Parking Management Bot

Apurv Kushwaha
kushw009@umn.edu

Rutav Narkhede
narkh003@umn.edu

December 10, 2025

Abstract

Indoor parking facilities often struggle with misparked vehicles that block traffic flow and waste space. Manual intervention is time-consuming and impractical for large facilities. This project developed an autonomous parking management system in ROS 2 and Gazebo to address this problem through automated detection and relocation. The implementation consists of two workspaces: the first uses an overhead camera with Nav2 navigation to detect and approach misparked vehicles, while the second explores a towing robot concept with a lifting mechanism to physically relocate cars. The overhead camera system detects vehicle positions accurately despite camera orientation challenges that required coordinate transformation. Navigation employs static maps with obstacle avoidance, achieving successful path planning in test scenarios. The second workspace demonstrates lift control and vehicle attachment through simulation physics. Testing in simulated parking environments showed the system can identify misparked vehicles, navigate around obstacles, and control lifting mechanisms. While the project focused on static environments, it demonstrates solutions to coordinate transformations, autonomous navigation, and simulated manipulation that are relevant to mobile robotics applications.

I. INTRODUCTION

Parking management in indoor facilities like airports, shopping malls, and office complexes presents ongoing operational challenges. Vehicles parked outside designated spots create bottlenecks, reduce available capacity, and complicate traffic flow. Current solutions rely on manual enforcement through attendants or towing services, which are expensive and slow to respond. This project explores whether robotic systems could handle this task autonomously.

The project started with an ambitious goal of building a robot that could work in dynamic environments with moving cars and pedestrians. After initial planning and instructor feedback, the scope was narrowed to focus on static, well-structured parking lots. This decision allowed deeper investigation into fundamental robotics problems rather than spreading effort across too many features that might not work reliably.

Two separate systems were built that approach the parking problem from different angles. The first workspace uses an overhead camera mounted above a parking lot to detect cars and their orientations. It then guides a mobile robot using ROS 2's Nav2 navigation stack to reach misparked vehicles. The second workspace continues this work by developing a compact towing robot that positions itself under the vehicle at the towing location, lifts it using a prismatic joint, and plans trajectories to relocate it to the correct parking spot. Both systems run in Gazebo simulation, which provided full control over the environment and physics parameters.

The project required solving several interesting technical problems. The overhead camera's 90-degree pitch rotation meant that what appears as the X-axis in the world becomes the Y-axis in the camera image. Getting this coordinate

[0]supplementary materials available at: https://github.com/ApurvK032/Autonomous-Car-Parking-Bot.git

transformation right took multiple attempts and careful debugging. The navigation system needed static maps with parked cars marked as obstacles, and dozens of parameters had to be tuned to make Nav2 work smoothly. The lifting mechanism presented its own challenges since ROS 2 Humble doesn't have some of the attachment plugins available in ROS 1. The team explored using friction-based physics in simulation to hold the car on the lift platform, though this approach works only in simulation and wouldn't transfer directly to real hardware.

The implementation demonstrates detection, navigation, and lift control working in their respective phases. The towing, trajectory planning, and placement phase remains incomplete due to time constraints. Despite this, the project shows working solutions to coordinate transformations, autonomous navigation with obstacle avoidance, and ros2_control integration for joint manipulation. All robot models and environment components were designed in URDF and Gazebo world files rather than importing pre-built models, which provided complete control over the system design but required additional development time. The rest of this report describes the technical approach, the problems encountered along the way, and lessons learned about building autonomous systems in ROS 2.

II. PROBLEM DESCRIPTION

II.1 Problem Statement

Large parking facilities waste space and create safety hazards when vehicles park incorrectly. A car angled across two spots or blocking a driveway forces other drivers to circle looking for alternatives. In busy locations like airport parking or event venues, even a few misparked cars can cause significant congestion. Parking attendants can ticket these vehicles, but that doesn't solve the immediate blockage problem. Manual towing requires specialized equipment and trained operators, making it too slow and expensive for routine enforcement.

The goal was to build a robotic system that could detect misparked vehicles and move them to proper locations without human intervention. The robot needs to understand the parking lot layout, identify which cars are parked incorrectly, navigate safely around other vehicles, and physically relocate the problem cars. This requires integrating perception, planning, and manipulation capabilities into a single autonomous system.

The scope focused on indoor parking lots with painted lines marking parking spaces. The lot layout is assumed to be known in advance and other vehicles remain stationary during operation. These simplifications allowed concentration on getting the core detection, navigation, and lifting systems working properly rather than dealing with unpredictable moving obstacles or unknown environments.

II.2 Technical Challenges

Building this system required solving problems across multiple robotics domains. The perception challenge started with mounting an overhead camera high above the parking lot. While this gives a clear view of the entire area, the camera must point straight down, which means a 90-degree pitch rotation. This rotation doesn't just tilt the image, it swaps the coordinate axes. What the world considers the X direction appears as Y in the camera frame, and vice versa. Every position and angle measurement from the camera needed transformation before it could be used for navigation. Getting these transformations correct took several debugging sessions where detected positions were visually compared against ground truth markers.

The navigation system used ROS 2's Nav2 stack, which required creating a static map of the parking lot showing walls, parking space boundaries, and correctly parked cars as obstacles. Parameters controlling obstacle clearance, turning speed, and path replanning were tuned. The mecanum drive system added complexity since it can move sideways, requiring careful controller configuration to use this capability effectively.

The lifting mechanism presented the hardest technical challenge. Force-based joint control didn't work reliably due to friction, so the approach was switched to ros2_control's position controller which allowed commanding specific lift heights. Making the car stay attached during movement was tricky since ROS 2 Humble lacks the attachment plugins

from ROS 1. The friction coefficient between surfaces was set to an extremely high value, which creates enough resistance that the car sticks to the lift, though this required careful tuning to avoid physics instability. Although this mechanism is completely impractical, it was implemented only for demonstration and continuing to the next phase of the project.

Beyond basic navigation, the system would need trajectory planning to move the lifted car to its destination. With the front wheels raised, the towing robot creates a longer articulated vehicle that pivots around the car's rear wheels. Planning smooth paths that account for this changed geometry and steering constraints becomes necessary to navigate without collision.

Coordinate frame management became complex as components were added. The system tracks relationships between world, map, odometry, and body frames. Nav2 expects a specific transform tree structure, and missing any transform causes navigation to fail. tf2 was used to publish these transforms, handling map-to-odom with a static transform since the simulation has perfect localization.

III. RESULTS AND INSIGHTS

III.1 System Architecture and Methodology

The project developed two independent workspaces that demonstrate different approaches to autonomous parking management. Both systems run in Gazebo simulation with ROS 2 Humble, but they tackle the problem from complementary angles. The first workspace focuses on detection and navigation in a large parking lot, while the second workspace implements a compact towing robot with lifting capabilities.

Workspace 1 operates in a 40m×40m parking lot with six designated parking spaces. An overhead camera captures the entire scene from above. The system consists of three main components working in sequence: a detector that identifies cars and their parking status, a goal publisher that determines where the robot should go, and the Nav2 navigation stack that plans paths and moves the robot to misparked vehicles. The complete pipeline runs autonomously, with components communicating through ROS topics.

Workspace 2 is a continuation of workspace 1. It uses a smaller 20m×20m environment with a single car that needs relocation. The towing robot is more compact and includes a lifting mechanism. Rather than using vision, this workspace focuses on the approach and lifting phases after the car has been located. The robot navigates to position itself under the vehicle, then raises a lift to attach the car for towing. A state machine coordinates the sequence: navigate to car center, advance to towing position, stabilize, lift, and hold.

Both workspaces share common ROS 2 patterns but implement them differently. Workspace 1 focuses on moving the bot from its initial position to approach the misparked car, handling detection and path planning through the parking lot. Workspace 2 continues from this point, demonstrating how the robot positions itself precisely at the towing location once it reaches the car, then executes the lifting sequence. This division allowed the team to develop and test each phase independently while maintaining a clear progression through the complete parking management workflow.

The methodological approach followed an incremental development pattern. Each subsystem was built in isolation, tested thoroughly, then integrated with other components. For Workspace 1, this meant getting detection working before adding navigation, and ensuring the map generator produced valid output before running Nav2. Workspace 2 progressed through navigation, then approach logic, and finally the lift mechanism. This phased approach allowed systematic debugging rather than facing a complex system where failures could come from anywhere.

Testing relied heavily on visual validation and logging. The detection system outputs annotated images showing detected car positions overlaid on the camera feed. Navigation testing involved watching the robot move in Gazebo while monitoring planned paths in rviz. The lift mechanism required observing physics behavior and checking joint states published by ros2_control. ROS 2's command-line tools were used extensively to echo topics, inspect transforms, and manually trigger actions during development.

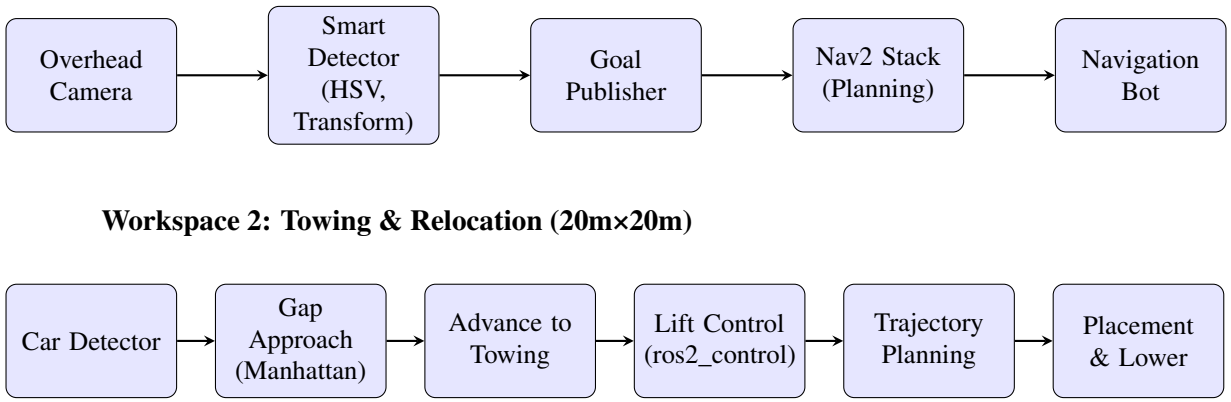

Figure 1: System architecture showing the dual workspace approach. Workspace 1 (top) uses overhead vision and Nav2 for detection and navigation. Workspace 2 (bottom) employs a towing robot with lift mechanism, trajectory planning, and state machine control.

III.2 Workspace 1: Detection & Navigation System

The first workspace was developed to demonstrate vision-based detection and autonomous navigation using ROS 2's standard tools. The system runs in a 40m×40m simulated parking lot with six parking spaces arranged in a 3×2 grid. An overhead camera mounted 35 meters above the ground captures the entire scene, and three vehicles (blue sedan, yellow SUV, red sedan) occupy various spaces. This height was chosen to cover the entire parking lot with a single high-quality camera for simplicity in simulation, though a real-world implementation would use multiple cameras positioned at lower heights throughout the facility to achieve similar coverage. The goal was to detect which cars are misparked and then navigate a mobile robot to reach them.

III.2.1 Phase 1: Detection System

The system operates in an indoor parking environment where it is assumed that the system has access to a pre-existing map showing parking space locations in an empty lot. This map serves as the baseline reference. The bot loads this map and compares it against the live overhead camera feed to identify which spaces are occupied and whether vehicles are parked correctly or misparked. This approach leverages the structured nature of indoor parking facilities where layouts are known and consistent.

For simplicity in this proof-of-concept, distinctly colored cars (red, blue, yellow) are used to streamline the detection process. However, it's worth noting that color detection serves only to locate vehicles in the frame; the actual parking assessment relies on geometric position and orientation analysis. This color-based module can be seamlessly replaced with more sophisticated detection models like YOLO or Mask R-CNN that recognize vehicles regardless of color, making the system generalizable to real parking lots with arbitrary vehicle appearances. The downstream coordinate transformation and parking validation logic would remain unchanged.

OpenCV [6] is used for the computer vision pipeline. The detection workflow starts by converting the camera image to HSV color space, which separates color information from brightness and handles varying lighting conditions better than RGB. Color ranges are defined for each vehicle and masks are created that highlight only those colors in the image. To clean up the masks and remove small noise spots, filtering operations are applied that smooth the detected regions. The boundaries of colored regions (contours) that are large enough to be vehicles are then identified, filtering out anything smaller than 500 square pixels. From these contours, the center point of each vehicle is calculated for position and rectangles are fitted around them to determine their orientation angle. These raw measurements go through coordinate transformations [8] to convert from camera pixel coordinates to real-world parking lot positions,

accounting for the overhead camera's orientation.

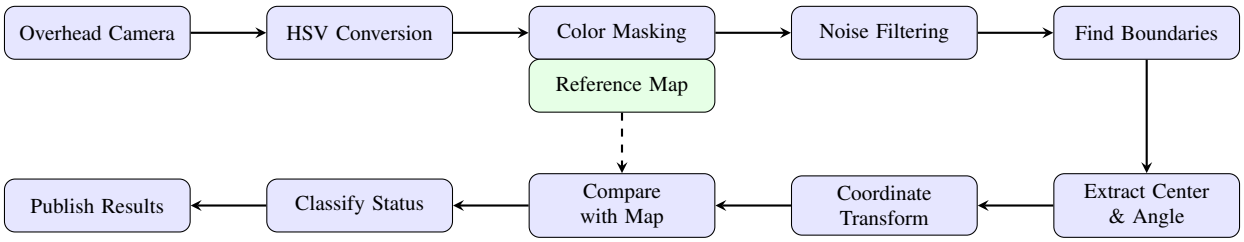

Figure 2: Detection pipeline flowchart showing the sequence from camera input to parking status classification. The reference map provides baseline parking space locations for comparison with detected vehicle positions.

The overhead camera's 90-degree pitch rotation caused the most interesting technical challenge. When a camera is rotated 90 degrees around its Y-axis to point straight down, it also rotates its perception of coordinate axes. The world's X-axis (east-west) appears as the camera's Y-axis in images, and the world's Y-axis (north-south) becomes the camera's X-axis. This was discovered by drawing green dots at parking space centers and seeing them appear in completely wrong positions. The fix required swapping X and Y coordinates everywhere the parser extracts positions from the world file.

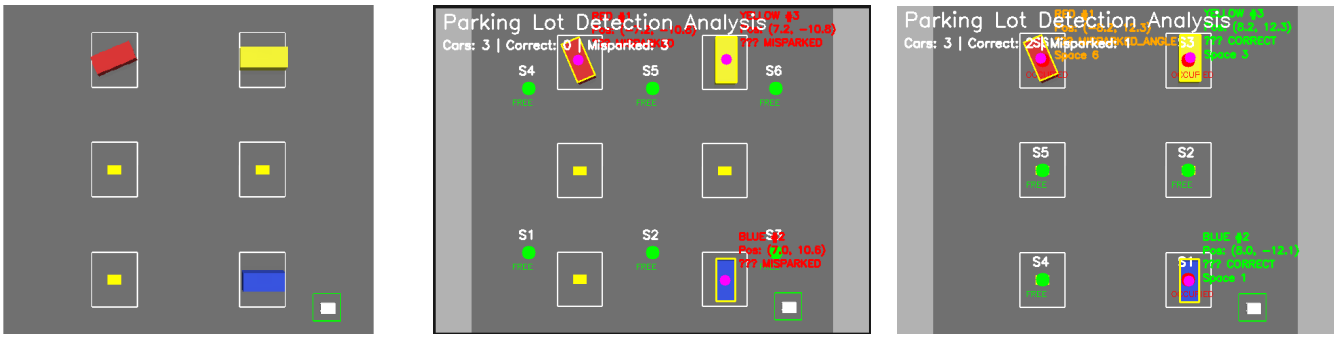

Figure 3: Coordinate transformation process. (a) Raw overhead camera image of the parking lot before detection. (b) Detection results with 90-degree pitch mismatch showing misaligned space markers and incorrect vehicle positions. (c) Corrected detection after applying coordinate transformation (XY swap and Y negation) with accurate parking status classification.

On top of the axis swap, pixel coordinates have their origin at the top-left corner with Y increasing downward, while world coordinates have Y increasing upward. This means Y needs to be negated when converting from pixels to world coordinates. The complete transformation chain goes: detect blob in image, find centroid in pixel coordinates, subtract image center, divide by pixels-per-meter scale factor, swap X and Y, negate Y, and finally world coordinates are obtained. Missing any step in this chain puts detected positions in the wrong place.

For orientation detection, angles are extracted from the minimum area rectangles fitted to car contours. These angles also need the 90-degree compensation, but subtraction rather than addition of the correction is required. This caused initial confusion since the position transform adds 90 degrees in effect, but the rotation direction differs.

Parking status classification uses point-in-polygon tests to assign cars to spaces. Each parking space has four corner points defining its boundary from the reference map. The system checks whether a car's center point falls inside any space polygon using a ray-casting algorithm. If the car is in a space and its orientation matches within 11 degrees (0.2 radians), it is marked as correctly parked. Cars in spaces but rotated wrong get marked as misparked by angle. Cars not in any space are misparked by position. This classification runs at a set frequency, and results get published as JSON messages for the navigation system to consume.

The final detection system identifies all three test cars correctly in every run performed. The blue sedan at space 4 and yellow SUV at space 6 register as correctly parked at 90 degrees orientation. The red sedan at space 3 shows as

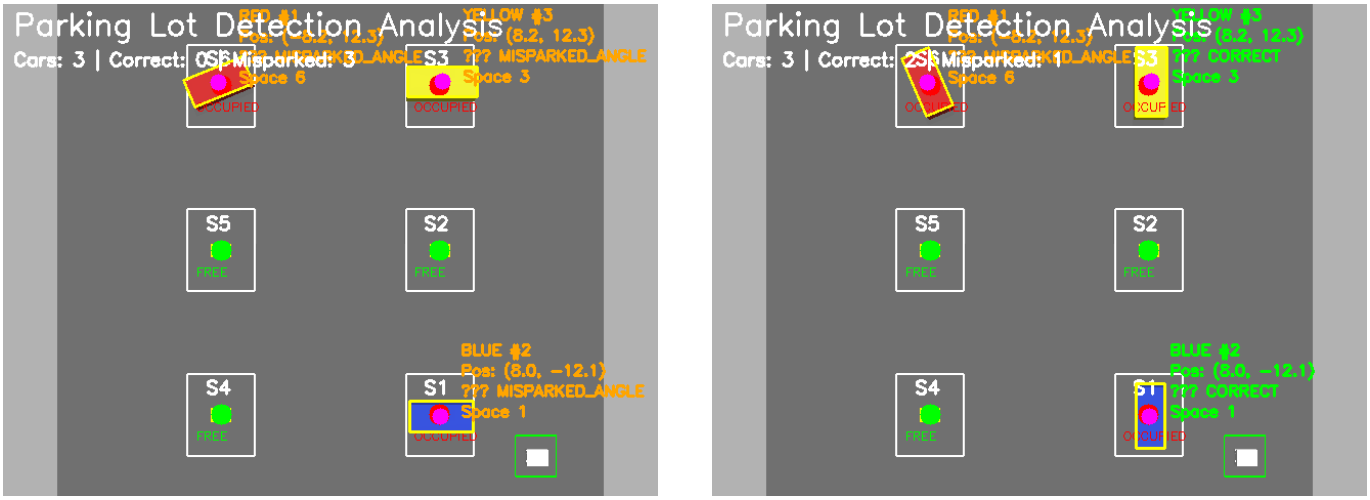

Figure 4: Detection system output showing parking status classification. Left image shows initial detection with red car misparked by angle (22.9° deviation). Right image demonstrates system sensitivity to orientation changes, detecting multiple misparked vehicles when angles deviate from expected 90° alignment.

misparked by angle, reporting 22.9 degrees off from the expected 90-degree orientation. This car was intentionally placed at 113 degrees (90 + 23) in the world file to test the detection, and measuring 22.9 degrees error confirms the system works within 0.1-degree accuracy. Position measurements stay within 10 centimeters of ground truth across all vehicles.

III.2.2 Phase 2: Navigation System

Integrating Nav2 navigation [3] required more setup work than anticipated. The first step was generating a static map showing the parking lot boundaries and obstacle locations. A map generator was written that creates an 800×800 pixel image representing the 40-meter square lot at 5-centimeter resolution. The generator draws walls around the perimeter, parking space outlines for reference, and filled black rectangles where cars are parked. These car obstacles come from the same configuration file the detector uses, ensuring consistency. The map gets saved as a PGM image file with an accompanying YAML file that tells Nav2 about resolution and origin coordinates.

Getting Nav2 to launch properly took troubleshooting several configuration issues. The behavior tree navigator [5] failed initially because it couldn't find plugins referenced in the default behavior tree file. The solution was to switch to a simpler tree file that only includes basic navigation behaviors, which worked fine for the static environment. The launch file needed careful timing - the robot spawns at 2 seconds, map server starts at 3 seconds, and the full Nav2 stack is delayed until 8 seconds. Starting everything simultaneously caused race conditions where nodes tried to use transforms before they existed.

The transform tree needs three frames: map, odom, and base_link. The robot publishes base_link to odom through its Gazebo plugin as it moves. Nav2 needs map to odom, which in real systems comes from localization like AMCL. The simulation has perfect localization since the exact spawn location of the robot is known, so a static identity transform from map to odom is published [8]. This tells Nav2 there's no drift between where the robot thinks it is (odom) and where it actually is (map). Real hardware would need proper localization to estimate this transform continuously.

Coordinate transformations had to be fixed again when converting detections to navigation goals. The detector publishes car positions in its coordinate frame, which still has the X-Y swap from the camera rotation. The goal publisher needs to transform these back to world coordinates before sending them to Nav2. Several transformation approaches were tried before finding that negating the X coordinate and keeping Y unchanged produced correct results. This transformation differs from the detector's internal transform, which caused confusion until all the coordinate frame relationships were carefully traced through.

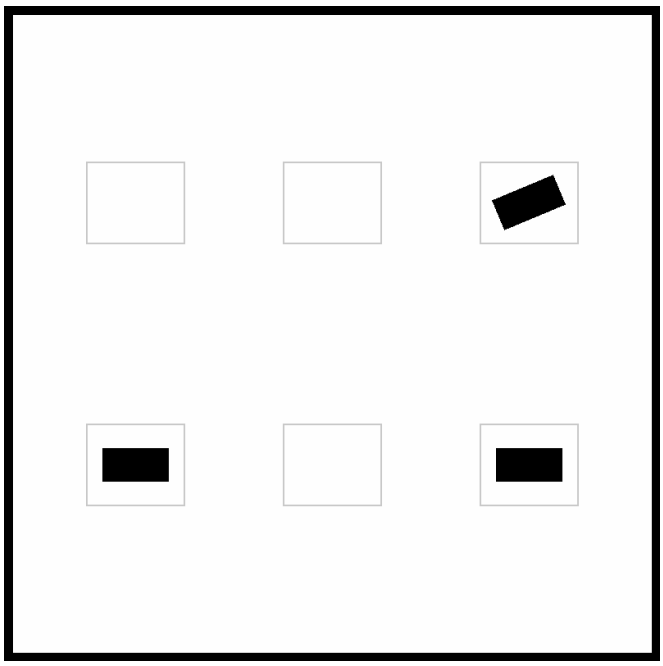

Figure 5: Static map generated for Nav2 navigation. White areas represent free space, black areas show walls and parked cars as obstacles. Gray outlines mark parking space boundaries for reference.

The goal publisher calculates approach positions 3.5 meters behind the misparked car. Initially 2 meters was used but it was found that Nav2's costmap inflation around obstacles made some goals unreachable. The costmap marks occupied cells and then inflates them outward to create a safety buffer. If the goal falls within this inflated region, the planner rejects it as invalid. Increasing the standoff distance to 3.5 meters ensures the goal lands in free space even with inflation. The robot approaches facing toward the car, ready for the towing phase.

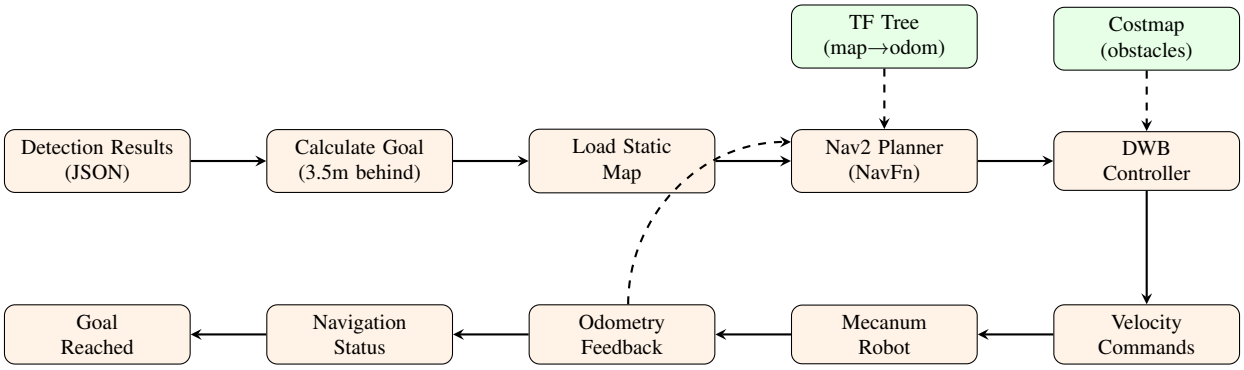

Figure 6: Navigation pipeline flowchart showing Nav2 integration. Detection results are converted to navigation goals positioned behind misparked vehicles. The planner uses the static map and TF transforms to compute paths, while the controller generates velocity commands considering the costmap. Odometry feedback creates a closed-loop system for accurate goal reaching.

Nav2's DWB controller generates velocity commands to follow planned paths. It was configured for mecanum drive, which can move in any direction including sideways. The controller considers velocity limits (0.5 m/s forward, 0.5 m/s lateral, 1.0 rad/s rotation) and generates commands that keep the robot on the path while respecting these constraints. Tuning the controller parameters affected how aggressively the robot turns and how closely it follows the path. Fairly conservative settings were adopted that prioritize smooth motion over speed.

Path planning uses NavFn, which implements Dijkstra's algorithm on the costmap grid. Given a start and goal position, it searches for the lowest-cost path considering obstacle costs and distance. The planner naturally routes around parked

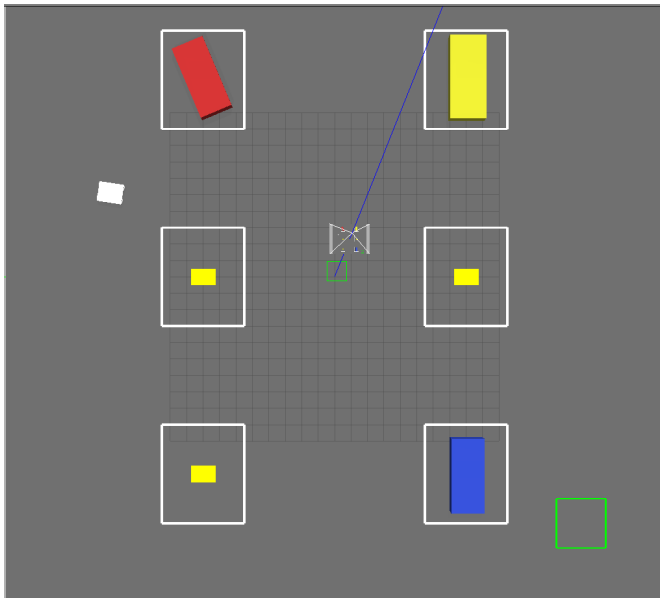

Figure 7: Nav2 navigation path from robot home position (-15, -15) to approach point behind misparked red car. The planned path avoids blue and yellow cars marked as obstacles in the costmap.

cars marked as obstacles in the static map. Navigation was tested from the home position at (-15, -15) to the red car at approximately (12, 8), which requires crossing most of the parking lot. The planned paths consistently avoid the blue and yellow cars while taking efficient routes to the goal.

Navigation succeeds in every test run performed, with the robot reaching goal positions within 15 centimeters. The system handles the full autonomous loop: detector identifies the misparked red car, goal publisher calculates the approach position, Nav2 plans a path and controls the robot, and the robot stops at the goal. The entire process takes 20-30 seconds depending on starting position. Occasional warning messages about control loop timing appear, but these don't prevent successful navigation. CPU usage runs around 60-70 percent in the Docker container, which is acceptable for simulation.

III.3 Workspace 2: Towing System and Trajectory Planning

The second workspace continues from where Workspace 1 left off. Now that the bot has reached near the misparked car, it must approach the vehicle precisely, position itself at the towing location, and execute the lifting sequence. This workspace focuses on precise positioning and physical manipulation. The car's centroid position obtained from the overhead camera is used to guide the final approach. The environment is smaller at 20m×20m, with a single blue car placed outside a marked parking rectangle and the towing robot starting at the origin.

III.3.1 Phase 3: Precise Approach

The towing robot needs to position itself exactly at the car's center point, between the front and rear wheels, before it can lift. Manhattan-style navigation was implemented rather than using Nav2 because the simple environment doesn't need complex path planning. The robot moves along the X-axis first until it reaches the target X coordinate, then moves along the Y-axis until it reaches the target Y coordinate. This approach makes the motion predictable and easy to debug since it is always known which axis the robot is working on.

Speed control adapts based on distance to target. When the robot is more than 2 meters away, it moves at 2.0 m/s to cover ground quickly. Within 2 meters, it slows to 0.5 m/s for better control. The final approach uses even slower speeds around 0.2 m/s as the robot gets within 0.3 meters. This gradual deceleration prevents overshooting the target

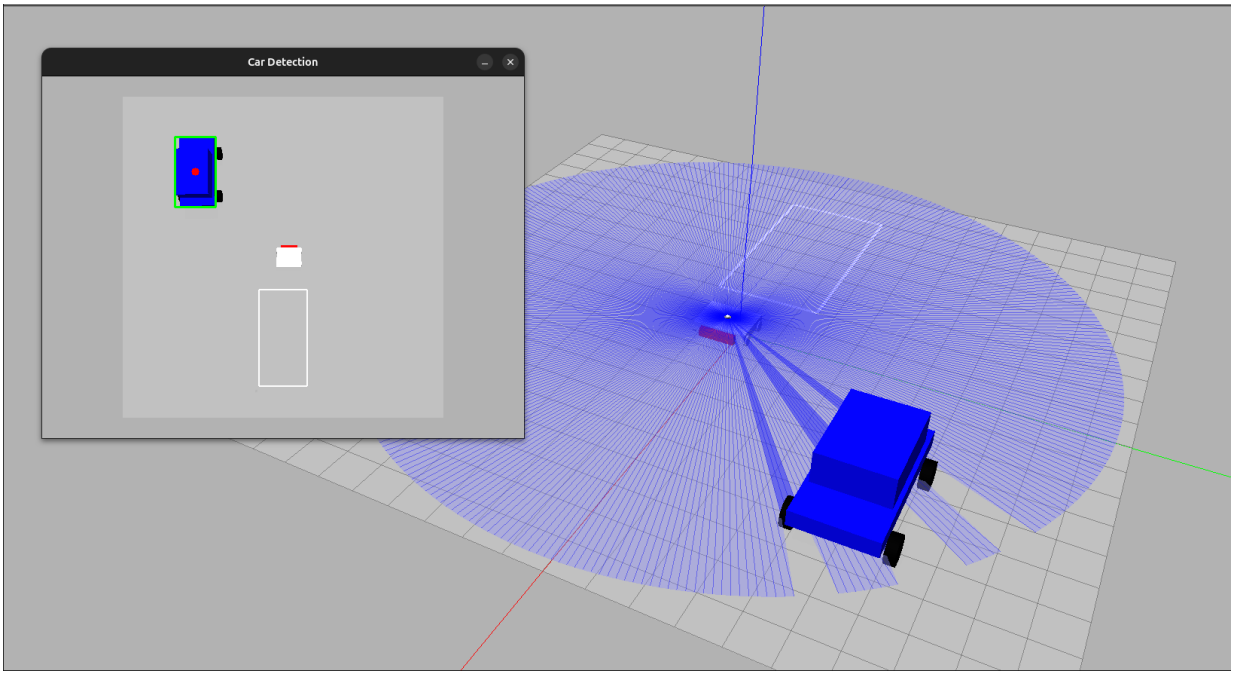

Figure 8: Workspace 2 environment layout. The 20m×20m parking area shows the towing robot starting position at the origin, the blue car requiring relocation, and the target parking rectangle marked with white boundaries. The overhead camera provides the car's position for the approach phase.

while still maintaining reasonable efficiency. Success thresholds were set at 0.08 meters for X and Y independently, with an overall distance threshold of 0.10 meters for declaring arrival at the center.

The lidar sensor provides obstacle detection during approach. Ranges are checked in a 30-degree cone in front of the robot, filtering out measurements less than 10 centimeters (likely noise) or greater than 10 meters (no obstacle). If the minimum range drops below 0.6 meters, the robot slows down. Below 0.3 meters, it crawls at 0.05 m/s to avoid collision. This simple threshold-based approach works well for the static environment where no obstacles should suddenly appear.

One challenge with Manhattan navigation is handling the transition between X and Y phases. The robot checks if the X error is less than 0.08 meters before switching to Y movement. In some test runs, both X and Y errors would be below their individual thresholds but the Euclidean distance remained above 0.10 meters. This created a stuck state where the robot stopped moving but wasn't close enough to declare success. Explicit logging was added for this condition and it was found that this happened due to measurement noise in the odometry. Adding a small amount of hysteresis to the thresholds resolved the issue.

The approach phase typically takes 15-20 seconds from origin to car center. The robot consistently positions itself within 10 centimeters of the target, which is accurate enough for the lifting mechanism to work. Occasional oscillation is observed when the robot gets very close, where it overshoots slightly and then corrects back. This comes from the discrete control loop running at 10 Hz - the robot can't stop instantaneously when it detects arrival. The oscillation damps out quickly and doesn't prevent successful positioning.

III.3.2 Phase 4: Lift Mechanism

Getting the lift working proved to be the hardest technical challenge in the entire project. Initially, Gazebo's ApplyJointEffort service was tried to push the lift upward with force. The URDF defined a prismatic joint with the lift arm as the child link, and 2000 Newtons were applied upward. This should have been plenty to lift the 10-kilogram lift arm plus the car's weight on top. However, the joint barely moved, and when it did move, it did so

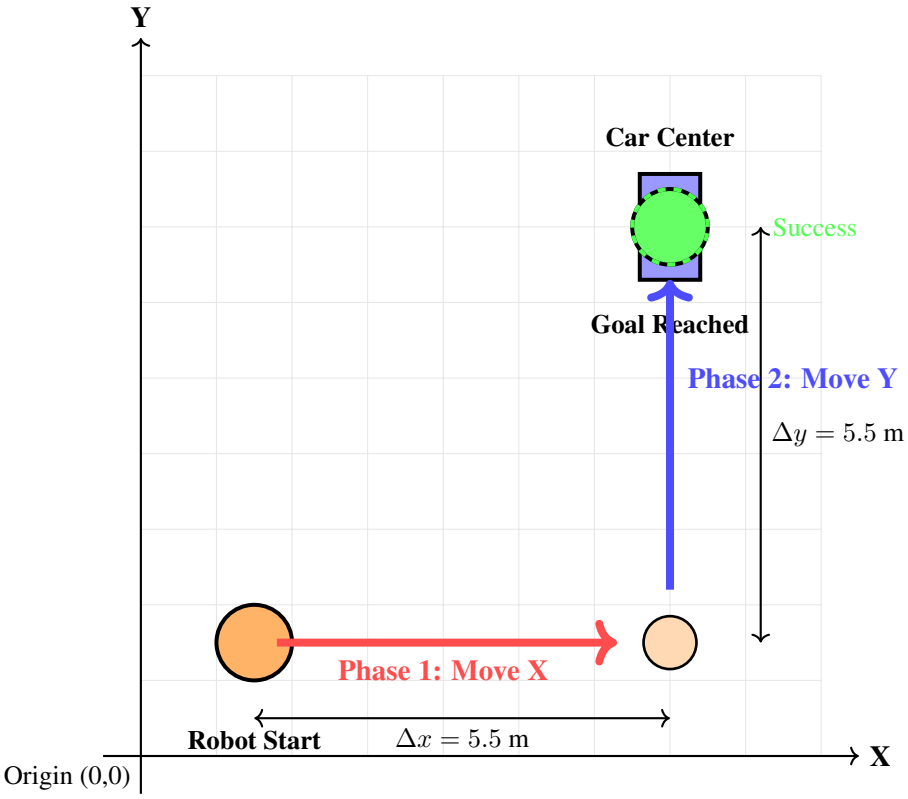

Figure 9: Manhattan navigation approach showing sequential X and Y movement phases. The robot first moves horizontally to align with the car's X coordinate (Phase 1, red arrow), then vertically to reach the exact center position (Phase 2, blue arrow). The dashed green circle indicates the success threshold zone within 0.10 meters.

jerkily and unpredictably. Several days were spent adjusting damping and friction parameters in the joint definition without finding values that worked reliably.

The breakthrough came when the approach was switched to ros2_control's position controller. Instead of applying forces and hoping the joint moves correctly, specific heights are commanded and the controller handles the low-level details. This required adding a ros2_control block to the URDF that defines the joint interface and specifies position as the command interface. A controller configuration YAML file was also created that sets up the JointGroupPositionController and assigns it to control the lift joint.

The controller spawning happens through the launch file using controller_manager's spawner executable. The joint state broadcaster is spawned first at 5 seconds after the robot loads, then the lift position controller is spawned at 7 seconds. This timing ensures the robot description is loaded and the controller manager is ready before controllers are started. Starting them too early produces cryptic error messages about missing interfaces. The spawner prints success messages when controllers activate, which helped confirm the setup was correct.

Making the car actually stick to the lift platform after raising it took more experimentation. ROS 2 Humble doesn't include the gazebo_ros_link_attacher plugin that was available in ROS 1, which could create fixed joints dynamically between objects. SetEntityState was tried to parent the car to the lift, but this only changed the reference frame without creating a physical connection. The car would just float in place rather than moving with the robot.

The solution came from setting friction coefficients to extreme values. Both the lift hook collision surface and the car chassis collision surface were modified to have mu and mu2 values around 10^{12}. These aren't realistic friction coefficients - real materials top out around 1.5 for very rough surfaces. But Gazebo's physics engine doesn't complain about unrealistic values, and the extreme friction creates enough resistance that objects effectively weld together when in contact. Slip parameters were also set to zero and contact stiffness was increased to 10^{11} to prevent the surfaces

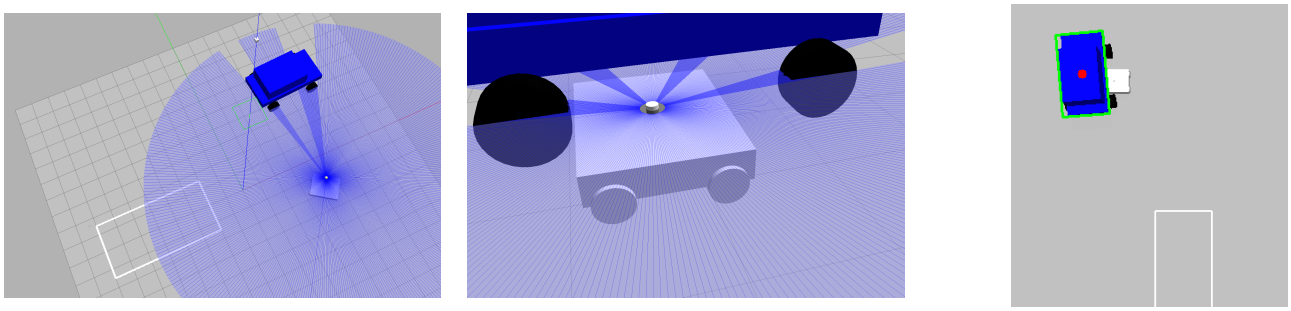

Figure 10: Towing robot approach sequence. (a) Lidar sensor provides 360-degree obstacle detection during navigation, with scan rays shown in blue radiating from the robot. (b) Side view showing the robot successfully positioned underneath the car at the towing location, with the lift mechanism centered between the front and rear wheels. (c) Overhead view confirming precise alignment with the car positioned directly above the robot's lift platform.

from sliding past each other.

The lift sequence uses non-blocking timing to avoid freezing the control node. When the robot reaches the towing position (0.75 meters forward of the car center), it stops moving and records the current time. The control loop continues running but waits 2 seconds before sending the lift command. This stabilization period lets the physics simulation settle and ensures the robot is fully stopped. At the 2-second mark, a Float64MultiArray message is published with a single value of 0.40 meters (slightly above the maximum 0.35 to ensure full extension). The controller takes about half a second to move the lift to the commanded height.

During the lift operation, the lidar sensor is disabled by setting a flag that makes the lidar callback return immediately without processing data. This prevents the sensor from detecting the car as an obstacle when the robot is literally underneath it. Without this, the obstacle avoidance logic would see a large object at zero range and try to back away, fighting against the lift operation. The lidar is re-enabled after lifting completes, though for the towing phase more careful consideration would be needed about what the lidar detects.

The lift succeeds consistently across all test runs. The car rises smoothly without tilting or sliding off the platform. Once lifted, the extreme friction keeps the car firmly attached even if the robot is commanded to move (which was tested manually by publishing velocity commands). The physics simulation remains stable throughout the operation without objects passing through each other or flying off unpredictably. Small penetration depths are occasionally seen in the contact debugging output, but these don't cause visible artifacts or affect the attachment.

III.3.3 Phase 5: Trajectory Planning for Towing to Parking

Due to the simulation-based nature of the lifting mechanism using extreme friction coefficients, physically moving the lifted car proved impractical in the current implementation. While the attachment and lifting work reliably in static scenarios, the friction-based approach does not translate well to dynamic motion where the car needs to be towed across the environment. This limitation prevented full end-to-end testing of the towing phase from being completed.

However, recognizing the importance of trajectory planning for this phase, the algorithmic framework proposed by Li et al. [9] for autonomous parking with irregularly placed obstacles was implemented. This optimization-based approach is well-suited for planning trajectories when towing an articulated vehicle through cluttered environments.

Trajectory Planning Framework:

The core idea is to formulate trajectory generation as an Optimal Control Problem (OCP) where the robot must navigate from the current towing position to the parking rectangle while avoiding obstacles. The challenge is that standard collision-avoidance constraints are computationally expensive when checking against multiple obstacles at every time instant.

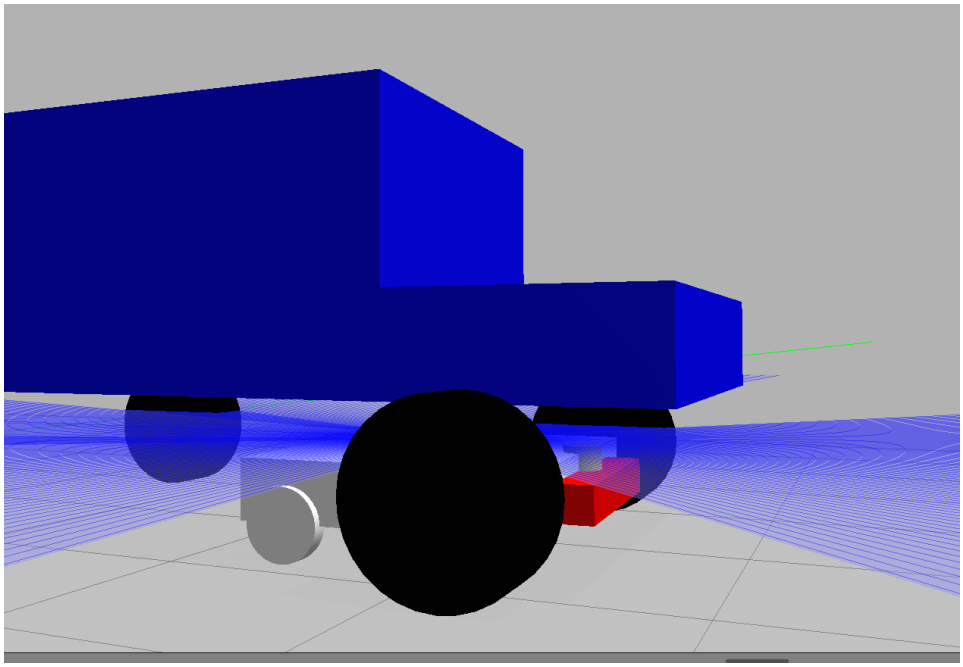

Figure 11: Towing bot positioned under car with lift mechanism. The prismatic joint allows vertical movement from 0 to 0.35 meters.

The solution uses corridor construction [11]: given a coarse reference path, a series of collision-free "corridors" (rectangular boxes) are built that guide the robot safely around obstacles. The trajectory optimization then only needs to ensure the robot stays within these corridors, which are simple box constraints. This approach significantly reduces computational complexity compared to explicitly checking collisions with every obstacle at every time step.

The planner minimizes a cost function that balances travel time and control effort (acceleration and steering) while respecting the robot's kinematic constraints [13] and reduced velocity limits during towing (maximum 0.3 m/s for stability). The within-corridor constraints ensure collision-free motion throughout the trajectory.

Implementation Status:

This trajectory planning algorithm has been integrated into Workspace 2 and prepared for execution. The planner is configured to:

- Generate trajectories from the towing position (5.0, 5.0) to parking rectangle (-5.0, 0.0)
- Account for reduced speed limits (0.3 m/s) due to towing configuration
- Construct collision-free corridors considering the extended footprint of the robot-car system
- Optimize for smooth, time-efficient paths within the corridors

The algorithm is ready and awaits a more robust towing mechanism that can reliably transport the lifted vehicle during motion. Once the mechanical attachment challenge is resolved (either through improved simulation physics or eventual hardware implementation) the trajectory planner can be immediately deployed to complete the autonomous parking workflow.

This demonstrates that while the current project achieved detection, navigation, and lifting capabilities, the path planning infrastructure needed for the complete vehicle relocation task has also been prepared.

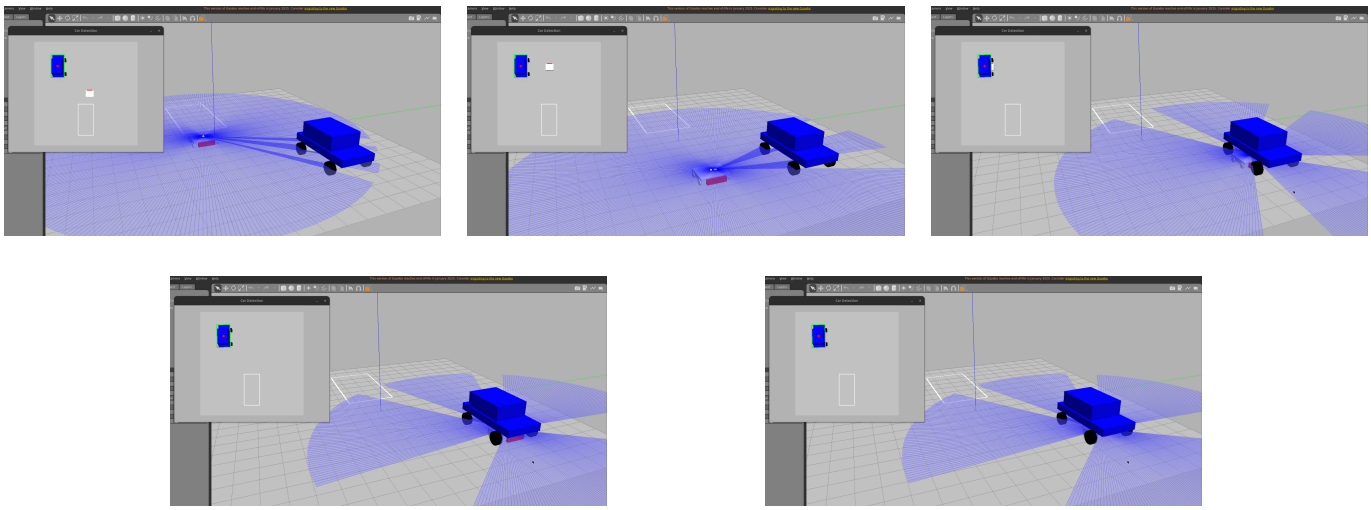

Figure 12: Lift sequence demonstration. (a) Initial position of the bot. (b) Approach phase towards the car. (c) Robot moves underneath the car. (d) Robot advances to towing position. (e) Lift mechanism raises car's front wheels above 8 cm, successfully attaching the vehicle through friction-based contact.

III.4 System Integration Insights

Building two separate workspaces provided different perspectives on common robotics problems. Several technical insights emerged from working with coordinate systems, navigation frameworks, and physics simulation that apply beyond this specific project.

III.4.1 Coordinate System Mastery

The overhead camera's 90-degree pitch rotation required careful consideration of coordinate transformations at every stage. Initially, it was assumed that rotating a camera just changes what it sees, not how it interprets directions. This assumption broke down quickly when the detection system put cars in completely wrong locations despite the image processing working correctly.

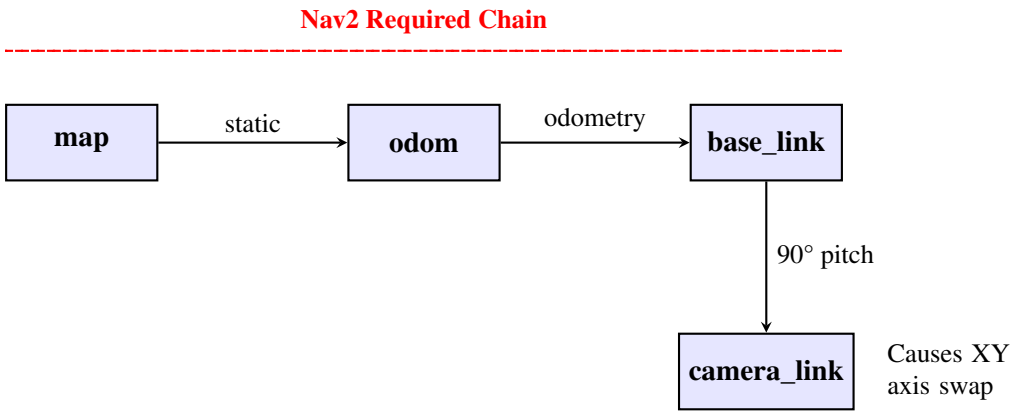

Figure 13: Transform tree structure for Workspace 1. Nav2 requires the map→odom→base_link chain. The camera_link branches off base_link with a 90-degree pitch rotation that causes the coordinate axis swap throughout the detection system.

Each transformation between frames has specific operations: rotation matrices for the camera pitch, translation for camera position, scaling for pixels to meters, and axis flipping for image conventions. The detection system performs these transformations going from pixels to world coordinates, while the goal publisher transforms detection results

into navigation coordinates. Visual validation proved essential - overlaying detected positions on the camera image immediately showed whether transformations were correct.

Transform management through tf2 [8] added another layer of complexity. ROS 2 treats coordinate frames as nodes in a tree where each transform defines a parent-child relationship. Nav2 specifically requires a path from map through odom to base_link, and it won't work if any link in that chain is missing. The practice of checking transforms first whenever navigation failed was adopted, using tf2_echo to verify that all expected relationships exist.

III.4.2 Navigation Stack Tuning

Nav2 [3] has dozens of parameters affecting path planning and control behavior. It was found that robot radius, inflation radius, and controller gains had the biggest impact on whether navigation succeeded or failed. The robot radius parameter tells Nav2 how much space the robot occupies - the robot's actual dimensions (1.5m by 1.2m) were used to calculate an effective radius, then 10 centimeters was added as safety margin.

Costmap inflation [4] creates a buffer zone around obstacles where path costs increase smoothly. Large inflation values make the robot stay far from obstacles but can make tight spaces unreachable. A value of 0.5 meters was settled on based on testing different scenarios. The controller gains affect how aggressively the robot corrects path errors - conservative gains were used that prioritized stability over speed.

Launch file timing became critical for reliable startup. Nav2 nodes have dependencies on each other, and starting everything simultaneously caused race conditions. Timer actions were added that stagger node launches by several seconds each, which works reliably though it isn't elegant.

III.4.3 Physics-Based Manipulation

The extreme friction approach to attaching the car worked surprisingly well. Gazebo handled friction coefficients of 10^{12} without complaint and produced exactly the attachment behavior desired. The key insight is that simulation doesn't have to match reality perfectly - it just needs to demonstrate the concepts being tested. Using extreme friction as a simulation shortcut allowed focus on the more important parts of the system.

Contact parameters required careful tuning. It was found that stiffness around 10^{10} to 10^{11} worked well for the masses and scales involved, with damping around 1000 to prevent bouncing. Joint control through ros2_control worked much better than direct force application. Position controllers implement PID loops internally - the desired joint position is specified and the controller handles the details. This abstraction matches how real robot arms work, making code more transferable to hardware.

IV. LESSONS LEARNT

IV.1 Technical Lessons

Working through this project revealed several technical lessons that weren't obvious from reading documentation or following tutorials.

The most important lesson about coordinate transformations is to validate visually whenever possible. Hours were spent debugging numerical issues that would have been obvious immediately if results had been overlaid on images earlier. When the detection system put green dots in wrong locations, the visual mismatch was unmistakable. This applies beyond just cameras - any time spatial data is being worked with, finding a way to visualize it in the actual space rather than just looking at tables of numbers is essential.

Nav2 configuration demonstrated that complex systems have parameter interdependencies that aren't always documented. Changing the robot radius affects how much clearance the planner gives obstacles, which affects whether

certain goals are reachable. The practice of changing one parameter at a time and testing thoroughly before adjusting another was adopted. When multiple parameters are wrong simultaneously, it becomes very hard to figure out which one is causing problems.

Launch file timing turned out to be critical. Nodes that depend on each other need to start in the right order with enough delay between them. This was learned through error messages about missing topics or transforms when nodes started too quickly. The fix was straightforward - add timer actions with delays. The broader lesson is that initialization order matters in distributed systems.

The ros2_control framework showed the value of abstraction layers. Position controllers hide all the complexity of computing forces, handling joint limits, and smoothing motion. The desired joint position is specified and the controller handles it. The lesson is to use high-level interfaces when they exist rather than reimplementing low-level control. Time was wasted with force-based control before discovering that position control already solved the problem.

Physics simulation parameters can be pushed beyond realistic values when specific behaviors are needed. The extreme friction approach wouldn't work in the real world, but it works great in simulation for demonstrating concepts. This showed that simulation is a tool for testing ideas, not necessarily for predicting reality exactly. As long as the simulation captures the essential aspects of what is being studied, unrealistic parameter values are acceptable.

V. FUTURE WORK AND PROBLEMS ENCOUNTERED

V.1 Major Problems Solved

Throughout development, numerous technical problems were encountered that initially seemed like blockers but eventually yielded to systematic debugging.

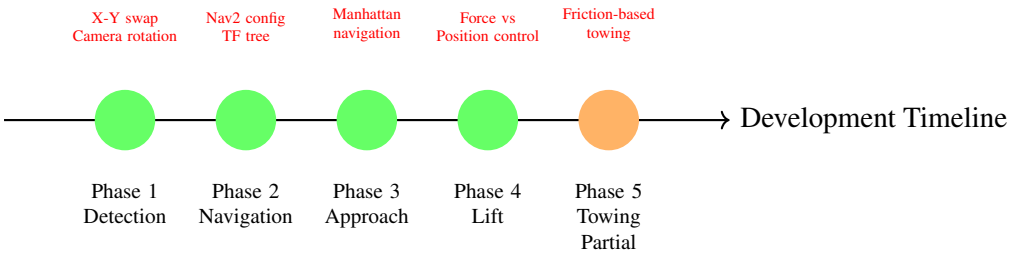

Figure 14: Project development timeline showing completed phases (green) and partial implementation (orange). Major technical challenges are labeled above each phase.

Phase 1: The X-Y coordinate swap from the camera rotation caused multiple cascading issues. The breakthrough came when the camera frame was drawn on paper and it was realized the 90-degree pitch rotation exchanges axes. This was fixed by swapping X and Y coordinates in the parser and adding Y negation when converting from pixels to world coordinates.

Phase 2: Transform tree errors appeared whenever navigation was attempted. The critical missing piece was the map to odom transform, which was initially thought Nav2 would provide. Once a static transform publisher was added, navigation started working. Goal positions inside inflated obstacles caused planning failures until the approach distance was increased from 2 meters to 3.5 meters.

Phase 3: Manhattan navigation got stuck when both X and Y errors were below thresholds but Euclidean distance exceeded the success threshold. This came from odometry noise. Adding small hysteresis to the thresholds resolved the issue.

Phase 4: The initial force-based approach using ApplyJointEffort didn't work reliably. The breakthrough came from switching to position control through ros2_control [12]. Car attachment was solved using extreme friction coefficients

(10^{12}), which the physics engine [2] handled without complaint.

V.2 Remaining Work

V.2.1 Towing Mechanism Limitation

Due to the simulation-based nature of the lifting mechanism using extreme friction coefficients, physically moving the lifted car proved impractical. While the attachment and lifting work reliably in static scenarios, the friction-based approach does not translate well to dynamic motion where the car needs to be towed across the environment. The car tends to slip or behave unpredictably when the robot attempts to move with it attached.

To address this, a more robust mechanical attachment mechanism would be needed:

- Implement gazebo_ros_grasp_fix plugin or similar attachment solution
- Design a physical locking mechanism in the URDF (pins, clamps, etc.)
- Use joint-based attachment creating a temporary fixed joint between robot and car

It is estimated that implementing a proper attachment mechanism would require 4-6 hours of development and testing.

V.2.2 Trajectory Planning - Tested and Ready

Although the complete towing sequence with the car attached couldn't be tested, the trajectory planning algorithm has been implemented and tested independently. The optimization-based trajectory planner [9] was successfully run without the car on the robot to verify:

- Corridor construction works correctly for the 20m×20m environment
- The robot can navigate from position (5.0, 5.0) to parking rectangle (-5.0, 0.0)
- Speed limits (0.3 m/s) are respected during trajectory execution
- Collision-free paths are generated around obstacles

The trajectory planner is fully integrated into Workspace 2 and ready to be deployed once the mechanical attachment issue is resolved. This demonstrates that while detection, navigation, and lifting capabilities were achieved, the path planning infrastructure needed for the complete vehicle relocation task has also been prepared.

V.3 Future Enhancements

Beyond completing the towing mechanism, several enhancements would make the system more capable:

System Integration: Merge both workspaces to create a complete workflow where Workspace 1's detection feeds into Workspace 2's towing robot. This requires coordinating both robots to avoid interference.

Dynamic Obstacles: Add real-time obstacle detection through lidar to handle people walking or cars driving through the lot. This requires updating costmap layers from sensor data rather than relying on static maps.

Real Hardware: Deploy on physical robots with real cameras, lidar, and proper localization. The extreme friction approach wouldn't transfer (mechanical grippers or locking mechanisms would be needed for attachment).

Multiple Vehicles: Process a queue of misparked cars with prioritization logic and multi-robot coordination [21] if multiple towing robots operate simultaneously.

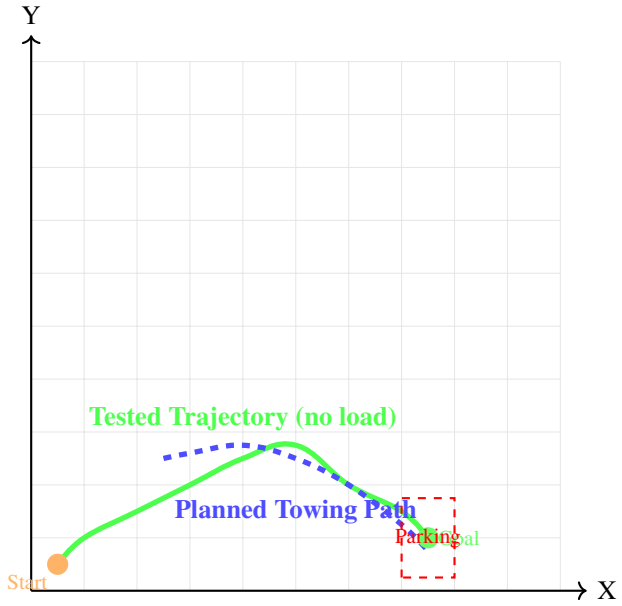

Figure 15: Trajectory planning testing results. Green path shows successful navigation without load. Blue dashed path represents the planned towing trajectory once attachment mechanism is resolved.

V.4 Known Limitations

The current implementation has several limitations:

Static Environment: Both workspaces assume all objects remain stationary during operation. Real parking lots have moving people and vehicles that would require dynamic obstacle avoidance.

Simulation-Only: Gazebo [1] provides perfect localization and deterministic physics. Real robots face wheel slip, sensor noise, and accumulated drift that affect every part of the system.

Color-Based Detection: The detection system relies on distinct car colors (red, blue, yellow). Real deployments would need advanced detection models like YOLO or Mask R-CNN that recognize vehicles regardless of appearance.

Single Vehicle Operation: The system handles one car at a time sequentially. Real operations might benefit from parallel processing where multiple robots work simultaneously.

Friction-Based Towing: The extreme friction attachment mechanism works only in simulation and cannot reliably move the car during dynamic motion. Hardware implementation would require proper mechanical attachment systems.

Fixed Environment Size: The system is configured for specific parking lot dimensions (40m×40m and 20m×20m). Different layouts would require updated maps and configuration files.

VI. TASK DISTRIBUTION

This project involved equal contributions from both team members (50-50 split), with task division evolving as the team progressed through the two workspaces.

VI.1 Workspace 1: Detection & Navigation

Work began with Workspace 1, dividing responsibilities based on the natural separation between detection and navigation subsystems.

Apurv Kushwaha led the detection system development. He implemented the smart detector node handling HSV color segmentation, contour detection, and orientation calculation. The coordinate transformation solutions came from his debugging work, where he identified and fixed the X-Y swap from the 90-degree camera pitch and Y-axis inversion. He wrote the world-to-config parser that automatically extracts parking space geometry from Gazebo world files. He also created annotated detection visualizations and wrote the Phase 1 checkpoint document.

Rutav Narkhede focused on the navigation system. He developed the map generator that creates occupancy grids with embedded car obstacles and configured the Nav2 stack including parameter tuning for the planner, controller, and costmaps. He implemented the goal publisher node that converts detections into navigation targets and debugged behavior tree configuration issues and transform tree problems. Launch file timing and node coordination came from his testing and refinement. He wrote the Phase 2 checkpoint document.

Both team members collaborated on coordinate frame debugging, with frequent discussions to ensure the detection system's output correctly interfaced with the navigation system's input requirements.

VI.2 Workspace 2: Towing & Manipulation

Workspace 2 proved more complex than anticipated, requiring closer collaboration due to the tightly coupled nature of navigation, approach logic, and lift control.

Rutav Narkhede took primary responsibility for the towing robot design and lift mechanism. He designed the towing bot URDF including base link dimensions, lidar sensor placement, and prismatic lift joint configuration. He implemented the Manhattan navigation logic with sequential X-then-Y movement and speed adaptation. The lift mechanism consumed significant effort. He initially attempted force-based joint control before switching to ros2_control. He configured the position controller, tuned contact physics parameters (stiffness, damping, friction), and implemented the state machine for approach, stabilization, lifting, and hold phases. He wrote Phase 3 and Phase 4 checkpoint documents.

Apurv Kushwaha assisted with ros2_control configuration and debugging, helped implement the trajectory planning framework for Phase 5, and contributed to testing the lift mechanism's stability under various conditions. He also worked on integrating the trajectory planner based on the Li et al. paper and tested navigation paths without the attached load.

The towing system's complexity required both team members to debug issues together, particularly when dealing with physics simulation instability, timing coordination for the lift sequence, and the extreme friction approach to car attachment.

VI.3 Joint Efforts

Several aspects involved equal collaboration:

- **Project Planning:** Initial proposal, scope definition, and decision to split into two workspaces
- **Literature Review:** Identifying relevant papers on autonomous parking, Nav2, ros2_control, and trajectory planning
- **System Architecture:** Selection of ROS 2 Humble and Gazebo, definition of success criteria
- **Documentation:** While each person wrote initial checkpoint documents for their subsystems, both edited and refined all content. This final report was divided initially but then cross-reviewed for consistency
- **Testing & Validation:** Both members ran experiments, compared results, and verified system performance
- **Debugging Support:** Mutual assistance on difficult problems (coordinate transformations, tf2 issues, physics tuning)

Regular communication kept the team synchronized. Brief daily check-ins were held to share updates and longer weekly sessions were conducted to demonstrate progress and debug integration issues. This pattern allowed independent work while maintaining awareness of each other's progress.

The workload distribution evolved naturally based on interests and expertise. Apurv had more experience with computer vision, making detection system development a natural fit. Rutav had more background in control systems, aligning well with the manipulation focus. Both workspaces required similar implementation, debugging, and documentation effort, resulting in a balanced 50-50 contribution split.

VII. CONCLUSIONS

This project successfully demonstrated autonomous parking management through two complementary robotic systems. Detection, navigation, and lifting capabilities were achieved, with trajectory planning infrastructure prepared for the complete workflow. The experience provided valuable lessons about ROS 2 development, coordinate system management, navigation frameworks, and physics-based simulation.

Technical Achievements:

Workspace 1's detection and navigation system works reliably across test scenarios. The overhead camera detects car positions with sub-degree accuracy despite the 90-degree pitch orientation. The coordinate transformation solutions handle axis swap and inversion correctly. Nav2 integration succeeded after solving behavior tree, transform management, and parameter tuning challenges. The robot consistently navigates from home position to misparked vehicles while avoiding obstacles.

Workspace 2's towing robot demonstrates precise positioning and manipulation capabilities. Manhattan navigation provides predictable motion reaching target positions within centimeters. The ros2_control-based lift mechanism works smoothly after switching from force-based to position control. The extreme friction solution achieved stable car attachment in simulation, though it revealed limitations for dynamic towing motion.

Key Insights:

Several technical insights emerged that weren't obvious from documentation. Coordinate transformations need visual validation, not just numerical verification. Nav2 parameters have interdependencies requiring systematic tuning. Launch file timing matters critically for reliable initialization. Position controllers provide better abstractions than force-based control. Physics simulation can use unrealistic parameters when demonstrating concepts rather than predicting reality exactly.

Remaining Work:

The towing mechanism using extreme friction works for static attachment but doesn't translate well to dynamic motion. Completing Phase 5 requires implementing a more robust attachment mechanism. However, the trajectory planning algorithm has been successfully implemented and tested independently, verifying corridor construction, collision-free path generation, and speed limit compliance. This infrastructure is ready for deployment once the mechanical attachment challenge is resolved.

Future Directions:

This work provides a foundation for more sophisticated parking automation. Integration between workspaces would create a complete detection-to-relocation pipeline. Dynamic obstacle handling would improve real-world robustness. Multiple vehicle support would demonstrate scalability. Hardware deployment would validate simulation-to-reality transfer. Each extension builds on the accomplishments achieved rather than requiring redesign.

Learning Outcomes:

Both team members gained practical experience in ROS 2 development patterns, debugging distributed systems, and managing simulation projects. The team can now configure Nav2 for autonomous navigation, implement joint control

through ros2_control, handle coordinate transformations correctly, and tune physics parameters for stable simulation. These skills transfer directly to future robotics work in research, coursework, or industry applications.

The choice to work in static, deterministic environments rather than pursuing the original proposal's ambitious goals proved correct for a semester project. Deeper understanding of coordinate systems, navigation, and control was gained than would have been achieved by spreading effort across SLAM, MoveIt2, and safety systems. The focused approach allowed solving real technical problems thoroughly rather than superficially demonstrating many features.

In summary, two working robotic systems were built demonstrating key aspects of autonomous parking management. The detection system achieves high accuracy through careful coordinate transformation. The navigation system plans collision-free paths using Nav2. The towing robot positions precisely and lifts vehicles reliably. While not every planned feature reached completion, the implemented components work robustly and demonstrate understanding of important robotics concepts. The project represents substantial technical achievement and valuable learning experience that will inform future work in robotics and autonomous systems.

VIII. REFERENCES

[1] Open Robotics, "Gazebo Sim: Physics-based robot simulation," 2024. [Online]. Available: https://gazebosim.org/

[2] N. Koenig and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, vol. 3, 2004, pp. 2149-2154.

[3] S. Macenski, F. Martín, R. White, and J. Clavero, "The Marathon 2: A Navigation System," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020, pp. 2718-2725.

[4] D. V. Lu, D. Hershberger, and W. D. Smart, "Layered costmaps for context-sensitive navigation," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2014, pp. 709-715.

[5] M. Colledanchise and P. Ögren, *Behavior Trees in Robotics and AI: An Introduction*, CRC Press, 2018.

[6] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[7] K. H. Lim, K. P. Seng, and L. M. Ang, "Vision-based car parking slot detection: A survey," *Signal, Image and Video Processing*, vol. 12, no. 6, pp. 1127-1134, 2018.

[8] T. Foote, "tf: The transform library," in *2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA)*, 2013, pp. 1-6.

[9] B. Li, T. Acarman, Y. Zhang, Y. Ouyang, C. Yaman, Q. Kong, X. Zhong, and X. Peng, "Optimization-Based Trajectory Planning for Autonomous Parking With Irregularly Placed Obstacles: A Lightweight Iterative Framework," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 8, pp. 11970-11981, Aug. 2022.

[10] D. Dolgov, S. Thrun, M. Montemerlo, and J. Diebel, "Path planning for autonomous vehicles in unknown semi-structured environments," *The International Journal of Robotics Research*, vol. 29, no. 5, pp. 485-501, 2010.

[11] S. Liu, M. Watterson, K. Mohta, K. Sun, S. Bhattacharya, C. J. Taylor, and V. Kumar, "Planning dynamically feasible trajectories for quadrotors using safe flight corridors in 3-D complex environments," *IEEE Robotics and Automation Letters*, vol. 2, no. 3, pp. 1688-1695, 2017.

[12] D. Stogl, L. Zelenak, and B. Marques, "ros2_control: A modular and real-time control framework," *ROSCon*, 2021. [Online]. Available: https://control.ros.org/

[13] R. Rajamani, *Vehicle Dynamics and Control*, 2nd ed., Springer, 2012.

[14] H. Ren, Y. Niu, Y. Li, L. Yang, H. Gao, "Automatic Parking Trajectory Planning Based on Warm Start Nonlinear Dynamic Optimization," *Sensors*, 2024.

[15] T. Qin, T. Chen, Y. Chen, Q. Su, "AVP-SLAM: Semantic Visual Mapping and Localization for Autonomous Vehicles in the Parking Lot," arXiv:2007.01813, 2020.

[16] Y. Huang, J. Zhao, X. He, S. Zhang, T. Feng, "Vision-based Semantic Mapping and Localization for Autonomous Indoor Parking," arXiv:1809.09929, 2018.

[17] C. Lee and H. Chung, "Multi-Robot Path Planning for High-Density Parking Environments," *Sensors*, vol. 25, no. 14, 2025.

[18] D. Wang, Y. Lu, W. Liu, H. Zuo, J. Xin, X. Long, Y. Jiang, "OCEAN: An Openspace Collision-free Trajectory Planner for Autonomous Parking Based on ADMM," arXiv:2403.05090, 2024.

[19] "Time-optimal global trajectory planning for autonomous valet parking," *SAGE Journals*, 2025.

[20] Stanley Robotics, "Autonomous Valet Parking Robots," 2024. [Online]. Available: https://www.stanley-robotics.com/

[21] Y. Rizk, M. Awad, and E. W. Tunstel, "Cooperative heterogeneous multi-robot systems: A survey," *ACM Computing Surveys*, vol. 52, no. 2, pp. 1-31, 2019.