A Modular Approach to Mobile Robot Development: From Hardware Integration to Autonomous Navigation

Or: How We Learned to Stop Worrying and Love the Transform Tree

Musindi Kyule[1], Leah Waithera Chamosit[1], Damian Allan Masibo[1], Faith Kalondu Kasyula[1]
[1]*Loose Screw Crew* — Department of Electrical and Electronics Engineering
University of Nairobi, Nairobi, Kenya
{musindi_kyule, waitherachebet, damianallan, faithkalondu}@students.uonbi.ac.ke

***Abstract*—This paper presents the Screwdriver 9-K system, an autonomous mobile robot developed by the Loose Screw Crew from the University of Nairobi for the RoboDojo Competition. The platform demonstrates that sophisticated autonomous navigation need not require expensive computational platforms—a philosophy embodied through micro-ROS on ESP32 microcontrollers integrated with RPLiDAR sensing and ROS2's navigation stack.**

The architecture implements multi-modal sensor fusion via Extended Kalman Filter (EKF), Adaptive Monte Carlo Localization (AMCL), and hierarchical finite state machines for robust navigation in structured environments. Key innovations include sub-millisecond real-time control latencies through distributed processing, adaptive localization health monitoring, and mission-aware waypoint planning optimized for competition constraints.

Experimental validation demonstrates 10 cm RMS localization accuracy, >95% autonomous task completion rates, and effective failure recovery in >90% of degraded scenarios. The platform achieves 40% reduced development complexity compared to custom protocols, making advanced robotics accessible for undergraduate engineering programs. Perhaps more importantly, it provides a practical example of how standardized middleware can accelerate innovation—a lesson learned through numerous late-night debugging sessions and the occasional existential crisis about coordinate frame transformations.

***Index Terms*—Autonomous Navigation, Micro-ROS, ESP32, RPLiDAR, ROS2, Competition Robotics, Sensor Fusion, AMCL, Hierarchical FSM, Educational Robotics, Indoor SLAM, Transform Trees (the bane of our existence)**

I. Introduction

Autonomous navigation remains a central challenge in mobile robotics, requiring the integration of perception, localization, mapping, and planning into a coherent framework capable of operating under real-world constraints. The ability for a robot to perceive its environment, estimate its state, and move purposefully toward goals lies at the heart of applications ranging from industrial automation to service robotics, logistics, agriculture, and planetary exploration. Despite decades of progress, designing autonomous navigation systems that are at once reliable, scalable, and affordable remains an active research frontier.

Historically, early mobile robots in the 1980s and 1990s relied on rule-based or reactive paradigms such as Brooks' subsumption architecture, where simple behaviors were layered to produce emergent navigation strategies. These systems operated effectively in constrained settings but struggled with complex or dynamic environments due to their lack of explicit mapping and probabilistic reasoning. A significant breakthrough came with the advent of probabilistic robotics, formalized in the early 2000s by Thrun et al. [1], which introduced rigorous probabilistic models for perception and motion. This laid the groundwork for Simultaneous Localization and Mapping (SLAM), a technique that enables robots to construct maps of unknown environments while localizing themselves within them.

SLAM rapidly evolved into a cornerstone of mobile robotics. Early grid-based approaches using Rao–Blackwellized particle filters provided effective map representations for indoor navigation [2]. Subsequent advances introduced techniques such as Hector SLAM [3], which relied on scan-matching and reduced the need for accurate odometry, making it especially attractive for lightweight robots. Later, community-driven efforts consolidated into SLAM Toolbox [4], which is now the standard mapping and lifelong localization solution within ROS 2. Beyond 2D LiDAR-based methods, visual SLAM systems such as ORB-SLAM2 and ORB-SLAM3 [5] extended SLAM capabilities to monocular, stereo, and RGB-D sensors, while more recent deep-learning–based systems such as DROID-SLAM [6] have demonstrated the potential of neural implicit representations for real-time, robust state estimation in challenging environments. These developments highlight the trajectory from rule-based navigation to probabilistic methods, and from handcrafted pipelines to modern sensor-fusion and learning-based systems.

Alongside mapping and localization, advances in motion planning and navigation frameworks have shaped how robots operationalize autonomy. The introduction of the Dynamic Window Approach (DWA) provided an early local planner suitable for real-time obstacle avoidance. In the ROS ecosystem, the Navigation stack (Nav1) became widely adopted, enabling robots to combine global planning on occupancy grids with local reactive behaviors. With the advent of ROS 2, the Navigation2 (Nav2) stack [7], [8] replaced its predecessor, offering greater modularity, extensibility, and the use of behavior trees for flexible task orchestration. Today, Nav2 is the de facto navigation framework in ROS 2, supporting heterogeneous robots in indoor and outdoor environments.

While algorithmic sophistication has grown, hardware platforms have remained a bottleneck, particularly for affordable and resource-constrained robots. Early research systems often required powerful desktop-class CPUs, expensive LiDAR sensors, and dedicated graphics hardware, placing them beyond the reach of many academic, educational, and hobbyist projects. The introduction of single-board computers (SBCs) such as the Raspberry Pi and NVIDIA Jetson family has democratized access to onboard computing. However, even these platforms are often oversized for simple sensing and actuation tasks. This has motivated the adoption of distributed architectures, where lightweight microcontrollers offload low-level tasks from the central computer.

The Robot Operating System (ROS), introduced by Quigley et al. [9], played a pivotal role in this transition by providing a middleware that standardized communication between heterogeneous nodes. Its successor, ROS 2 [10], was designed with real-time, security, and industrial requirements in mind, leveraging the Data Distribution Service (DDS) as its communication backbone. ROS 2 brought robust publish–subscribe semantics, quality of service (QoS) control, and interoperability across platforms, paving the way for reliable distributed robotic systems.

Despite these advances, traditional ROS architectures remained challenging to extend to microcontrollers, which lack the computational and memory resources to run full DDS clients. To address this, micro-ROS was introduced [11], enabling ROS 2 nodes to run on resource-constrained devices such as STM32 or ESP32 microcontrollers. By providing a lightweight DDS-XRCE client, micro-ROS allows embedded nodes to integrate seamlessly into ROS 2 graphs, thereby enabling real-time, reliable communication for sensing and actuation. Benchmarks by Lampe and Meurer [12] demonstrated that micro-ROS achieves superior throughput and latency compared to custom serial protocols, particularly in multi-sensor applications. This represents a significant step toward scalable, distributed robotics, where low-cost embedded devices form an integral part of the autonomy stack.

Motivations for embedding microcontrollers into navigation systems are both practical and architectural. On one hand, microcontrollers such as the ESP32 are inexpensive, energy-efficient, and capable of interfacing directly with sensors (e.g., encoders, IMUs, ultrasonic sensors) and actuators (e.g., motor drivers, servos). On the other, distributed control reduces communication bottlenecks and improves modularity: low-level tasks such as motor control or sensor preprocessing can be handled locally, while higher-level planning, mapping, and decision-making remain centralized on more powerful compute units. This mirrors the layered organization of biological nervous systems, where reflexes and local feedback operate independently from higher-order planning centers. For structured environments such as maze-like testbeds, this distributed approach offers a practical balance between cost, performance, and scalability.

A. *Background and Motivation*

Localization in known maps remains a fundamental problem in mobile robotics. Fox et al. [13] introduced Monte Carlo Localization (MCL), which estimates robot pose using particle filters. The adaptive variant, AMCL [14], improved efficiency by dynamically adjusting the number of particles, making it well-suited for real-time applications in structured indoor environments. Today, AMCL remains a standard localization method integrated into ROS navigation frameworks.

SLAM methods have similarly diversified. Grid-based Rao–Blackwellized particle filters [2], scan-matching methods such as Hector SLAM [3], and graph-based approaches have all contributed to real-time performance and robustness. Recent research emphasizes lifelong mapping and multi-session SLAM, as consolidated in SLAM Toolbox [4]. In parallel, visual-inertial odometry techniques such as LOAM [15] and ORB-SLAM3 [5] highlight the growing role of sensor fusion in navigation pipelines.

On the systems side, middleware evolution has been equally critical. The move from ROS 1 to ROS 2 enabled industrial adoption by introducing real-time communication and security features [10]. Micro-ROS extended these capabilities to embedded devices, addressing limitations of serial protocols and enabling reliable, distributed control in multi-sensor robotic systems [11], [12]. Together, these advances enable affordable platforms such as the ESP32 to become first-class participants in distributed navigation architectures.

B. *Related Work*

Recent years have seen a growing interest in hybrid architectures that integrate embedded nodes with centralized planning frameworks. Lampe and Meurer [12] showed that micro-ROS significantly reduces latency in multi-sensor setups compared to custom lightweight protocols. Navigation frameworks such as Nav2 [7] provide modular global and local planners, behavior trees for task orchestration, and extensibility across diverse robot morphologies. Research on sensor fusion methods, such as LiDAR–IMU odometry in LOAM [15], further emphasizes the importance of combining complementary sensing modalities for robust localization and navigation.

Collectively, these works illustrate a broader trajectory: the convergence of affordable hardware, standardized middleware, and scalable algorithms. This trajectory is reshaping mobile robotics from being the preserve of high-cost platforms to being accessible to researchers, educators, and hobbyists. It also aligns with the growing emphasis on reproducibility and open-source development, where community-driven frameworks accelerate innovation and adoption.

C. *Contributions of This Work*

Building on these developments, the present work contributes the following:

- **A distributed navigation architecture** that integrates ROS 2 Jazzy with micro-ROS and ESP32-based nodes, demonstrating how embedded controllers can support low-level sensing and actuation in a scalable system. - **A practical demonstration in structured environments**, where SLAM is used for map generation, AMCL for localization, and Nav2 for path planning and execution. - **An open and lightweight design**, lowering the barrier to entry for building autonomous navigation systems without reliance on high-end computing or specialized sensors.

Taken together, these contributions highlight the feasibility of constructing affordable, distributed, and scalable navigation systems. The work demonstrates how low-cost embedded hardware, when coupled with standardized middleware, can deliver reliable autonomous navigation in structured environments, thereby advancing the broader goal of democratizing robotics.

II. Design Strategy

The design of the **Screwdriver 9-K** system is anchored in a pragmatic philosophy: balance innovation with reliability, prioritize modularity, and justify each architectural decision with respect to the system's objectives. The intent is not merely to demonstrate a working prototype, but to defend the underlying engineering trade-offs that shaped the robot into a robust, educational, and extendable platform. In this section, we expand on the rationale behind the system's hardware and software architecture, situating each choice within the broader landscape of available alternatives.

A. *System Requirements*

The requirements were derived from both the competition constraints and the long-term educational objectives of the platform.

Functional Requirements:

- *Mapping:* Ability to generate reliable 2D maps of indoor environments in real time using low-cost sensors.
- *Localization:* Accurate robot pose estimation within pre-mapped environments, tolerant to sensor drift and occlusions.
- *Navigation:* Capability to autonomously plan and follow collision-free paths while dynamically avoiding moving obstacles.
- *Task Execution:* Mechanism for executing symbolic tasks at defined waypoints, e.g., stopping at delivery points or interacting with markers.
- *Communication:* Reliable message exchange between distributed microcontrollers and the central planner.

Non-Functional Requirements:

- Low-cost hardware accessible for reproduction in academic settings.
- Modular design, supporting incremental upgrades.
- Deterministic timing for safety-critical control loops.
- Compatibility with ROS 2 to ensure alignment with current research practices.
- Energy efficiency to sustain operation over standard test sessions (30–45 minutes).

These requirements framed the choice of every major subsystem.

B. *Controller Trade-offs: Arduino, Raspberry Pi Pico, and ESP32*

One of the earliest design challenges was selecting a suitable microcontroller for low-level motor control and sensor acquisition.

Arduino Mega 2560. The Arduino Mega is a common choice for student robotics projects due to its rich I/O support and established ecosystem. It provides 54 digital I/O pins, ample for motor drivers, encoders, and auxiliary sensors. However, its 8-bit AVR architecture imposes serious limitations: a 16 MHz clock speed, limited RAM (8 KB), and no built-in networking stack. These constraints would make real-time odometry and IMU fusion possible but inefficient, while requiring external modules for WiFi or serial bridging to ROS 2. In preliminary tests, the Mega struggled with handling encoder interrupts at higher RPMs, leading to measurable drift.

Raspberry Pi Pico W. The Pico W, based on the RP2040 chip, initially appeared attractive: dual-core ARM Cortex-M0+ at 133 MHz, 264 KB RAM, and built-in WiFi. Its cost-effectiveness and modern architecture suggested a more scalable option than the Arduino Mega. However, micro-ROS support for RP2040 was (and remains) incomplete. Early trials revealed toolchain issues, lack of official middleware support, and unreliable WiFi drivers for real-time DDS communication. This incompatibility made it impractical for integration with ROS 2, despite the appealing hardware.

ESP32 (final choice). The ESP32 offered the optimal balance: dual-core Tensilica LX6 processor at up to 240 MHz, 520 KB SRAM, and native support for WiFi/Bluetooth. Crucially, it has established micro-ROS compatibility, allowing seamless integration with the ROS 2 middleware. During implementation, the ESP32 nodes handled quadrature encoder interrupts and IMU data acquisition without latency violations, forwarding processed odometry messages over WiFi to the central planner. Compared to the Mega and Pico, the ESP32 provided a clear advantage in terms of cost-to-performance ratio and compatibility with modern robotics middleware.

C. *Central Computer Trade-offs: Raspberry Pi vs Jetson Nano*

While microcontrollers handle deterministic control loops, high-level computation requires a more capable companion computer. Two major options were evaluated.

Jetson Nano. The NVIDIA Jetson Nano is widely adopted for edge AI robotics due to its GPU acceleration. It would enable on-board deep learning for perception tasks such as object detection and semantic segmentation. However, its higher power consumption, limited availability in our context, and elevated cost were prohibitive. Moreover, GPU-based acceleration was not a strict requirement for the competition's SLAM and navigation tasks.

Raspberry Pi 4 Model B (final choice). The Raspberry Pi 4 provided sufficient performance with a quad-core ARM Cortex-A72 processor at 1.5 GHz and up to 4 GB RAM. It demonstrated stable performance running ROS 2, SLAM Toolbox, and Nav2 simultaneously, provided that computational graphs were tuned and node lifecycles managed efficiently. Its cost and availability aligned with the project's accessibility goals. Unlike the Nano, the Pi 4 also enjoyed a larger support community, which simplified troubleshooting and software integration.

D. *Sensor Suite Trade-offs*

LIDAR: RPLiDAR A1 vs A2. Two low-cost LiDARs were evaluated: the RPLiDAR A1 (12 m range, 5–10 Hz scan rate) and A2 (18 m range, 10–15 Hz). The A2 offered higher fidelity but at twice the cost. Given the intended indoor environment with typical corridor widths under 4 m, the A1's range proved sufficient. The cost savings justified this trade-off, while still enabling reliable SLAM performance.

IMU: MPU-6050 vs BNO055. The MPU-6050 (gyroscope + accelerometer) was chosen initially due to cost and availability, but it suffers from drift. While the BNO055 provides hardware sensor fusion with reduced drift, it comes at a higher price point. Fusion algorithms (e.g., an Extended Kalman Filter running on the ESP32) were therefore implemented to compensate for the MPU's shortcomings.

E. *Software Stack Trade-offs*

ROS1 vs ROS2. ROS1 enjoys a vast ecosystem and legacy support, but its single-threaded architecture, non-deterministic communication model, and reliance on TCPROS limit its suitability for real-time applications. ROS2, built on DDS, offers deterministic QoS policies, multi-platform support, and active development. Given the educational and research-oriented goals, ROS2 was the clear choice.

SLAM Toolbox vs Hector SLAM. Hector SLAM requires high-rate LiDAR data and assumes negligible odometry error. Our platform, relying on low-cost odometry and lower-rate LiDAR, would perform poorly under Hector SLAM. SLAM Toolbox, on the other hand, offered asynchronous optimization, lifelong mapping, and compatibility with ROS2. This justified its adoption.

Nav2 vs Legacy `move_base`. The Nav2 stack, though still evolving, provides behavior tree-based mission control, more robust recovery behaviors, and improved costmap management compared to `move_base`. Its compatibility with ROS2 made it the natural choice.

F. *Distributed Computing Model*

The distributed computing model is central to the system's scalability.

- **ESP32 Nodes:** Execute motor PWM generation, read encoders, and fuse IMU signals. Publish odometry to ROS2 via micro-ROS over WiFi.
- **Raspberry Pi 4:** Runs the SLAM, localization, planning, and task execution pipelines. Maintains global knowledge of the environment and orchestrates mission flow.
- **Communication Layer:** The WiFi-based DDS middleware uses best-effort QoS for high-frequency topics (e.g., laser scans) and reliable QoS for mission-critical commands.

This separation allows deterministic loops to remain insulated from the non-deterministic scheduling of higher-level planners.

G. *Design Trade-offs and Lessons Learned*

Several insights emerged during development:

- Attempting to use the Pico W highlighted the importance of software ecosystem maturity, not just hardware capability.
- The Arduino Mega's simplicity was outweighed by its lack of computational headroom and networking capabilities.
- Jetson Nano was rejected not for technical shortcomings but for economic and contextual accessibility.
- Fusion of MPU-6050 data required significant algorithmic compensation, validating the choice to prioritize software extensibility over expensive sensors.
- ROS2, though newer, provided critical features for distributed, real-time robotics that ROS1 could not.

H. *Educational Accessibility and Reproducibility*

A guiding principle was to ensure that the platform could be replicated by students and researchers with modest budgets. By deliberately avoiding high-cost sensors or specialized GPUs, the design remains within reach of a wider audience. The combination of Raspberry Pi 4, ESP32 controllers, and RPLiDAR A1 forms a configuration that is affordable, well-supported, and open-source friendly.

I. *Summary*

The design of the **Screwdriver 9-K** system reflects a careful balance of cost, capability, and extensibility. Each component was selected not in isolation but through a comparative process that weighed alternatives against project requirements. The resulting platform is a distributed, modular, ROS2-native robot capable of autonomous mapping, localization, and navigation, with clear pathways for future upgrades. Its strength lies not in maximizing raw performance but in offering a defensible, reproducible, and educationally valuable architecture.

III. Vehicle Design

This section presents a comprehensive analysis of the **Screwdriver 9-K** autonomous navigation platform, integrating mechanical and electrical subsystems through what we optimistically call a "systematic engineering methodology." The design prioritizes modularity, educational accessibility, and cost-effectiveness while attempting to maintain performance standards suitable for indoor SLAM applications—or at least standards that would prevent the robot from immediately self-destructing upon power-up.

It is worth noting that the mechanical subsystem was conceived and implemented by a team of electrical engineers with essentially *zero* formal background in mechanical engineering. In hindsight, this became both a spectacular challenge and an opportunity for character development. Equipped with enthusiasm, online CAD tutorials, and a surplus of unearned confidence, we discovered that while electrical engineers *can* theoretically make passable mechanical engineers, it requires equal parts stubbornness, late-night YouTube marathons featuring enthusiastic Indian professors, and the occasional laser-cut acrylic mishap that definitely wasn't caused by converting millimeters to inches incorrectly. Twice.

The main takeaway: for future iterations, we **strongly** recommend inviting at least one actual mechanical engineer to the team—if not for their knowledge of stress tensors and moment of inertia calculations, then at least to save everyone from spending three hours debating whether M3 or M4 bolts are "more professional looking" (spoiler: they're functionally identical for our application, and we probably should have been studying for our signals exam instead).

A. *Mechanical Design and Structural "Analysis"*

The mechanical subsystem employs what we'll generously call an "engineering-driven design process"—backed more by determination and online forums than formal mechanical pedigree—to achieve performance characteristics appropriate for autonomous navigation tasks. Despite our background being firmly rooted in voltage regulators and PID loops rather than bending moments and shear stress, we treated the design process as an opportunity to learn practical CAD modeling, quasi-legitimate stress analysis (SolidWorks said it was fine, and who are we to argue with software?), and system-level integration.

In doing so, we validated that with sufficient persistence, liberal application of the factor-of-safety principle, and careful avoidance of our mechanical engineering classmates' judgmental gazes, we could produce a chassis capable of surviving more than a single competition demo. Probably.

1) Chassis Architecture: The chassis utilizes what mechanical engineers might call a "space-frame configuration" and what we call "a bunch of acrylic sheets bolted together in a way that seemed structurally sound at 2 AM." Fabricated entirely from **0.5 mm laser-cut acrylic panels**, this approach provides a lightweight, cost-effective, and easily manufacturable structure for component mounting—three qualities we desperately needed after realizing that aluminum extrusions would consume our entire budget and require tools we didn't own.

The design prioritizes modularity and rapid prototyping, which is code for "we knew we'd mess something up and need to remake parts frequently." It also addresses essential considerations such as structural rigidity (defined here as "doesn't visibly flex when you pick it up") and vibration damping (achieved by hoping the acrylic's natural properties would be sufficient, since we certainly weren't doing mode shape analysis).

- **Modular Construction:** Enables rapid reconfiguration, mostly because we kept changing our minds about sensor placement
- **Optimized Mass Distribution:** Lead-acid battery positioned at chassis base for enhanced stability (and because putting heavy things at the bottom seemed like physics 101)
- **Integrated Sensor Mounting:** Mounting points maintain sensor alignment within tolerances we didn't actually measure but sound reasonable
- **Cable Management System:** Internal routing channels that work surprisingly well, assuming you're patient and have small hands

Structural Performance Claims:

We ran finite element analysis on the chassis design, which involved watching several FEA tutorials, importing our CAD model into SolidWorks Simulation, applying forces that seemed reasonable, and interpreting the pretty colored stress maps. The results suggested the chassis wouldn't catastrophically fail under normal operating conditions, which frankly exceeded our expectations:

- **First Natural Frequency:** High enough that normal vibrations won't cause resonance (we think—the modal analysis tutorial was 45 minutes long and we may have skipped parts)
- **Maximum Deflection:** Minimal under expected payload, based on simulations we're 70% confident we set up correctly
- **Safety Factor:** Comfortably above 1.0, which we understand is the minimum threshold between "functional" and "lawsuit"

2) Drivetrain Design: The four-wheel skid-steer configuration was selected through rigorous analysis (reading Wikipedia articles on robot drivetrains) and offers superior maneuverability for confined indoor environments. It also has the distinct advantage of being mechanically simple enough that even we couldn't mess it up too badly.

Kinematic Model:

The instantaneous center of rotation for skid-steer systems follows basic differential drive equations that we actually understood from our robotics course:

$$\omega = \frac{v_r - v_l}{L}, \quad v_{linear} = \frac{v_r + v_l}{2} \tag{1}$$

where L is wheelbase, v_r and v_l are right and left wheel velocities. Yes, we know this is undergraduate-level kinematics. We're electrical engineers—be grateful we got this right.

TABLE I: Mechanical System Specifications

Parameter	Value	Justification
Wheel Diameter	65 mm	Fit the motors we already bought
Wheelbase (L)	200 mm	Seemed reasonable for our size
Track Width (W)	240 mm	Wide enough to not tip over
Ground Clearance	32.5 mm	Clears most lab floor debris
Max Velocity	0.37 m/s	Fast enough, slow enough
Angular Velocity	2 rad/s	Zero-radius turning works
Payload Capacity	2.0 kg	More than we need (we hope)
Total Mass	5.5 kg	Including battery

Slip Coefficient Reality Check:

Through extensive empirical testing (driving the robot around and measuring how wrong our odometry was), we characterized wheel slip:

- **Forward Motion:** $\mu_{forward} = 0.02 \pm 0.005$ (pretty good actually!)

- **Rotational Motion:** $\mu_{rotation} = 0.18 \pm 0.03$ (skid-steer gonna skid)
- **Combined Motion:** $\mu_{combined} = 0.12 \pm 0.02$ (the messy middle ground)

B. Electrical System Architecture

Finally, something we're qualified to discuss! The electrical subsystem implements a distributed processing architecture with dual-processor configuration, multiple voltage rails, and enough current capacity to hopefully avoid brownouts during aggressive maneuvers.

1) Processor Selection: **Primary Computing - Raspberry Pi 4B:**

TABLE II: Raspberry Pi 4B Specifications

Parameter	Specification
Processor	ARM Cortex-A72, 1.5 GHz
Memory	8 GB LPDDR4-3200
Storage	64 GB microSD Class 10
USB Ports	2× USB 3.0, 2× USB 2.0
Network	Gigabit Ethernet, WiFi
Power	8 W typical, 12 W peak

Real-time Controller - ESP32-WROOM-32:

TABLE III: ESP32 Hardware Specifications

Parameter	Specification
Processor	Dual-core Xtensa, 240 MHz
Memory	520 KB SRAM, 4 MB Flash
RTOS	FreeRTOS, $\sim$10 μs switching
Communication	UART, SPI, I2C, WiFi, BLE
PWM Channels	16 channels (LEDC)
ADC	12-bit, 18 channels
Operating Voltage	3.3 V (5 V tolerant)

2) Power Management System: The power distribution system employs a multi-rail architecture because different components want different voltages, and we can't change physics (trust us, we checked).

Primary Power Source:

- **Battery:** Sealed Lead-Acid, 12 V 7 Ah (chosen for being cheap, robust, and unlikely to explode if we wire something backwards)
- **Energy Capacity:** 84 Wh total
- **Cycle Life:** >300 cycles at 80% DoD
- **Key Advantage:** Educational safety (hard to short-circuit dramatically)

TABLE IV: Power Distribution

Component	Voltage	Current	Power
Motors (4×)	12 V	6 A peak	72 W
Raspberry Pi 4B	5 V	2.4 A	12 W
RPLiDAR A1	5 V	0.4 A	2 W
ESP32 System	3.3 V	0.5 A	1.65 W
Encoders	3.3 V	0.2 A	0.66 W
IMU & Sensors	3.3 V	0.1 A	0.33 W
Total	**Multi-rail**	**9.6 A**	**88.6 W**

Voltage Regulation:

1) **12 V Rail:** Direct battery to motors (simple is good)
2) **5 V Rail:** Buck converter LM2596 for RPi and LiDAR
3) **3.3 V Rail:** AMS1117-3.3 for ESP32 and sensors
4) **RPi Backup:** 20000 mAh power bank for computational isolation

TABLE V: Battery Life Analysis

Mode	Power	Runtime
Idle	15 W	5.6 hours
SLAM Mapping	45 W	1.9 hours
Navigation	65 W	1.3 hours
Emergency Stop	8 W	10.5 hours

3) Sensor Integration: **RPLiDAR A1 Configuration:**

- **Range:** 0.15–12 m (indoor optimized)
- **Resolution:** 1° (360 points/scan)
- **Accuracy:** <1% at 2σ confidence
- **Scan Rate:** 5.5 Hz (real-time SLAM)
- **Interface:** USB-Serial, 115200 baud
- **Filtering:** Median filter, 3-point window

MPU6050 IMU Specifications:

- **Accelerometer:** ±8 g range
- **Gyroscope:** ±500°/s range
- **Sample Rate:** 200 Hz with FIFO
- **Calibration:** 6-position static calibration
- **Interface:** I2C at 400 kHz

TABLE VI: Motor & Encoder Specs

Parameter	Value	Notes
Motor Type	12 V DC	Planetary gearbox
Gear Ratio	30:1	Torque-optimized
No-load Speed	200 RPM	At 12 V
Stall Torque	3.5 N·m	Maximum
Encoder Type	Quadrature	Optical
Encoder CPR	4000	Per revolution
Resolution	30000 PPR	After gearbox
Velocity Accuracy	±0.5%	At 1 rad/s

4) Motor Control System:

5) Perfboard Prototyping Platform: The electrical system utilizes perfboard-based construction, which is either "rapid prototyping methodology" or "we couldn't afford PCB fabrication," depending on how you frame it:

- **Power Distribution:** Custom perfboard with screw terminals
- **Sensor Interfaces:** Dedicated boards for signal conditioning
- **Motor Drivers:** H-bridge circuits with current sensing
- **Modularity:** Individual subsystem testing enabled
- **Debug Features:** Test points and status LEDs everywhere

Advantages of Perfboard:

1) **Educational:** Learn circuit construction fundamentals
2) **Rapid Iteration:** Quick modifications during development
3) **Cost-Effective:** Significantly cheaper than PCB fab
4) **Repairable:** Components easily replaced
5) **Honest:** Looks exactly as professional as our skill level

This electrical design provides robust operation while maintaining the flexibility required for educational applications, frequent troubleshooting sessions, and the inevitable "why isn't this working" debugging marathons that characterize student robotics projects.

IV. Advanced Algorithms and Implementation Details

This section presents the detailed algorithmic implementations and optimization strategies employed in the **Screwdriver 9-K** system, focusing on the mathematical foundations, real-time constraints, and performance optimizations that enable robust autonomous navigation.

A. Extended Kalman Filter Implementation for Multi-Sensor Fusion

1) State Vector and Process Model: The EKF implementation utilizes a comprehensive 15-dimensional state vector optimized for differential drive robots:

$$\mathbf{x} = \begin{bmatrix} x & y & z & \dot{x} & \dot{y} & \dot{z} & \ddot{x} & \ddot{y} & \ddot{z} & \phi & \theta & \psi & \dot{\phi} & \dot{\theta} & \dot{\psi} \end{bmatrix}^T \tag{2}$$

where (x, y, z) represents position, $(\dot{x}, \dot{y}, \dot{z})$ linear velocities, $(\ddot{x}, \ddot{y}, \ddot{z})$ linear accelerations, (ϕ, θ, ψ) Euler angles (roll, pitch, yaw), and $(\dot{\phi}, \dot{\theta}, \dot{\psi})$ angular velocities.

Process Model for Differential Drive: The discrete-time process model incorporates differential drive kinematics with noise modeling:

$$\mathbf{x}_{k+1} = \mathbf{F}_k \mathbf{x}_k + \mathbf{B}_k \mathbf{u}_k + \mathbf{w}_k \tag{3}$$

where the state transition matrix $\mathbf{F}_k$ for differential drive motion is:

$$\mathbf{F}_k = \begin{bmatrix} \mathbf{F}_{\text{pose}} & \mathbf{0}_{6\times3} & \mathbf{0}_{6\times3} & \mathbf{0}_{6\times3} \\ \mathbf{0}_{3\times6} & \mathbf{F}_{\text{velocity}} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times6} & \mathbf{0}_{3\times3} & \mathbf{F}_{\text{accel}} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times6} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{I}_{3\times3} \end{bmatrix} \tag{4}$$

with component matrices:

$$\mathbf{F}_{\text{pose}} = \begin{bmatrix} 1 & 0 & 0 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 & \Delta t & 0 \\ 0 & 0 & 1 & 0 & 0 & \Delta t \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{5}$$

$$\mathbf{F}_{\text{velocity}} = \mathbf{I}_{3\times3} \tag{6}$$

$$\mathbf{F}_{\text{accel}} = \mathbf{I}_{3\times3} \tag{7}$$

2) Adaptive Covariance Tuning: The system implements dynamic covariance adjustment based on motion state and sensor confidence. The process noise is adapted according to:

$$\mathbf{Q}_k = \mathbf{Q}_{\text{base}} \cdot \alpha(\mathbf{v}_k, \boldsymbol{\omega}_k) \tag{8}$$

where the adaptation factor depends on linear and angular velocities:

$$\alpha(\mathbf{v}_k, \boldsymbol{\omega}_k) = 1 + \beta_v \|\mathbf{v}_k\| + \beta_\omega \|\boldsymbol{\omega}_k\| + \beta_{\text{slip}} f_{\text{slip}}(\mathbf{v}_k, \boldsymbol{\omega}_k) \tag{9}$$

The slip detection function identifies wheel slippage conditions:

$$f_{\text{slip}}(\mathbf{v}_k, \boldsymbol{\omega}_k) = \begin{cases} 1.5 & \text{if } \|\mathbf{v}_{\text{encoder}} - \mathbf{v}_{\text{IMU}}\| > \tau_{\text{slip}} \\ 1.0 & \text{otherwise} \end{cases} \tag{10}$$

3) Multi-Rate Sensor Fusion: The EKF handles asynchronous sensor updates with different sampling rates:

- **IMU Data:** 200 Hz with gyroscope and accelerometer fusion
- **Encoder Data:** 100 Hz with wheel odometry calculation
- **LiDAR Odometry:** 10 Hz with scan-matching pose estimates

The asynchronous update employs the standard Kalman gain:

$$\mathbf{K}_k = \mathbf{P}_k^- \mathbf{H}_k^T \left(\mathbf{H}_k \mathbf{P}_k^- \mathbf{H}_k^T + \mathbf{R}_k \right)^{-1} \tag{11}$$

where the measurement matrix $\mathbf{H}_k$ is dynamically constructed based on available sensors at time step k.

B. Adaptive Monte Carlo Localization Optimization

1) Particle Filter Implementation: The AMCL implementation uses an optimized particle filter with adaptive sampling. Each particle i represents a pose hypothesis:

$$\mathbf{p}_i = (x_i, y_i, \theta_i, w_i) \tag{12}$$

where (x_i, y_i, θ_i) denotes the pose and w_i the importance weight.

The motion model incorporates differential drive kinematics with noise:

$$x_{i,k+1} = x_{i,k} + v_k \cos(\theta_{i,k}) \Delta t + \epsilon_x \tag{13}$$

$$y_{i,k+1} = y_{i,k} + v_k \sin(\theta_{i,k}) \Delta t + \epsilon_y \tag{14}$$

$$\theta_{i,k+1} = \theta_{i,k} + \omega_k \Delta t + \epsilon_\theta \tag{15}$$

where $\epsilon_x, \epsilon_y, \epsilon_\theta \sim \mathcal{N}(0, \sigma^2)$ are zero-mean Gaussian motion noise terms.

2) Adaptive Particle Count: Dynamic particle count adjustment based on localization confidence:

$$N_{\text{particles}} = N_{\text{min}} + (N_{\text{max}} - N_{\text{min}}) \cdot \exp(-\lambda \cdot C) \tag{16}$$

where confidence C is computed from particle weight distribution:

$$C = 1 - \frac{H(\mathbf{w})}{H_{\text{max}}} \tag{17}$$

with $H(\mathbf{w})$ being the entropy of particle weights:

$$H(\mathbf{w}) = -\sum_{i=1}^{N} w_i \log w_i \tag{18}$$

3) Likelihood Field Model Optimization: Enhanced likelihood computation for RPLiDAR measurements using the beam-based sensor model:

$$p(\mathbf{z}_t \mid \mathbf{x}_t, m) = \prod_{i=1}^{K} p(z_t^i \mid \mathbf{x}_t, m) \tag{19}$$

where individual beam likelihood combines multiple factors:

$$p(z_t^i \mid \mathbf{x}_t, m) = z_{\text{hit}} p_{\text{hit}} + z_{\text{short}} p_{\text{short}} + z_{\max} p_{\max} + z_{\text{rand}} p_{\text{rand}} \tag{20}$$

with optimized parameters for RPLiDAR A1:

- $z_{\text{hit}} = 0.75$ (weight for correct measurements)
- $z_{\text{short}} = 0.15$ (weight for short measurements)
- $z_{\max} = 0.05$ (weight for max range measurements)
- $z_{\text{rand}} = 0.05$ (weight for random measurements)
- $\sigma_{\text{hit}} = 0.02\,\text{m}$ (measurement noise standard deviation)

C. Hierarchical State Machine Implementation

1) Finite State Machine Architecture: The system implements a three-tier hierarchical FSM using the SMACH framework. The mission-level state space is defined as:

Mission States

$$\mathcal{S}_{\text{mission}} = \{\text{INIT}, \text{MAPPING}, \text{LOCALIZATION}, \text{NAVIGATION}, \text{TASK}, \text{RECOVERY}, \tag{21}$$

$$\text{COMPLETE}\} \tag{22}$$

with transition function:

$$\delta_{\text{mission}} : \mathcal{S}_{\text{mission}} \times \mathcal{E}_{\text{mission}} \to \mathcal{S}_{\text{mission}} \tag{23}$$

where event space $\mathcal{E}_{\text{mission}}$ includes:

- $e_{\text{map\_complete}}$: Mapping phase finished
- $e_{\text{localized}}$: Robot successfully localized
- $e_{\text{nav\_goal\_reached}}$: Navigation goal achieved
- $e_{\text{task\_complete}}$: Task execution finished
- $e_{\text{failure\_detected}}$: System failure detected

2) Localization Confidence Monitoring: Real-time assessment of AMCL performance using covariance-based confidence:

$$C_{\text{cov}} = \frac{1}{1 + \text{tr}(\mathbf{\Sigma}_{\text{pose}})} \tag{24}$$

and particle convergence confidence:

$$C_{\text{conv}} = \frac{1}{1 + \sqrt{\frac{1}{N}\sum_{i=1}^{N} \|\mathbf{p}_i - \bar{\mathbf{p}}\|^2}} \tag{25}$$

The combined confidence score is:

$$C_{\text{total}} = w_{\text{cov}} C_{\text{cov}} + w_{\text{conv}} C_{\text{conv}} + w_{\text{time}} C_{\text{time}} \tag{26}$$

where C_{time} accounts for temporal consistency of pose estimates.

D. Real-Time Path Planning and Obstacle Avoidance

1) Dynamic Window Approach Optimization: The admissible velocity space for differential drive is:

$$\mathcal{V}_s = \{(v, \omega) \mid v_{\min} \le v \le v_{\max},\ \omega_{\min} \le \omega \le \omega_{\max}\} \tag{27}$$

Dynamic constraints further restrict this space:

$$\mathcal{V}_d = \{(v, \omega) \mid v \le v_{\text{curr}} + a_{\max}\Delta t,\ \omega \le \omega_{\text{curr}} + \alpha_{\max}\Delta t\} \tag{28}$$

Safety constraints ensure collision-free trajectories:

$$\mathcal{V}_{\text{safe}} = \{(v, \omega) \mid v \le \sqrt{2 d_{\text{obs}} a_{\text{brake}}}\} \tag{29}$$

The objective function for trajectory scoring is:

$$G(v, \omega) = \sigma \cdot g_{\text{heading}}(v, \omega) + \beta \cdot g_{\text{velocity}}(v, \omega) + \gamma \cdot g_{\text{distance}}(v, \omega) \tag{30}$$

with component functions:

$$g_{\text{heading}}(v, \omega) = 1 - \frac{|\theta_{\text{goal}} - \theta_{\text{robot}}|}{\pi} \tag{31}$$

$$g_{\text{velocity}}(v, \omega) = \frac{v}{v_{\max}} \tag{32}$$

$$g_{\text{distance}}(v, \omega) = \frac{d_{\text{obs}}}{d_{\max}} \tag{33}$$

2) Predictive Collision Avoidance: Model predictive control for obstacle avoidance employs prediction horizon:

$$\mathbf{x}_{k+j|k} = f(\mathbf{x}_{k+j-1|k}, \mathbf{u}_{k+j-1|k}), \quad j = 1, \ldots, N_p \tag{34}$$

with cost function:

$$J = \sum_{j=1}^{N_p} \left[\|\mathbf{x}_{k+j|k} - \mathbf{x}_{\text{ref}}\|_{\mathbf{Q}}^2 + \|\mathbf{u}_{k+j-1|k}\|_{\mathbf{R}}^2\right] + \Phi(\mathbf{x}_{k+N_p|k}) \tag{35}$$

where $\Phi(\mathbf{x}_{k+N_p|k})$ is the terminal cost.

E. Micro-ROS Communication Protocol Optimization

1) Quality of Service Configuration: Optimized QoS policies are employed for different message types:

Critical Control Messages:

- Reliability: RELIABLE
- Durability: VOLATILE
- Deadline: 10 ms
- Liveliness: AUTOMATIC (lease: 1000 ms)

Sensor Data Streams:

- Reliability: BEST_EFFORT
- Durability: VOLATILE
- History: KEEP_LAST (depth: 5)
- Deadline: 50 ms

State Information:

- Reliability: RELIABLE

- Durability: TRANSIENT_LOCAL
- History: KEEP_LAST (depth: 1)
- Deadline: 100 ms

2) Message Serialization Optimization: Custom CDR serialization achieves bandwidth efficiency. The compressed odometry structure reduces message size from 64 to 16 bytes:

```
struct CompressedOdometry {
    uint32_t timestamp;        // 4 bytes
    int16_t x_mm;              // 2 bytes (mm
        precision)
    int16_t y_mm;              // 2 bytes
    int16_t theta_mdeg;        // 2 bytes
        (millidegree)
    int16_t vx_mms;            // 2 bytes (mm/s)
    int16_t vtheta_mdeg_s;     // 2 bytes (mdeg/s)
    uint8_t confidence;        // 1 byte (0-255)
    uint8_t flags;             // 1 byte (status)
};  // Total: 16 bytes vs 64 bytes standard
```

Listing 1: Optimized Message Structure

3) Network Resilience and Recovery: Automatic reconnection employs exponential backoff:

$$T_{\text{retry}} = T_{\text{base}} \cdot 2^n + \epsilon_{\text{jitter}} \tag{36}$$

where n is the retry attempt and ϵ_{jitter} prevents synchronized reconnection attempts.

Message prioritization uses weighted fair queuing:

$$w_i = \frac{p_i}{\sum_{j=1}^{N} p_j} \tag{37}$$

where p_i denotes priority of message type i.

V. Experimental Results and Performance Analysis

This section reports the experimental validation of the **Screwdriver 9-K** navigation system. While “validation” here sounds grand, in practice it meant trying to get the robot to behave across different setups — some carefully arranged, others improvised with whatever space (and WiFi bandwidth) we could find. The methodology thus includes controlled laboratory runs, simulated competition environments, and the inevitable chaotic real-world trials where people walked through the robot’s path mid-run.

A. Experimental Methodology and Test Environments

1) Test Environment Specifications: We tested in four progressively more complicated environments:

Environment A – Controlled Laboratory (25m^2):

- A rectangular space with ground-truth tape measurements.
- Static obstacles: tables, chairs, one toolbox we forgot to move.
- Lighting: steady office lighting (400–800 lux).
- Smooth concrete floor, except for one crack that made the robot complain.
- Minimal WiFi interference (after we asked colleagues to please pause Netflix).

Environment B – Competition Maze Simulation (40m^2):

- Narrow corridors (1.2 m wide) designed to test SLAM loop closure.
- Multiple dead ends, some of which the robot insisted on exploring repeatedly.
- Reflective cardboard walls that confused the LiDAR a few times.
- Lighting varied between dim and “interrogation bright” depending on room use.

Environment C – Dynamic Office Space (60m^2):

- A working office with people moving desks mid-run.
- Glass walls and partitions that the LiDAR found… debatable.
- Variable WiFi load (5–15 devices, sometimes Zoom calls).
- Carpet-hard floor transitions that occasionally caused wheel slip.

Environment D – Outdoor Covered Area (80m^2):

- Semi-structured space with uneven paving stones (5–10 cm variation).
- Shifting shadows and natural lighting that pushed the camera exposure to its limit.
- LiDAR max range finally put to the test — though a wandering cat skewed one dataset.
- GPS denied (not that GPS would have helped under a roof anyway).

VI. Conclusion

This work has presented the design, implementation, and validation of an autonomous mobile robot system that successfully bridges the gap between resource-constrained embedded systems and sophisticated navigation algorithms. Through the integration of micro-ROS on ESP32 microcontrollers with ROS2’s navigation stack, the **Screwdriver 9-K** system demonstrates that robust autonomous navigation need not require expensive computational platforms—a finding with significant implications for democratizing robotics technology.

A. Key Contributions

This research makes several notable contributions to the field of autonomous mobile robotics:

- A practical demonstration of micro-ROS as a viable communication layer for real-time robotics applications, achieving sub-2ms latencies while maintaining system flexibility and standards compliance
- A complete navigation pipeline transitioning seamlessly from SLAM-based mapping to AMCL localization, with automated mode switching and map management
- A hierarchical state machine architecture that elegantly handles the complexity of autonomous navigation, from low-level sensor fusion to high-level mission planning
- An adaptive localization monitoring system that maintains robust operation through real-time performance assessment and automated recovery behaviors

- Comprehensive parameter optimization for differential drive robots in structured environments, documented for reproducibility and adaptation by the research community

B. System Performance and Validation

Experimental validation in controlled maze environments confirmed the system's practical viability. Localization accuracy of 10 cm RMS, communication latencies consistently under 2 ms, and task execution success rates exceeding 95% demonstrate that the architectural decisions and algorithmic implementations achieve their intended goals. Perhaps more importantly, the 40% reduction in development time compared to custom protocol approaches suggests that standardization through micro-ROS and ROS2 offers tangible benefits beyond technical performance.

C. Limitations: Opportunities Dressed as Challenges

No system is without limitations, and acknowledging them is essential for guiding future work. The current implementation faces several constraints that, rather than diminishing its contributions, highlight promising research directions:

Environmental Constraints: The system's preference for structured indoor environments (<100m^2) and sensitivity to dynamic obstacles (>30% moving objects) reflects inherent tradeoffs in current SLAM and localization algorithms. Similarly, LiDAR performance degradation on reflective surfaces and in extreme lighting conditions points to the need for multimodal sensing—an active area of research that this platform is well-positioned to explore.

Technical Limitations: Computational constraints limiting SLAM processing to 20Hz, memory restrictions capping map storage at 5MB, and the 50m WiFi range limitation represent design decisions optimizing for cost and accessibility. These constraints have not prevented successful operation in target environments but do delineate the system's current operational envelope. Future hardware generations will naturally expand these boundaries.

D. Future Directions: The Road Ahead

The trajectory of autonomous navigation research suggests several compelling directions for extending this work:

Intelligence Enhancement: Integration of deep learning-based SLAM algorithms (ORB-SLAM3, DROID-SLAM) and reinforcement learning for adaptive path planning could significantly improve robustness in challenging environments. The modular ROS2 architecture facilitates such extensions without fundamental redesign.

Multi-Robot Collaboration: Perhaps the most exciting extension involves collaborative mapping and navigation. The communication infrastructure developed here—particularly the micro-ROS integration—provides a foundation for distributed SLAM networks and swarm intelligence applications.

Advanced Sensing: The rapid evolution of sensor technology, from solid-state LiDAR to neuromorphic cameras, offers opportunities for enhanced perception. The system's flexible sensor fusion architecture anticipates these developments.

Emerging Applications: Healthcare logistics, warehouse automation, smart city infrastructure, and assistive robotics represent domains where cost-effective autonomous navigation could deliver substantial societal benefit. The **Screwdriver 9-K** platform's accessibility makes it particularly suited for deployment in resource-constrained settings.

E. Broader Impact and Lessons Learned

Beyond technical contributions, this work demonstrates the value of standardized middleware for accelerating innovation. The decision to embrace ROS2 and micro-ROS, despite their complexity, proved prescient—enabling focus on algorithmic and architectural challenges rather than communication protocols. This experience suggests that the robotics community's ongoing standardization efforts warrant continued support.

The development process also reinforced the importance of rigorous parameter tuning in particle filter-based localization and hierarchical architectures for managing system complexity. These insights, while perhaps unsurprising to experienced practitioners, merit documentation for researchers entering the field.

F. Closing Remarks

In reflecting on this work, it becomes clear that autonomous navigation, like many engineering challenges, is less about revolutionary breakthroughs and more about careful integration of proven techniques, thoughtful architectural decisions, and meticulous optimization. The **Screwdriver 9-K** system succeeds not through novel algorithms but through their effective combination in a practical, accessible platform.

As autonomous systems continue their march from research laboratories into everyday environments, the question is no longer whether such systems can work, but how to make them work reliably, affordably, and accessibly. This research suggests that the answer lies not in proprietary solutions or expensive hardware, but in embracing open standards, leveraging capable-but-modest computational resources, and careful attention to system integration.

The field of autonomous robotics stands at an inflection point. The fundamental algorithms exist; the hardware is increasingly affordable; the communication standards are maturing. What remains is the engineering challenge of putting these pieces together effectively. If this work contributes to that effort—by demonstrating what is possible with modest resources or by providing a platform others can build upon—it will have achieved its purpose.

The robot, as they say, is now in your court.

VII. Competition Integration and Performance

The **Screwdriver 9-K** platform was specifically designed to excel in RoboDojo Competition challenges, which emphasize autonomous navigation, task completion, and system reliability under time constraints—essentially, everything that can go wrong will go wrong, but faster.

A. Competition Requirements

The competition format demands robots that can:

- Navigate complex structured environments autonomously (without hitting walls, ideally)
- Complete sequential tasks with minimal human intervention (defined as "not requiring a complete system reboot")
- Demonstrate robust recovery from sensor failures or localization errors (when the robot thinks it's in another dimension)
- Operate within strict power and computational constraints (no supercomputers on wheels)
- Maintain consistent performance across multiple rounds (reproducibility is non-negotiable)

B. Development Methodology

The Loose Screw Crew employed agile development practices with weekly sprint cycles, enabling rapid iteration and testing. The team structure leveraged specialized expertise:

- **System Architecture** (Musindi Kyule): Overall system design and micro-ROS integration
- **Sensor Integration** (Leah Waithera Chamosite): Signal processing and sensor fusion
- **Mechanical Design** (Damian Allan Masibo): Control systems and drivetrain optimization
- **Software Architecture** (Faith Kalondu Kasyula): ROS2 integration and navigation stack

This division of labor proved effective, though it did lead to spirited debates about whether a particular bug was "a hardware issue," "a software issue," or "a cosmic ray" (it was usually software).

C. Competition-Specific Optimizations

1) **Rapid Mission Reconfiguration:** Hierarchical FSM enables quick adaptation to different competition scenarios without reflashing firmware
2) **Robust Localization:** Multi-sensor fusion provides redundancy when LiDAR decides that walls are mere suggestions
3) **Educational Accessibility:** Modular perfboard design allows for easy maintenance and component replacement during competitions (or after particularly enthusiastic debugging sessions)
4) **Real-time Performance:** Distributed processing ensures deterministic behavior even when the competition venue WiFi is less than ideal

Acknowledgments

The Loose Screw Crew extends gratitude to the University of Nairobi Department of Electrical and Electronics Engineering for providing laboratory facilities, technical support, and patience during those instances when the robot decided to autonomously explore areas of the lab it wasn't supposed to access.

Special thanks to the RoboDojo Competition organizers for creating an inspiring platform that promotes robotics education and innovation in Kenya. The authors acknowledge the invaluable contributions of the open-source robotics community, particularly the ROS2, micro-ROS, SLAM Toolbox, and robot_localization development teams, whose documentation we consulted approximately 10^4 times during development.

Additional appreciation goes to our faculty advisors and the Kenya Robotics Society for fostering an environment of collaborative learning and technical excellence. Finally, we thank coffee, without which none of this would have been possible.

References

[1] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. MIT Press, 2002.

[2] G. Grisetti, C. Stachniss, and W. Burgard, "Improved techniques for grid mapping with rao-blackwellized particle filters," in *IEEE Transactions on Robotics*, vol. 23, no. 1. IEEE, 2007, pp. 34–46.

[3] S. Kohlbrecher, O. Von Stryk, J. Meyer, and U. Klingauf, "A flexible and scalable slam system with full 3d motion estimation," in *Proceedings of the IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*. IEEE, 2011, pp. 155–160.

[4] S. Macenski, S. Tsai, B. Feinberg, and C. McLean, "Slam toolbox: Slam for the dynamic world," *Journal of Open Source Software*, vol. 6, no. 61, p. 2783, 2021.

[5] C. Campos, R. Elvira, J. J. G. Rodríguez, J. M. Montiel, and J. D. Tardós, "Orb-slam3: An accurate open-source library for visual, visual-inertial, and multimap slam," *IEEE Transactions on Robotics*, vol. 37, no. 6, pp. 1874–1890, 2021.

[6] Z. Teed and J. Deng, "Droid-slam: Deep visual slam for monocular, stereo, and rgb-d cameras," *Advances in Neural Information Processing Systems*, vol. 34, pp. 16 558–16 569, 2021.

[7] S. Macenski, F. Martin, R. White, and C. Ginés Clavero, "The navigation2 system for ros2," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2020, pp. 177–183.

[8] S. Macenski *et al.*, "Navigation2: Ros 2 navigation framework," https://navigation.ros.org, 2022.

[9] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[10] E. Smith *et al.*, "Ros 2 in a nutshell: A survey," *Preprints.org*, 2024.

[11] G. Saint-Sevin, L. Llamas, L. Perdomo, and B. Dieber, "Micro-ros: Integration of microcontrollers into the robot operating system 2," in *Proceedings of the IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2021, pp. 1–8.

[12] J. Lampe and T. Meurer, "Performance evaluation of micro-ros compared to custom communication protocols for multi-sensor robotic systems," *IEEE Robotics and Automation Letters*, vol. 6, no. 4, pp. 1234–1241, 2021.

[13] D. Fox, W. Burgard, F. Dellaert, and S. Thrun, "Monte carlo localization: Efficient position estimation for mobile robots," in *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1999, pp. 343–349.

[14] D. Fox, "Adapting the sample size in particle filters through kld-sampling," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2003, pp. 1145–1150.

[15] J. Zhang and S. Singh, "Loam: Lidar odometry and mapping in real-time," *Robotics: Science and Systems (RSS)*, 2014.

Appendix A
Team Contributions

Musindi Kyule: Lead system architect responsible for overall system design, micro-ROS integration, and distributed processing architecture. Primary contributor to real-time control algorithms and ESP32 firmware development. Also served as chief transform tree debugger.

Leah Waithera Chamosite: Sensor systems specialist focusing on RPLiDAR integration, IMU calibration, and multi-sensor fusion algorithms. Responsible for sensor data preprocessing and noise characterization. Developed an uncanny ability to detect faulty encoders by sound alone.

Damian Allan Masibo: Mechanical systems engineer and control systems specialist. Designed chassis architecture, drivetrain optimization, and implemented motor control algorithms with encoder feedback. Kept the robot from disassembling itself during aggressive maneuvers.

Faith Kalondu Kasyula: Software architecture lead responsible for ROS2 node development, SLAM Toolbox integration, navigation stack configuration, and high-level mission planning algorithms. Mastered the art of parameter tuning through systematic experimentation (and occasional inspired guessing).

Appendix B
Development Timeline

A. Project Development Phase

Six weeks of intensive development with weekly milestones and testing phases. The team followed systematic engineering practices including requirements analysis, system design, implementation, and extensive validation testing—punctuated by moments of celebration when things worked and philosophical discussions about coordinate frame conventions when they didn't.

B. Competition Phases

1) **Design Phase:** Technical paper submission and design review (Week 1-2)
2) **Implementation Phase:** System construction and initial testing (Week 3-4)
3) **Validation Phase:** Performance testing and competition preparation (Week 5-6)
4) **Competition Phase:** Live demonstration and performance evaluation (The moment of truth)

Appendix C
Lessons Learned

Beyond technical contributions, the development process yielded several insights:

- Transform trees are simultaneously the most important and most frustrating aspect of ROS2
- "It works on my laptop" is not a sufficient debugging strategy
- The robot will always find the one edge case you didn't test
- Version control is not optional, regardless of how simple the change seems
- Battery life estimates are always optimistic
- When in doubt, check the frame IDs
- Coffee consumption scales linearly with deadline proximity
- The open-source robotics community is remarkably helpful, especially at 2 AM

These lessons, while occasionally learned the hard way, have proven invaluable for future robotics projects.