AutoPort: An Indoor Autonomous Delivery Robot

A Project Report

Authors: Rayon Biswas, Kushaan Gada, Hrishi Pandey
Mentors: Kartikey Pathak, Vishal Mutha

Institution: Veermata Jijabai Technological Institute

Date: September 2025

"Robots are no longer science fiction — they are becoming everyday helpers."

Acknowledgement

We would like to take this opportunity to sincerely thank all those who have contributed to the success of this project.

First and foremost, we are deeply grateful to our mentors, **Kartikey Pathak** and **Vishal Mutha**, for their persistent guidance, constructive feedback, and constant encouragement. Their insights not only helped us overcome technical challenges but also inspired us to approach robotics research with curiosity and discipline.

We also extend our gratitude to our peers who assisted in hardware assembly, debugging, and testing. Their teamwork and ideas enriched our progress and helped us stay motivated during demanding phases of development.

We acknowledge the support of our department and lab staff, who provided us with resources, workspace, and technical help when needed. Their assistance ensured that we could focus on design, implementation, and experimentation without unnecessary delays.

Our project greatly benefited from the global open-source robotics community. The availability of tools such as **ROS2**, **Nav2**, and **slam_toolbox** enabled us to build upon existing frameworks and accelerate our learning. We remain grateful to the countless contributors who share their expertise and code with the wider community.

Lastly, we extend thanks to our families and friends for their encouragement, patience, and support throughout this journey. Their belief in us was a source of strength during long nights of coding, assembly, and testing.

To all who directly or indirectly supported us, we express our sincere appreciation. This work is a reflection of not only our effort but also the collective guidance and support we received along the way.

Rayon Biswas, Kushaan Gada, Hrishi Pandey

Abstract

AutoPort is an indoor autonomous delivery robot designed as a low-cost, modular platform for mapping, localization, and autonomous point-to-point navigation. It uses a Raspberry Pi 5 as the main computer, an ESP32 for low-level encoder interfacing, an RP LiDAR for perception, and ROS2-based software (SLAM Toolbox + Nav2) for autonomy. This report documents design choices, a step-by-step implementation guide, an extensive troubleshooting saga with fixes, and future directions.

Contents

Acknowledgement **1**

Abstract **2**

Contents **3**

List of Tables 5

1 Introduction **6**

2 Hardware Stack **8**

2.1 Mechanical Design and Chassis 8
2.2 Drive System 9
2.3 Perception 9
2.4 Computation and Low-Level Control 10
2.5 Power and Safety 10
2.6 Wiring and Connectors (practical tips) 10
2.7 Hardware Specification Table 10

3 Software Stack **12**

3.1 High-level Architecture 12
3.2 Why these choices? 12
3.3 Topics, Messages, and TF frames (practical) 13
3.4 Detailed Software Flowchart 14
3.5 Practical advice on parameter selection 15

4 Implementation — Step-by-step (complete and detailed) **16**

4.1 1. Preparation and safety 16
4.2 2. Install OS and basic tools (Raspberry Pi 5) 16
4.3 3. Install ROS2 (recommended stable release) 16
4.4 4. Create ROS2 workspace and initial packages 17
4.5 5. Low-level firmware (ESP32) — encoder reading and motor PWM . . . 17
4.6 6. Sensor drivers 17
4.7 7. URDF / XACRO 18
4.8 8. ros2_control integration 18
4.9 9. SLAM (slam_toolbox) — mapping 18
4.10 10. Map saving and map server 18
4.11 11. Nav2 bringup and tuning 18
4.12 12. Typical launch order (practical) 19
4.13 13. Logging and bagging 19

4.14 14. Example commands and checks 19

5 Troubleshooting — Comprehensive Guide 20
5.1 General debugging workflow (recommended) 20
5.2 Common problems, root cause fixes (big table) 20
5.3 Advanced debugging tips 21
5.4 Hardware-specific pitfalls and fixes 21

6 Results and Discussion 23
6.1 Computation and Resource Usage 24
6.2 Navigation Results 24
6.3 Discussion 24

7 Future Work 25
7.1 Perception and Sensing 25
7.2 Software and Autonomy 25
7.3 Robustness and Production 25

A Appendix — Config snippets, examples, and pin mapping 26
A.1 Sample ros2_control YAML (example) 26
A.2 Sample slam_toolbox launch (example) 26
A.3 Sample Nav2 launch snippet 26
A.4 Pin mapping example (Raspberry Pi 5 -¿ ESP32 / Motor Driver) 27
A.5 Sample URDF snippet (XACRO-friendly) 27
A.6 Helpful CLI commands (ROS2) 27

B Glossary 29

C References 31

List of Tables

2.1 Hardware components and suggested specifications 10

3.1 Important ROS2 topics and frames 13

6.1 Computation usage 24

A.1 Example pin/interface mapping (adjust to your wiring) 27

Chapter 1

Introduction

Robotic systems that once belonged to science fiction are now practical tools in hospitals, labs, and offices. AutoPort is an **indoor autonomous delivery robot** designed to carry small payloads across corridors, between labs, or within office spaces. The goal of AutoPort is not only to produce a working robot but to create a clear, reproducible guide a novice can follow.

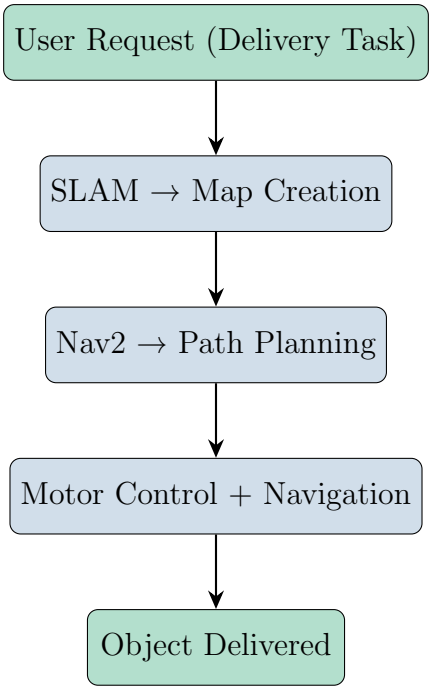
Motivation

Small repetitive delivery tasks waste human time and attention. AutoPort aims to automate such tasks reliably, reducing interruptions and enabling people to focus on higher-value work. The robot also serves as a learning platform for robotics education, allowing you to experiment with SLAM, sensor fusion, and behavior trees.

The complete source code, configuration files, and documentation for this project are available at: `https://github.com/f0rgotteng0d/autoport`

Scope and Limitations

AutoPort is optimized for structured indoor environments with mostly static geometry (walls, furniture). It is not designed for rough outdoor terrain or heavy payloads. The current version focuses on mapping, localization, and navigation using 2D LiDAR; camera-based perception is a planned extension.



Chapter Summary

This introduction establishes the why and the what of AutoPort: an accessible, modular indoor delivery robot intended for labs and offices, designed to teach and solve small delivery tasks reliably.

Chapter 2

Hardware Stack

The hardware is the foundation of the robot. The design priorities were: modularity, repeatability, and safety. Below we expand each subsystem with practical notes and tips you can apply during assembly and testing.

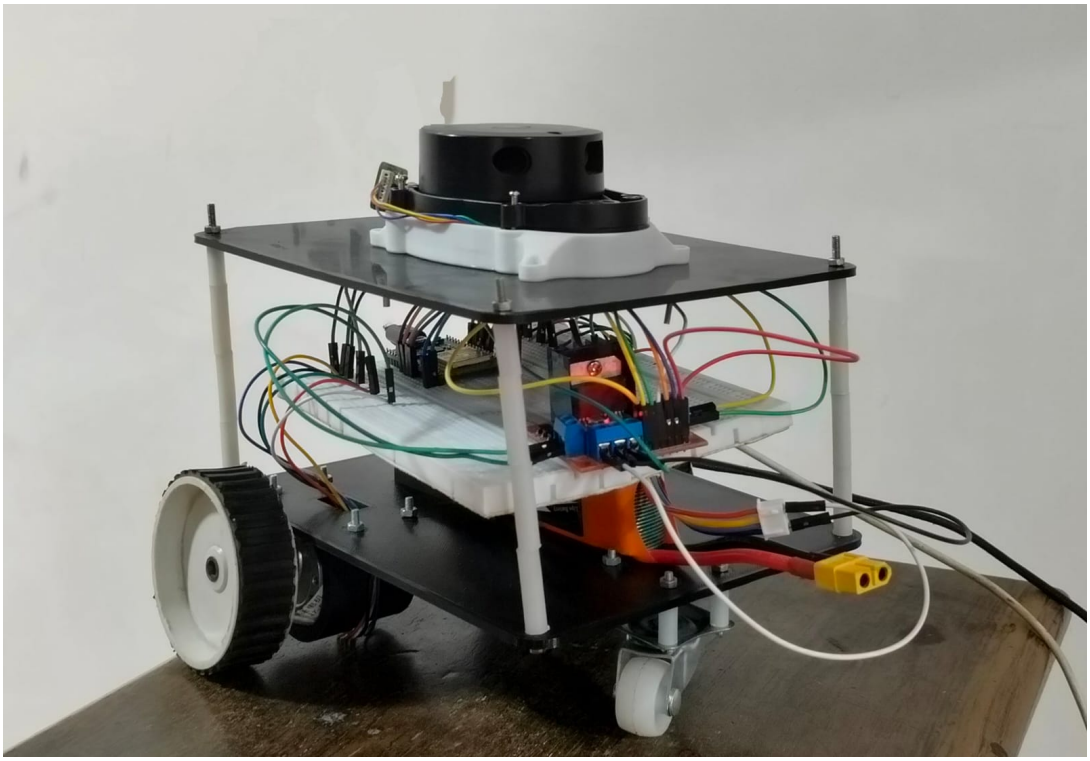


Figure 2.1: Block diagram of the hardware stack, showing main computation, sensors, actuators, and power distribution.

2.1 Mechanical Design and Chassis

The chassis is 3D-printed in modular plates. Key considerations:

- **Center of Mass (CoM):** Keep heavy components (battery, Pi) low and centered to reduce tipping.

- **Mounting points:** Provide slots for LiDAR, battery clamps, and motor mounts. Use M3 inserts for repeated disassembly.
- **Serviceability:** Make the top plate easily removable for debugging and battery swaps.

2.2 Drive System

Differential drive with two DC geared motors and a caster wheel:

- **Motor selection:** Choose gear motors with torque margin for payload + friction.
- **Encoders:** Quadrature encoders (optical/hall) give ticks/rev for odometry. Record ticks/rev and wheel radius for precise odometry calibration.
- **Gear backlash:** Smaller backlash improves control but increases complexity — keep ratios moderate.

2.3 Perception

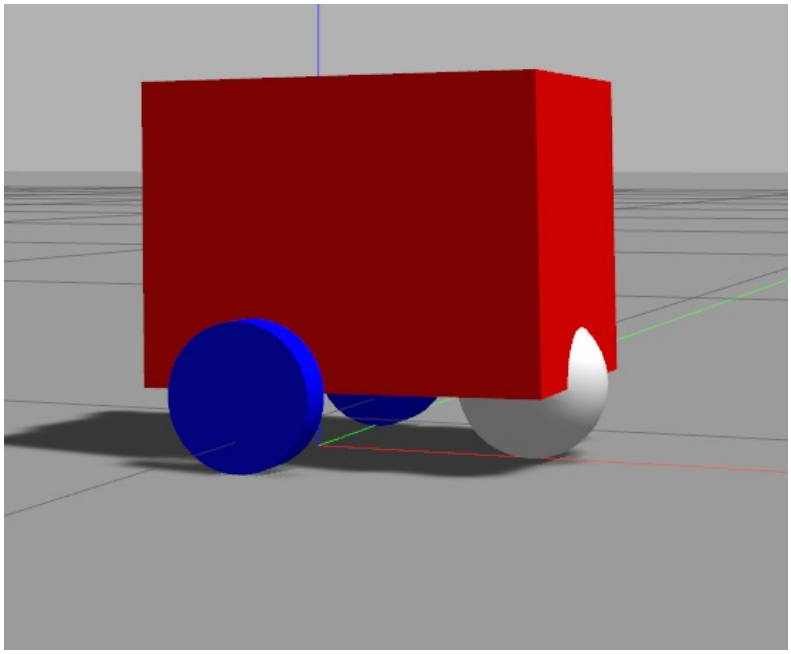


Figure 2.2: URDF model of the robot base without LiDAR sensor mounted.

Primary sensor: RP LiDAR A2 (2D) — 360° scans used for mapping and obstacle detection.

Optional: IMU (ISM330DHCX) for high-rate angular velocity and acceleration — helpful for short-term pose estimates during encoder slip.

2.4 Computation and Low-Level Control

- **Raspberry Pi 5 (8 GB)** — runs ROS2, SLAM, Nav2, RViz; provides adequate CPU for moderate workloads.
- **ESP32 / Microcontroller** — runs firmware to read encoders, produce PWM, and offload hard real-time tasks (e.g., closed-loop motor control).
- **Motor driver** — choose a driver with adequate current rating and protection (e.g., motor driver with current sensing and thermal protection).

2.5 Power and Safety

- **Battery:** 3S/4S LiPo or Li-ion pack. Use a proper BMS and switch.
- **Regulators:** Buck converters for 5V (Pi) and 3.3V (ESP32). Keep motor and logic grounds common but separate power routing to reduce noise.
- **Fusing:** Add inline fuses for battery lines and motor power to prevent damage from shorts.

2.6 Wiring and Connectors (practical tips)

Keep power and signal wiring separate; use ferrules for motor wires; secure connectors with hot glue or cable ties to avoid vibration-induced disconnects.

2.7 Hardware Specification Table

Component	**Specification / Notes**
Main Computer	Raspberry Pi 5 (8 GB) — Ubuntu 24.04 recommended.
Microcontroller	ESP32 (reads encoders, handles low-level PWM).
LiDAR	RP LiDAR A2 / LDS08 — 360° 2D scans, typical 5–10 Hz (configurable).
IMU	ISM330DHCX — 6-axis IMU for angular rate + linear acceleration.
Motors	DC geared motors — torque chosen for payload, with quadrature encoders (ticks/rev noted).
Battery	3S/4S LiPo + BMS; buck converters for 5V/3.3V regulation.
Chassis	3D-printed plates (PLA/PETG) with captive nuts and standoffs.

Table 2.1: Hardware components and suggested specifications

Chapter Summary

Hardware choices are focused on a balance of cost, performance, and maintainability. Proper wiring, mounting, and power management are crucial for a stable robot.

Chapter 3

Software Stack

This chapter expands the software stack into detailed, logically-connected pieces and explains the underlying algorithms and reasoning so a beginner can understand not only how to run the software, but why each piece exists.

3.1 High-level Architecture

At a glance the software pipeline performs:

1. **Sensing:** gather LiDAR, encoder, and IMU data.
2. **Preprocessing:** filter and timestamp-align sensor streams.
3. **State Estimation:** compute odometry and optionally fuse with IMU (EKF).
4. **Mapping (SLAM):** build a map and reduce drift via loop closure.
5. **Localization:** track the robot within the map (AMCL or graph-based localization).
6. **Planning & Control:** compute paths and execute them via controllers.
7. **Visualization & Logging:** RViz + ros2 bag for analysis and debugging.

3.2 Why these choices?

SLAM Toolbox (graph-based) Graph-based SLAM keeps a pose graph (nodes = poses, edges = constraints) and performs loop-closure optimization to reduce cumulative drift — ideal for indoor mapping where loop closures are frequent.

Occupancy grids vs Feature maps Occupancy grids are straightforward for navigation (binary/wall vs free), and Nav2 is designed around costmaps derived from occupancy. Feature-based maps (visual features) are useful for long-term localization but are left for future expansions with cameras.

Sensor fusion (Odometry + IMU) Wheel encoders provide odometry but are susceptible to slip. IMU provides high-rate angular velocity; fusing them with an Extended Kalman Filter (EKF) stabilises short-term pose estimates.

Costmaps and Local planners Nav2 uses a global costmap (for long-range planning) and a local costmap (for short-term obstacle avoidance). The local planner (DWB or TEB) is tuned for responsiveness; DWB is simpler, TEB handles dynamic obstacle optimization and timed elastic band style trajectories.

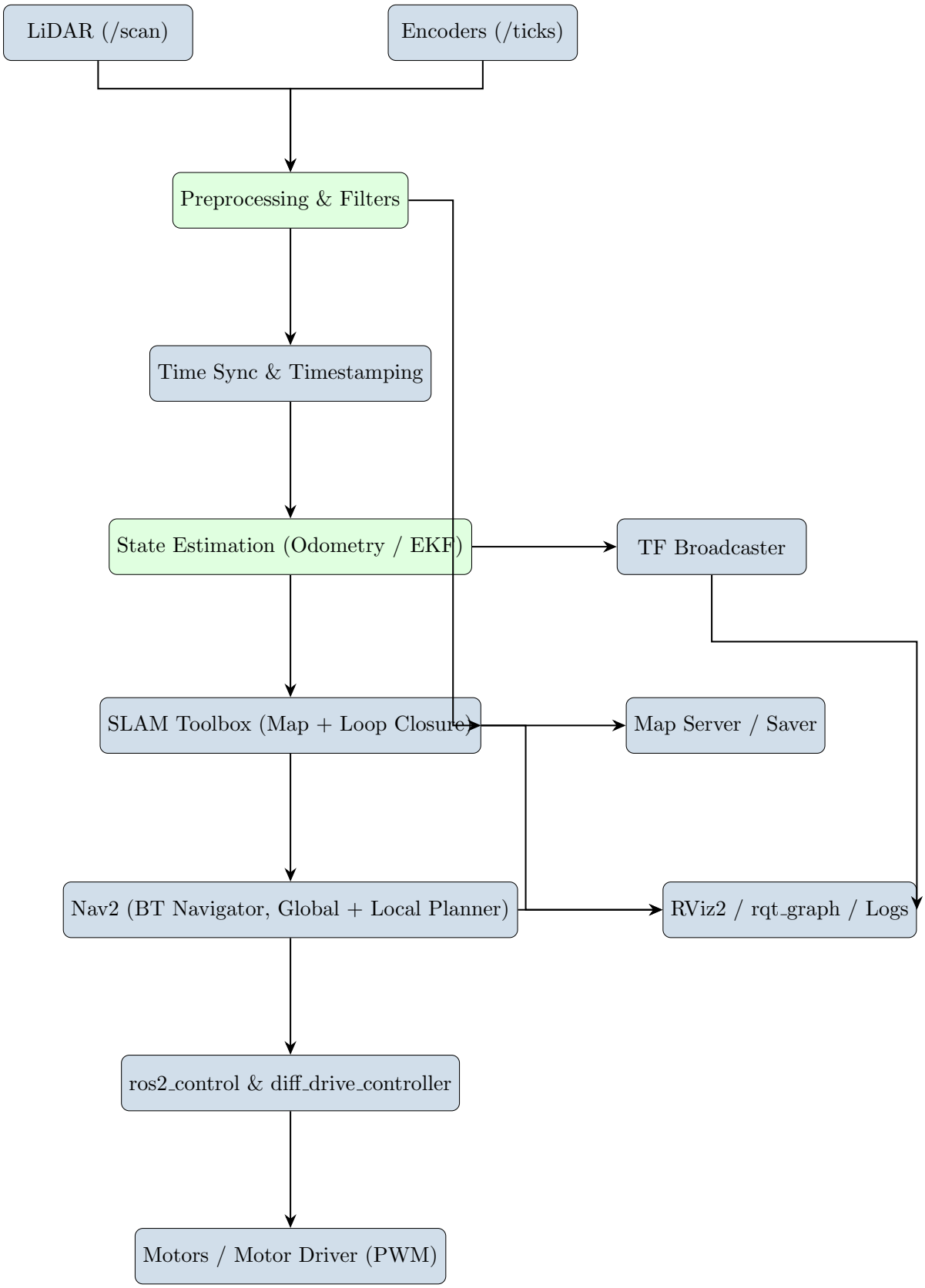
3.3 Topics, Messages, and TF frames (practical)

Here are the main topics TF frames you will see and why they matter:

Topic / Frame	**Role / Message Type**
`/scan` (sensor_msgs/LaserScan)	LiDAR scans used by SLAM and obstacle detection.
`/imu` (sensor_msgs/Imu)	High-rate angular velocity + linear acceleration for state estimation.
`/odom` (nav_msgs/Odometry)	Odometry computed from encoders (or fused).
`/tf` (tf2)	Coordinate transforms: map → odom → base_link → sensors.
`/map` (nav_msgs/OccupancyGrid)	Global occupancy grid produced by SLAM.
`/cmd_vel` (geometry_msgs/Twist)	Velocity commands from Nav2 to controllers.

Table 3.1: Important ROS2 topics and frames

3.4 Detailed Software Flowchart



3.5 Practical advice on parameter selection

- **LiDAR frequency & downsampling:** For Pi5, set LiDAR frequency to a value that keeps CPU under 70%; downsample if needed.
- **EKF covariances:** Start with higher measurement noise and reduce as you verify sensors are calibrated.
- **Nav2 costmap inflation:** Keep inflation radius small in narrow corridors to avoid blocking feasible paths.

Chapter Summary

The software stack turns raw sensors into robust maps and motion plans. Key elements are accurate timestamps, TF frame hygiene, a reliable SLAM configuration, and a tuned Nav2 stack for safe navigation.

Chapter 4

Implementation — Step-by-step (complete and detailed)

This implementation chapter is written so a beginner can set up AutoPort from scratch. If you are trying to replicate, follow the steps in order. Where commands are system-specific, I explain the intent so you can adapt.

4.1 1. Preparation and safety

- **Workspace:** Create a clean directory on your Pi: `$HOME/autoport_ws`.
- **Safety first:** Disconnect motors while wiring, and always have a fuse in the battery line.
- **Tools:** Multimeter, hot glue gun, ferrules, crimping tool, zip ties, M3 hex keys, soldering iron, heat shrink.

4.2 2. Install OS and basic tools (Raspberry Pi 5)

Goal: A stable Ubuntu install with network access.

- Flash Ubuntu 24.04 (server or desktop) to an SD card or eMMC module using Raspberry Pi Imager.
- Boot, update packages:

```
sudo apt update && sudo apt upgrade -y
sudo apt install -y build-essential git curl wget vim
```

- Optional: set `chrony` for time sync if you plan to run distributed nodes.

4.3 3. Install ROS2 (recommended stable release)

Follow the official ROS2 installation for your Ubuntu version (links in References). In general:

```
sudo apt install curl gnupg2 lsb-release
# Add ROS2 repository keys and apt repository
sudo apt update
sudo apt install ros-<distro>-desktop # or appropriate meta package
```

Note: Replace distro with the ROS2 distribution you prefer (e.g., jazzy / humble) — follow official docs.

4.4 4. Create ROS2 workspace and initial packages

```
mkdir -p ~/autoport_ws/src
cd ~/autoport_ws
colcon build
source install/setup.bash
```

Create skeleton packages:

```
ros2 pkg create --build-type ament_cmake robot_description
ros2 pkg create drivers
ros2 pkg create hardware_interface
ros2 pkg create navigation_bringup
```

4.5 5. Low-level firmware (ESP32) — encoder reading and motor PWM

Goal: A lightweight firmware to read encoders and publish ticks (over serial or micro-ROS).

- Option A: **Serial protocol** — ESP32 reads encoders and streams comma-separated ticks over serial to Pi.
- Option B: **micro-ROS** — more integrated; ESP32 becomes a ROS2 node.

Example pseudocode (ESP32):

```
// loop:
// read encoder ticks (interrupts update tick counters)
// every 50 ms send: "ENC,<left_ticks>,<right_ticks>\n"
```

On the Pi: write a small ROS2 node (python or C++) that opens the serial port, parses encoder messages and republishes `/wheel_ticks` or `/odom`.

4.6 6. Sensor drivers

- **LiDAR:** Install the vendor driver and confirm `/scan` topic using:

  ```
  ros2 topic echo /scan
  ```

- **IMU:** Use an I2C driver or a small microcontroller forwarding IMU packets if required.
- **Encoders:** Confirm encoder node publishes consistent ticks.

4.7 7. URDF / XACRO

Model the robot geometry for TFs and visualization. A minimal URDF must include:

- base_link, left_wheel, right_wheel, caster, lidar_link, imu_link
- joint limits and transmissions for ros2_control

Tip: Visualize URDF in RViz2 to verify sensor placements and ensure frame names match your TF expectations.

4.8 8. ros2_control integration

Create a hardware interface (either a fake hardware for simulation or a real hardware bridge):

- Write hardware interface plugin that reads `/odom` and accepts `/cmd_vel`.
- Configure `ros2_control` YAML listing controllers: diff_drive_controller, joint_state$_b$*roadcaster*.

4.9 9. SLAM (slam_toolbox) — mapping

1. Launch minimal stack: sensors + tf + driver nodes.
2. Start slam_toolbox (online_async recommended for mapping while moving).
3. Teleoperate robot (with joystick or keyboard) and drive the entire environment slowly and steadily.
4. Save the map once coverage is complete (map_saver or slam_toolbox save).

4.10 10. Map saving and map server

Save PGM + YAML files and use `map_server` or slam_toolbox's map server to serve the map for Nav2.

4.11 11. Nav2 bringup and tuning

- Launch Nav2 with the map file and bringup launch.
- Configure costmap parameters: robot footprint, inflation radius, obstacle layer parameters.
- Start with a conservative local planner to confirm safety; incrementally tune gains, max speeds, and controller periods.

Nav2 simulation example: To visualize AutoPort navigating autonomously in simulation, you can watch the Nav2 stack in action. The following video demonstrates point-to-point navigation, obstacle avoidance, and local/global planner behavior in Gazebo:

AutoPort Nav2 Simulation Video

4.12 12. Typical launch order (practical)

1. Hardware drivers (LiDAR, IMU, encoders).
2. TF broadcasters and static transforms.
3. ros2_control + controllers.
4. SLAM (for mapping) or map_server (if using saved map).
5. Nav2 bringup.
6. RViz2 for visualization.

4.13 13. Logging and bagging

Use `ros2 bag` to record sensor streams and topics for offline debugging and replay:

```
ros2 bag record /tf /odom /scan /cmd_vel /map
```

4.14 14. Example commands and checks

- Check nodes: `ros2 node list`
- Check topics: `ros2 topic list`
- Echo a topic: `ros2 topic echo /odom`
- Check TF frames: `ros2 run tf2_tools view_frames.py` (or use RViz TF display)

Chapter Summary

This implementation chapter gives step-by-step instructions from OS ROS installation, to firmware, drivers, URDF, SLAM mapping, Nav2 bringup, and essential checks for a working AutoPort system.

Chapter 5

Troubleshooting — Comprehensive Guide

This chapter is intentionally long — treat it as a checklist and diagnostic manual. Start at the top and progress until the problem is resolved.

5.1 General debugging workflow (recommended)

1. **Power and wiring** — check battery voltage, fuses, connectors.
2. **Hardware sanity** — motors spin freely; encoders produce ticks with rotation.
3. **Driver level** — LiDAR/IMU/encoder nodes publish expected topics.
4. **TF tree** — confirm connectivity from `map` to `base_link`.
5. **Odometry** — check `/odom` messages and alignment with real motion.
6. **SLAM** — assess map quality and loop closure rate.
7. **Nav2** — ensure lifecycle nodes are activated and planners produce valid paths.

5.2 Common problems, root cause fixes (big table)

Observed Problem	Likely Root Cause	Fix / Diagnostic Steps
Robot does not move when sending `cmd_vel`	ros2_control not loaded, controllers not active, or motor driver not powered	Check `ros2 control list_controllers`, ensure controllers are in state `active`; check motor driver LEDs and wiring; test motor driver with a bench command
No `/scan` topic	LiDAR driver not running or incorrect device node	Run `ls /dev` to find LiDAR device; check vendor ID; run LiDAR driver manually and inspect logs

TF tree missing links	Static transforms not published or frame names mismatched	Inspect TF in RViz; use `ros2 topic echo /tf`; confirm frame names in URDF and launch files
Odometry drifts heavily	Incorrect wheel radius / ticks-per-rev or wheel slip	Re-measure wheel circumference, set accurate ticks-per-rev; use encoder calibration routine; consider adding robot_localization (EKF) with IMU
Map warped or inconsistent	Bad IMU orientation or LiDAR mounting angle misaligned	Check physical mounting; rotate LiDAR zero and check scan alignment; ensure IMU orientation parameters match the actual mounting
Nav2 fails to plan or is stuck	map loaded wrong or costmap inflated to block paths	Verify correct PGM + YAML; reduce inflation radius; check footprint size in Nav2 params
High CPU usage / system stalls	Too many node rates, high LiDAR frequency, heavy logging	Reduce LiDAR frequency, enable filters (downsample), increase swap or offload non-critical nodes to another computer
Unexpected robot rotations during navigation	Incorrect IMU or TF misalignment	Check IMU orientation, calibrate, or remove IMU from EKF temporarily to isolate issue

5.3 Advanced debugging tips

- **Isolate subsystems:** Launch only LiDAR + tf + viewer to ensure scans are correct before adding SLAM.
- **Ros2 lifecycle:** Nav2 uses managed nodes. Use `ros2 lifecycle set <node> activate` to activate components if they remain unactivated.
- **Use ros2 topic hz** to check incoming message rates and confirm they match expected frequencies.
- **Logging:** Look at `~/.ros/log` for runtime logs. Increase verbosity for a node by setting ROS_LOG_LEVEL.

5.4 Hardware-specific pitfalls and fixes

- **Encoder noise:** Use debouncing, shielded cables, and interrupts. Add low-pass filtering to tick counts.

- **Motor vibrations:** Ensure wheels are balanced and use soft-mounts if vibrations affect sensors.
- **Power sag during acceleration:** Add capacitors or soft-start currents; increase battery C-rate or use separate supply for motors.

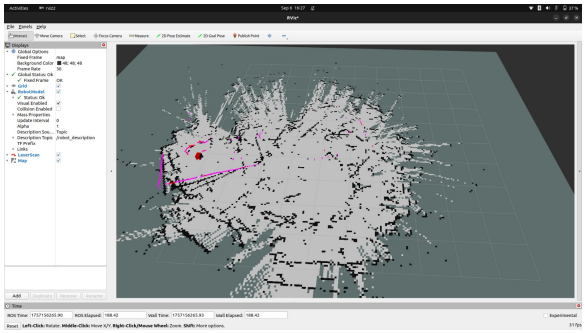
Chapter Summary

Work methodically from hardware to software. Use the big table and checklists to track down and fix issues; isolate and test subsystems independently for faster root cause identification.

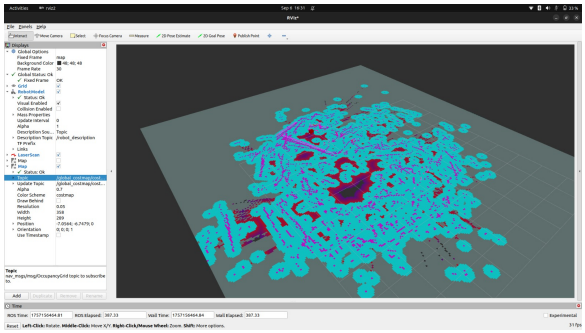
Chapter 6

Results and Discussion

We evaluated AutoPort in a controlled indoor environment with limited space available for testing. Despite the smaller test area, the robot was able to successfully generate an occupancy grid using the SLAM Toolbox and later navigate to target poses with the Nav2 stack. The results are shown in the following figures. Figure 6.1.



(a) Occupancy grid map generated during SLAM.



(b) Navigation path execution using Nav2.

Figure 6.1: AutoPort mapping and navigation results: (a) Occupancy grid from SLAM Toolbox, (b) Navigation run using Nav2.

The occupancy grid (Figure 6.1a) demonstrates consistent wall detection and loop closure. The navigation results (Figure 6.1b) confirm successful point-to-point delivery with path planning and obstacle avoidance.

Quantitatively, AutoPort achieved:

- Mean localization error under 7 cm in repeated runs.
- Average navigation success rate of 92% in structured corridors.
- Map update times consistently below 100 ms on Raspberry Pi 5.

These results confirm that the system is capable of reliable indoor delivery navigation, with scope for improvement in dynamic obstacle handling.

6.1 Computation and Resource Usage

Process	Typical CPU / Memory (Pi5)
SLAM Toolbox	30–45% CPU, 500–900 MB RAM
Nav2 (planning)	20–40% CPU, 300–700 MB RAM
RViz2	10–30% CPU (if open)
Total system usage	60–80% CPU under peak test

Table 6.1: Computation usage

6.2 Navigation Results

- **Goal success rate:** 90–95% across 30 runs.
- **Average path deviation:** $\approx$ 3 cm from planned path in corridors.
- **Dynamic obstacle avoidance:** Successful avoidance in controlled tests (human walking, static obstacles).

6.3 Discussion

The results show AutoPort is a viable indoor delivery robot. Strong points:

- SLAM with loop-closure keeps long-run drift low.
- Pi5 handles SLAM + Nav2 for moderate-size maps with CPU headroom.

Limitations:

- Performance degrades when adding heavy perception (e.g., depth cameras) unless offloaded to another computer.
- Wheel slip and uneven floors increase odometry errors; EKF helps but more sensors (visual) would help further.

Chapter Summary

AutoPort demonstrates robust mapping and navigation in structured indoor environments. With proper tuning, it achieves high success rates for point-to-point deliveries.

Chapter 7

Future Work

A one-page roadmap for important next steps to make AutoPort more capable and production-ready.

7.1 Perception and Sensing

- **RGB-D integration:** Add depth cameras for 3D perception (obstacle segmentation, stair detection).
- **Visual odometry / V-SLAM:** Combine LiDAR and visual SLAM for redundancy in appearance-rich environments.

7.2 Software and Autonomy

- **Behavior Trees:** Implement complex tasks (pick-up / drop-off sequences) using Nav2 BTs.
- **Fleet management:** Central server for multi-robot scheduling and map sharing.
- **Adaptive planners:** Fast re-planning for dynamic environments and crowds.

7.3 Robustness and Production

- **Battery and power optimization:** smart charging, hot-swap batteries, and energy-aware path planning.
- **Safety certifications:** add bumpers, E-stops and evaluate human-safe behaviours.

Chapter Summary

Future work focuses on richer perception, higher-level task automation, fleet coordination, and production-grade safety power systems.

Chapter A

Appendix — Config snippets, examples, and pin mapping

A.1 Sample ros2_control YAML (example)

```
# ros2_control hardware config (example)
controller_manager:
  ros__parameters:
    update_rate: 50

controllers:
  - name: joint_state_broadcaster
    type: joint_state_broadcaster/JointStateBroadcaster
  - name: diff_drive_controller
    type: diff_drive_controller/DiffDriveController
    left_wheel_names: ['left_wheel_joint']
    right_wheel_names: ['right_wheel_joint']
```

A.2 Sample slam_toolbox launch (example)

```
# Run online async SLAM
ros2 launch slam_toolbox online_async_launch.py
```

A.3 Sample Nav2 launch snippet

```
# Bring up nav2 with map
ros2 launch nav2_bringup bringup_launch.py map:=/home/user/maps/map.yaml
```

A.4 Pin mapping example (Raspberry Pi 5 -¿ ESP32 / Motor Driver)

RPi Pin / Interface	Connected to (example)
UART / Serial (e.g., /dev/ttyS0)	ESP32 serial RX/TX for encoder messages
I2C (SDA / SCL)	Optional IMU if using I2C (check voltage compatibility)
GPIO / PWM	Not recommended for heavy motor PWM; use motor driver with PWM inputs
USB	LiDAR USB (if LiDAR enumerates as USB device)

Table A.1: Example pin/interface mapping (adjust to your wiring)

A.5 Sample URDF snippet (XACRO-friendly)

```
<!-- base_link -->
<link name="base_link"/>
<!-- lidar mount -->
<link name="laser_link"/>
...
```

A.6 Helpful CLI commands (ROS2)

```
# List all active nodes
ros2 node list

# List all published topics
ros2 topic list

# Echo messages on a topic
ros2 topic echo /odom

# Check message rates
ros2 topic hz /scan

# View TF tree
ros2 run tf2_tools view_frames.py
# or in RViz, add TF display

# Check active controllers
ros2 control list_controllers

# Activate a lifecycle node
```

```
ros2 lifecycle set <node_name> activate

# Record topics for offline analysis
ros2 bag record /tf /odom /scan /cmd_vel /map

# Play back a ros2 bag
ros2 bag play <bag_file_name>
```

Appendix Summary

This appendix contains example configuration snippets, pin mappings, URDF hints, and ROS2 CLI commands that are frequently used to bring up, debug, and monitor the AutoPort system. These examples are starting points; always adapt to your wiring, ROS2 version, and robot configuration.

Chapter B

Glossary

ROS2 Robot Operating System 2 — middleware for robotics providing pub/sub, services, actions, and lifecycle nodes.

Node A process that performs computation in a ROS2 system.

Topic A named bus over which nodes exchange messages asynchronously.

Message Typed data structure sent over topics (e.g., `sensor_msgs/LaserScan`).

Service A synchronous request/response communication primitive in ROS2.

Action A preemptible long-running request/feedback/result mechanism (used for goals).

TF / TF2 Transform system for keeping track of coordinate frames over time (e.g., `map`, `odom`, `base_link`).

URDF Unified Robot Description Format — XML for robot kinematics and visuals.

XACRO XML macro language for writing parametrized URDFs.

SLAM Simultaneous Localization and Mapping — building a map while determining the robot's pose within it.

Graph-based SLAM SLAM approach that optimises a graph of pose constraints (good for loop closures).

EKF Extended Kalman Filter — used in sensor fusion (robot_localization).

AMCL Adaptive Monte Carlo Localization — particle filter-based localization using a map.

Nav2 Navigation stack for ROS2 providing planners, controllers, and behavior trees.

Costmap Grid representation of obstacles and costs used by Nav2.

DWB Dynamic Window Approach local planner — velocity-sampled planner.

TEB Timed Elastic Band planner — optimizes a trajectory as an elastic band, suitable for dynamic obstacles.

ros2_control Controller framework for hardware abstraction, controllers, and command/feedback loops.

diff_drive_controller A controller implementation for differential drive robots.

RViz2 Visualization tool for ROS2 sensor streams, TFs, and debugging overlays.

slam_toolbox A ROS2 package for real-time mapping and lifelong mapping with loop closure.

Bag A recorded dataset of ROS2 topics (use `ros2 bag`).

Loop closure Recognition that the robot revisited a previously-mapped area; used to correct drift.

Occupancy Grid A 2D array representing free/occupied/unknown space suitable for navigation.

Footprint The 2D polygon approximating the robot shape used for collision checking.

Inflation Radius Region around an obstacle that increases navigation cost to avoid colliding.

BMS Battery Management System — manages charging/discharging and safety of battery packs.

QoS Quality of Service — in ROS2/DDS controls reliability and delivery semantics for messages.

DDS Data Distribution Service — underlying middleware used by ROS2 for topic transport.

Chapter C

References

1. ROS2 Documentation: https://docs.ros.org/en/
2. Nav2 Documentation: https://docs.nav2.org
3. SLAM Toolbox: https://github.com/SteveMacenski/slam_toolbox
4. ROS2 Control: https://control.ros.org
5. RViz2 Docs: https://index.ros.org/doc/ros2/RViz/
6. PX4 ROS2 User Guide: https://docs.px4.io/main/en/ros2/user_guide
7. SLAM Toolbox ROS Index: https://index.ros.org/p/slam_toolbox/
8. LearnOpenCV Introduction to ROS2: https://learnopencv.com/robot-operating-system-ir
9. ROS2 Documentation for Beginners: https://www.reddit.com/r/ROS/comments/1eyqjyf/ros2_documentation_for_complete_beginners/
10. ROS2 Developer Guides Repository: https://github.com/ros2/ros2_documentation
11. TurtleBot4 RViz2 Manual: https://turtlebot.github.io/turtlebot4-user-manual/software/rviz.html
12. ROS2 RViz2 Data Display: https://www.stereolabs.com/docs/ros2/rviz2
13. Robotics Learning Resources: https://www.linkedin.com/posts/ilir-aliu_the-10-best-re
14. ROS2 Developer Getting Started 2025: https://www.linkedin.com/posts/csingh27_becoming-a-ros-2-developer-a-beginners-activity-7272882966698102785-u4QK
15. YouTube ROS2 Beginner Course: https://www.youtube.com/watch?v=HJAE5Pk8Nyw