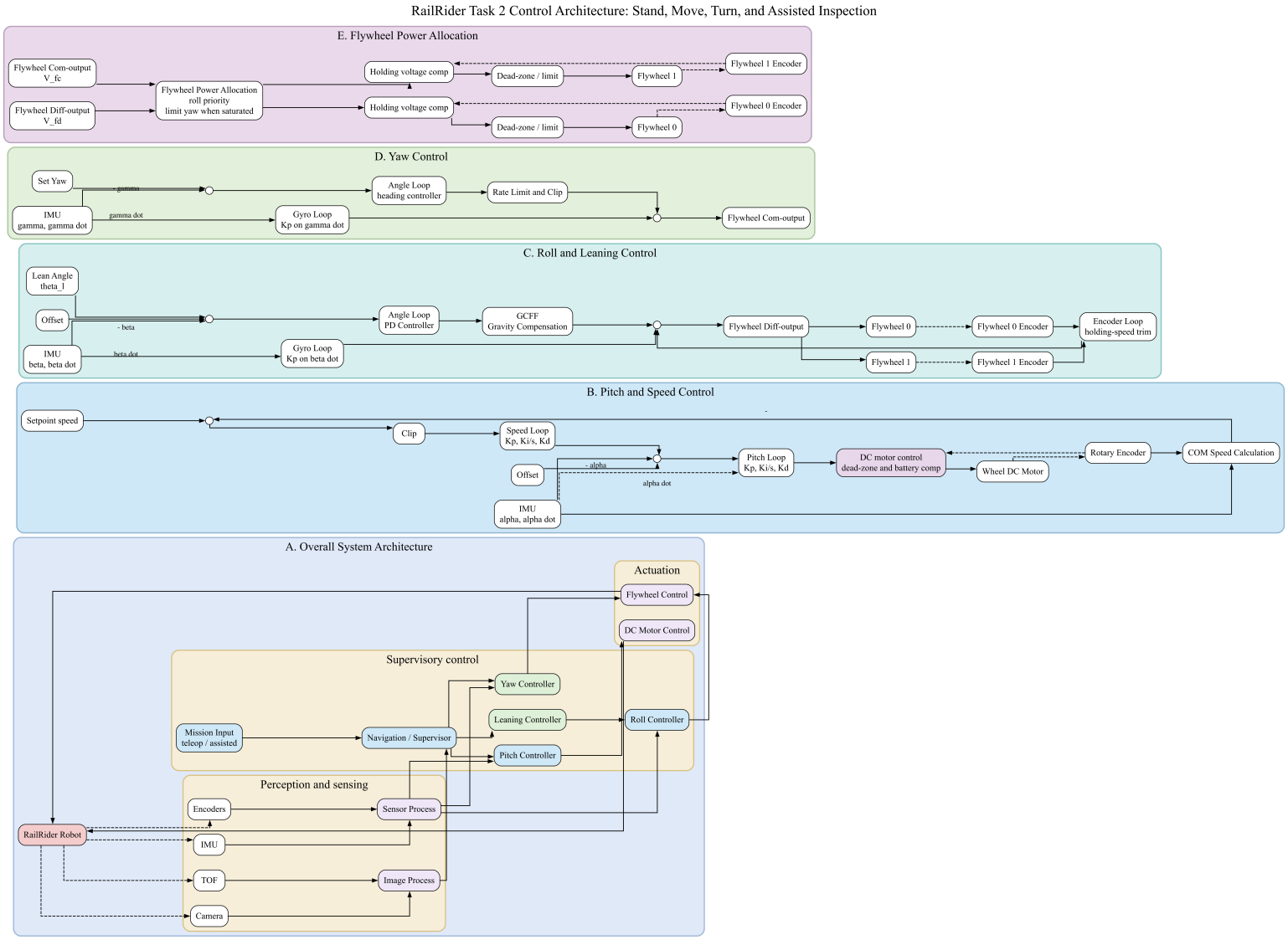

Figure 8: Task 2 full control architecture with pitch-speed control, lean-angle control, yaw control, and flywheel power allocation.

B.2 Task 2 Control Extension

After stand-still balance is achieved, Task 2 adds forward motion and turning through cascaded outer loops. The pitch channel is extended into a speed-attitude cascade, where the outer loop generates a pitch reference from center-of-mass velocity error:

$$\alpha^* = K_{p,v}(V^*_{\mathrm{COM}} - V_{\mathrm{COM}}) + K_{i,v}\int (V^*_{\mathrm{COM}} - V_{\mathrm{COM}})\,dt \tag{18}$$

The velocity term uses the center-of-mass estimate

$$V_{\mathrm{COM}} = \omega' R + \dot{\alpha}(R + r\cos\alpha) \tag{19}$$

For turning, the roll reference is shifted using a lean-angle feedforward command:

$$\theta_l = k_l \arctan\left(\frac{\omega_{zp} V_{\mathrm{COM}}}{g}\right) \tag{20}$$

where ω_{zp} is the target yaw rate and k_l is tuned experimentally. Yaw is added through the common-mode flywheel command:

$$\tau_{fc} = K_{p,\gamma}(\dot{\gamma}^* - \dot{\gamma}) \tag{21}$$

The final flywheel commands are mixed from the common-mode and differential components:

$$u_0 = \tau_{fc} + \tau_{fd}, \qquad u_1 = \tau_{fc} - \tau_{fd} \tag{22}$$

Because the same flywheels are used for both roll and yaw, roll stabilization is given higher priority and yaw is limited when the motors approach saturation [4].

References

[1] S. Halder and Y. J. Kim, "Robots in Inspection and Monitoring of Buildings and Infrastructure: A Systematic Review," *Applied Sciences*, vol. 13, no. 4, p. 2304, 2023.

[2] A. Alsayed, A. Yunusa-Kaltungo, M. K. Quinn, F. Arvin, and M. R. Nabawy, "Drone-Assisted Confined Space Inspection and Stockpile Volume Estimation," *Remote Sensing*, vol. 13, no. 17, p. 3356, 2021.

[3] S. Martínez-Rozas, J. V. López, F. García, and A. J. García, "Long-Duration Inspection of GNSS-Denied Environments with a Tethered UAV-UGV Marsupial System," *Drones*, vol. 9, no. 11, p. 765, 2025.

[4] W. Xi, T. Yin, Z. Liu, J. Wu, D. Xu, and C. Zhang, "Uncertainty-Handling Balance of a Unicycle Robot with Low-Power Flywheels," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E109-A, no. 2, pp. 137–141, Feb. 2026.

[5] S. Chantarachit and M. Parnichkun, "Development and Control of a Unicycle Robot with Double Flywheels," *Mechatronics*, vol. 40, pp. 28–40, 2016.

[6] J. Lee, S. Han, and J. Lee, "Decoupled Dynamic Control for Pitch and Roll Axes of the Unicycle Robot," *IEEE Transactions on Industrial Electronics*, vol. 60, no. 9, pp. 3814–3822, 2013.

[7] S. I. Han and J. M. Lee, "Balancing and Velocity Control of a Unicycle Robot Based on the Dynamic Model," *IEEE Transactions on Industrial Electronics*, vol. 62, no. 1, pp. 405–413, 2015.

[8] LKHFOAQM, "DC 12V Nidec 24H055M020 BLDC Brushless Motor Built-in Drive Board PWM Speed Regulation 100 Line Signal Encoder Feedback Servo Motor," product listing, accessed Apr. 2026.

[9] JGB, "JGB37-520 Hall Encoder Miniature DC Geared Motor 12V Forward And Reverse 530 RPM With Speed Measurement," product listing, accessed Apr. 2026.

[10] STMicroelectronics, "L298 Dual Full-Bridge Driver Datasheet," 2023.

[11] Espressif Systems, "ESP32-S3-DevKitC-1 Schematic," 2022. [Online]. Available: `https://dl.espressif.com/dl/schematics/SCH_ESP32-S3-DevKitC-1_V1.1_20220413.pdf`