Autonomous Obstacle-Avoiding Robotic Car

EMMA NGUYEN, University of California, San Diego, USA
YUWEI REN, University of California, San Diego, USA
SAMVATHNA EM, University of California, San Diego, USA
MOMINA HABIBI, University of California, San Diego, USA

Obstacle avoidance is essential in autonomous robotics for safe and reliable navigation. Traditional systems often rely on Light Detection and Ranging (LiDAR) or Simultaneous Localization and Mapping (SLAM), which can be expensive and computationally demanding. In this project, we developed a low-cost robotic car that uses a monocular camera and AprilTag landmarks for real-time obstacle detection. The system runs ROS 2 on a Qualcomm RB5 board paired with an mBot Mega platform. Our Minimum Viable Product (MVP) is navigating a 2m × 2m test space, detecting obstacles within 30cm, and responding within 500 milliseconds. These results demonstrate that effective obstacle avoidance is possible without relying on SLAM or depth sensors, making the system practical for low-power embedded applications.

CCS Concepts: • **Computing methodologies** → *Computer vision*; • **Computer systems organization** → **Robotics**; • **Hardware** → *Sensor devices and platforms*; • **Networks** → Cyber-physical networks.

Additional Key Words and Phrases: ROS 2, AprilTag, obstacle avoidance, visual perception, autonomous robotics, embedded systems.

1 Introduction

Autonomous robots are increasingly used in real-world applications from warehouse logistics and delivery services to household and industrial automation. A core requirement for these systems is the ability to navigate their environment and avoid obstacles in real time without human intervention. Whether it's a robot avoiding furniture in a living room or navigating a factory floor, quick and reliable environmental awareness is essential for safe operation[2].

Many advanced robots achieve this using LiDAR, infrared, ultrasonic sensors, or visual SLAM pipelines[8, 19]. While these methods are powerful, they are often too expensive or computationally demanding for small-scale or embedded platforms. This project explores a low-cost, lightweight alternative: using a single camera and visual markers (AprilTags) for real-time obstacle detection, targeting environments with limited processing power or budget.

Inspired by accessible consumer technologies like robot vacuums and delivery bots, our goal was to design and build an autonomous robotic car that can detect and avoid obstacles using just a monocular RGB camera and Inertial Measurement Unit (IMU) data[12, 18]. By focusing on vision-based sensing, we aimed to avoid the complexity of full SLAM while still demonstrating robust obstacle avoidance.

We built the system on the Qualcomm RB5 development board [13, 17] mounted on an mBot Mega platform [5], running Ubuntu 22.04 with ROS 2 Foxy. Our Minimum Viable Product (MVP) met three core requirements:

- Navigate within a 2 m × 2 m space.
- Detect AprilTag obstacles within 30 cm.
- React within 500 milliseconds.

We began with open-loop control by sending hardcoded ROS 2 commands and joystick inputs to validate motor behavior. Once stable movement was achieved, we transitioned to closed-loop control using visual feedback. The

Authors' Contact Information: Emma Nguyen, emn011@ucsd.edu, University of California, San Diego, La Jolla, California, USA; Yuwei Ren, yur003@ucsd.edu, University of California, San Diego, La Jolla, California, USA; Samvathna Em, saem@ucsd.edu, University of California, San Diego, La Jolla, California, USA; Momina Habibi, mohabibi@ucsd.edu, University of California, San Diego, La Jolla, California, USA.

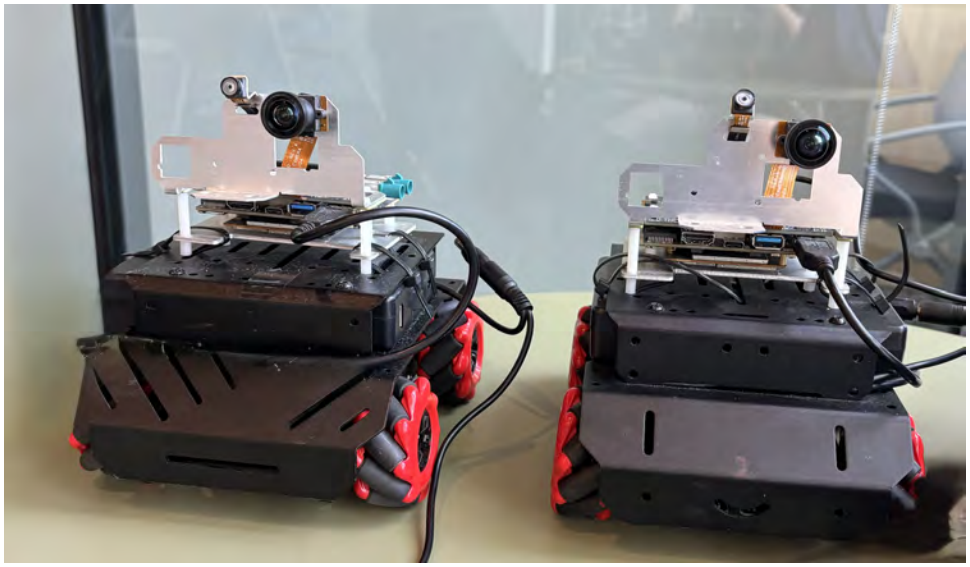

Fig. 1. Qualcomm RB5 integrated with the mBot Mega platform.

robot streamed its camera feed into ROS 2 using OpenCV and a Flask-based MJPEG server. We also used RViz2 to visualize sensor data and debug perception modules in real time.

Although we originally considered adding infrared or ultrasonic sensors, we decided to rely entirely on vision and IMU input to keep the system simple and low-power. We also explored integrating ORB-SLAM3 and RTAB-Map for full localization and mapping, but due to hardware constraints, we focused on AprilTag-based decision-making for obstacle detection.

To improve usability during testing, we added a web-based teleoperation interface built with Flask. It allowed users to manually control the robot, switch between auto and manual modes, and issue commands via browser-based directional buttons. These commands were published directly to the `/cmd_vel` ROS 2 topic.

Throughout the quarter, we followed a milestone-driven workflow. We used GitHub for version control, TeamGantt for scheduling [16], and Discord for communication. The team split into smaller sub-groups focused on specific tasks, allowing for more agile adjustments when technical issues arose.

Despite challenges with ROS 2 setup, camera integration, and node communication, we successfully met our MVP goals. The final system proves that robust, real-time obstacle avoidance is possible on an embedded platform using a purely vision-based approach. This project also lays the foundation for future enhancements such as SLAM-based localization and dynamic path planning using Nav2.

Key contributions of this project include:

- Designed and built a low-cost, vision-based autonomous robotic car using the mBot Mega platform and Qualcomm RB5 development board.
- Implemented open-loop and closed-loop control strategies using ROS 2 and AprilTag detection.
- Achieved real-time obstacle avoidance with a reaction time under 500 milliseconds for tags detected within 30 centimeters.
- Developed a ROS 2-compatible web-based teleoperation interface using Flask for manual and autonomous control.

- Integrated live camera streaming and visualization tools for monitoring and debugging, including OpenCV, MJPEG, and RViz2.
- Adjusted project scope to hardware limitations by choosing lightweight vision logic over full SLAM.
- Created a modular architecture ready for future integration with ORB-SLAM3 and Nav2 for enhanced autonomy.

2 Related Works

Autonomous obstacle-avoiding robots have been a long-standing topic in both academic research and industry, particularly in the domains of navigation, perception, and embedded control. This project draws inspiration from prior systems that demonstrate vision-based obstacle avoidance, sensor fusion, ROS 2 integration, and real-time behavior on constrained hardware.

2.1 Sensor-Based Obstacle Avoidance

Early work by Yılmaz and Özyer [19] and Nwokolo and Nwankwo [8] demonstrated reactive obstacle avoidance systems based on ultrasonic and IR sensors. These methods enabled basic navigation with minimal computational load. While our system ultimately excluded such sensors, these works helped us understand foundational reactive behaviors and early-stage tradeoffs.

2.2 Embedded Platforms and Architecture

Pandey et al. [12] developed a drone system that used AprilTags for localization in GPS-denied environments. Similarly, Volden et al. [18] used camera-IMU fusion and fiducial markers to dock surface vehicles. These works informed our use of AprilTags for indoor visual perception and reactive control. Lee et al. [15] explored how monocular vision and ROS-based systems could enable lightweight navigation, which aligns with our choice of tools. Patel et al. [20] also designed a minimal ROS 2 robot suitable for education, demonstrating that low-cost systems can perform real-time tasks with modest computing resources.

2.3 Vision-Based and AprilTag Navigation

The OpenCV library [11] enabled real-time image processing in our system. Using ROS 2 [9] and `apriltag_ros`, we extracted pose estimates for obstacle avoidance with only a monocular camera and no depth sensor. [14] demonstrated how AprilTag-based navigation systems could work reliably in structured indoor spaces using minimal hardware, validating our own use of AprilTags as lightweight markers.

2.4 Learning-Based and High-Speed Methods

Falanga et al. [4] achieved millisecond-level obstacle response using event cameras for drone control, while Alitappeh et al. [1] combined deep learning (LSTM, CNN) with sensor fusion for navigation tasks. Our system avoids such heavy computation to remain lightweight, but these studies supported our goal of real-time, vision-only response.

2.5 SLAM, Navigation, and ROS 2 Ecosystem

Although we experimented with ORB-SLAM3 and RTAB-Map, both proved computationally expensive to run in real time on the RB5. This limitation is consistent with findings in Bakhshalipour [2], where SLAM frameworks are shown to strain embedded resources. Shin and Park [6] specifically explored the difficulties of running RTAB-Map on edge devices, and their conclusions helped guide our decision to simplify. Nguyen et al. [7] also discussed limitations of monocular SLAM in indoor settings with limited depth cues. Nonetheless, the architecture and ROS 2 integration in these works served as a reference point for future work with `nav2` and path planning.

Community packages like `rtabmap_ros` and `nav2` [9, 10] helped structure our system. While we did not implement full path planning, classic algorithms like A* and Dijkstra's informed our thinking on map-based routing.

By referencing these works and building upon lightweight ROS 2-compatible approaches, our project aligns with the growing trend of designing affordable, camera-based robots for real-time decision-making in embedded environments.

3 Technical Material

Our autonomous robotic car combines two main components: the mBot Mega, which controls movement, and the Qualcomm RB5 development board, which handles sensing, perception, and decision-making (see Figure 2 and Figure 3).

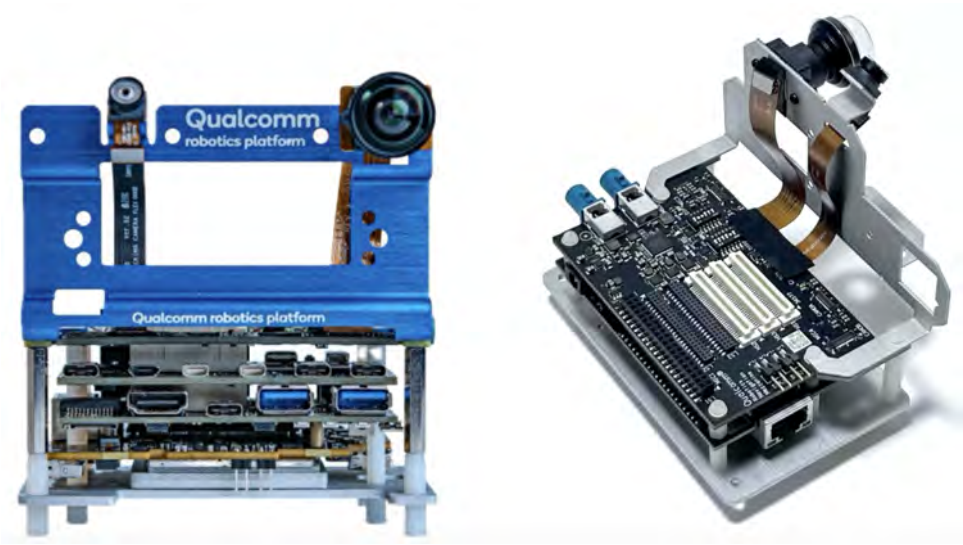

Fig. 2. Qualcomm RB5.

The system runs on Ubuntu 22.04 LTS and is built using the ROS 2 Foxy framework. The RB5 board uses its onboard camera and inertial measurement unit (IMU) to detect obstacles and understand the robot's position and orientation. For localization and navigation, it relies on AprilTag detection [14], which is fast, lightweight, and effective for indoor environments.

As shown in Figure 3, this architecture separates responsibilities: the mBot Mega executes motor commands, while the RB5 performs high-level perception and control. This division allows the robot to navigate a $2\,\text{m} \times 2\,\text{m}$ indoor space and respond to changes in under 500 milliseconds, enabling real-time autonomous behavior.
The system's core components and capabilities are summarized in Figure 4.

3.1 ROS 2 System Architecture

Nodes are the fundamental building blocks of a ROS 2 system. Each node is an independent process that performs a specific function, such as controlling the wheel motors, reading sensor data, or performing calculations. Nodes

Fig. 3. The fully assembled robotic car, with the RB5 board mounted on the mBot Mega base.

Component	Specification
Processing Unit	Qualcomm RB5 Development Board
Platform	mBot Mega
Operating System	Ubuntu 22.04 LTS
Framework	ROS 2 Foxy
Vision	RB5 Onboard Camera + AprilTag Detection
Control Interface	Web-based (Flask + HTML/CSS/JS)
Response Time	< 500ms
Navigation Area	2m × 2m indoor space

Fig. 4. Technical specifications of the robotic system.

are designed to be small and modular, allowing developers to break complex robotic systems into smaller, manageable parts that can run and communicate simultaneously.

Packages are how code and resources are organized. A package typically contains one or more nodes, along with configuration files, launch files, and any necessary dependencies. Packages make it easy to share, build,

and deploy ROS 2 projects, as the entire functionality for a particular feature or system component is grouped together in a single directory structure.

Publish/Subscribe is the most common communication mechanism in ROS 2. A node that generates data (such as a camera or a sensor) acts as a publisher, sending messages to a topic. Other nodes that need this data subscribe to the topic to receive the information in real time. This model supports flexible communication, where publishers and subscribers can share data without needing to be directly connected or aware of one another.

Services provide a request/response communication model. Unlike Publish/Subscribe, where data flows continuously, a service allows one node to send a request and wait for a reply from another node. Services are useful for tasks that require a response, such as commanding a robot to move to a specific position or querying system status.

A Topic is a communication channel that allows nodes to exchange data using a publish/subscribe system. Publishers send data to a topic, and subscribers receive that data and perform on the corresponding commands. This makes communication flexible and allows multiple nodes to easily share information.

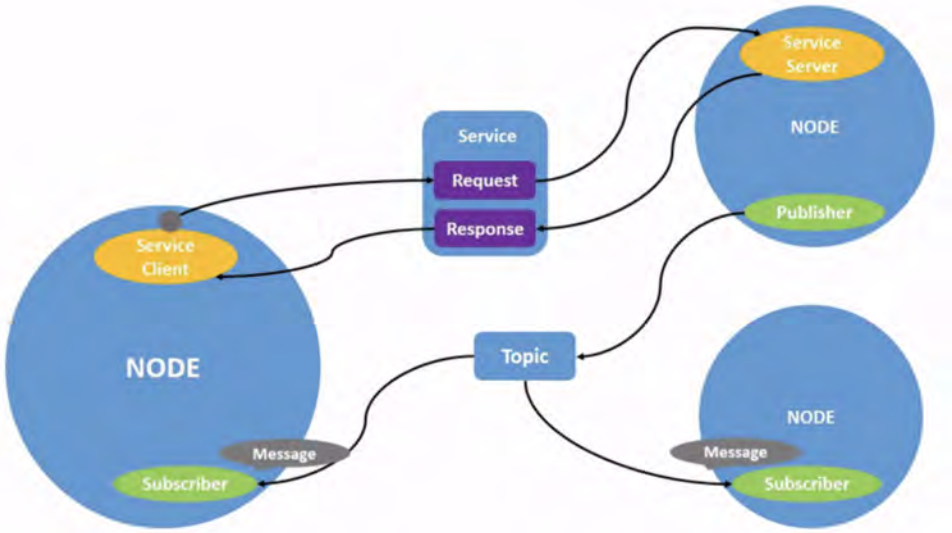

Fig. 5. ROS 2 System Architecture.

By combining nodes, packages, Publish/Subscribe messaging, and Services, developers can create scalable and efficient robot applications that can handle sensing, control, decision-making, and communication in a coordinated and reliable way. Hence, ROS 2 provides a flexible and modular framework for building complex robotic systems.

3.2 Camera Integration and Visualization

To monitor what the robot was seeing during development and testing, we used two methods. One was RViz2, a built-in tool in ROS 2, and the other was a custom web-based viewer using Flask. Both helped us check that the camera was publishing data correctly and allowed us to observe the live feed in real time.

RViz2

RViz2, which stands for ROS Visualization 2, is a tool that lets developers view data being sent across the ROS 2 system. It can show things like sensor readings, 3D models, and camera feeds. In our case, we used it to subscribe to the camera topic called `/camera_0/image`. This allowed us to see what the robot's camera was capturing, frame by frame, directly from the ROS 2 environment as shown in Figure 6.

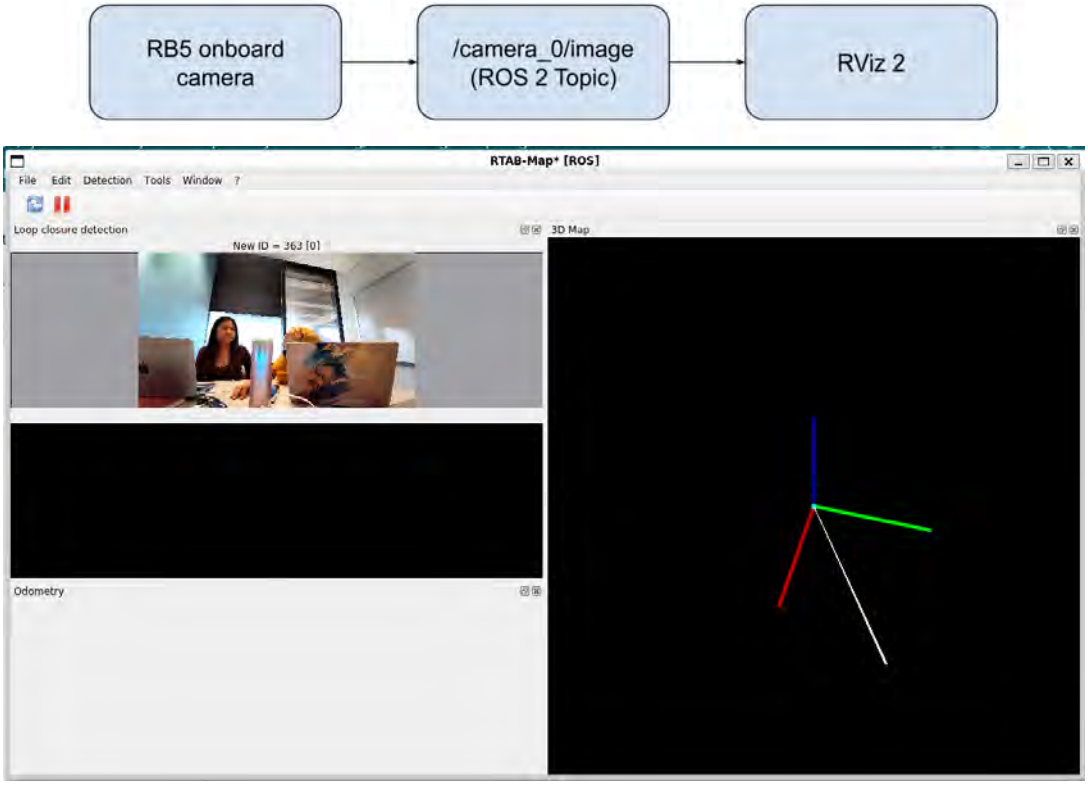

Fig. 6. ROS Visualization 2.

RViz2 was helpful in showing that the camera node was working as expected. We could confirm that the image topic was active and that the messages were being received correctly. This gave us a reliable way to make sure the connection between the camera and the system was stable and properly set up. RViz2 also gave us a wider view of the whole robot system. For example, it allowed us to check other data like orientation from the IMU or tags used for navigation. This made it a strong tool for debugging during early testing.

Flask Web Server

In addition to using RViz2, we also created a simple web interface using Flask. Flask is a lightweight web framework in Python that lets us build a live video feed that could be opened in any browser. This method worked by writing a subscriber node that listened to the same camera topic, `/camera_0/image`. We used OpenCV to decode the image stream, then sent each frame to the Flask server. From there, the frames were converted to MJPEG format and displayed on a web page as a live stream as shown in Figure 7.

Fig. 7. Flask Web Server.

This setup was especially helpful when we were not working directly on the robot. Since the video could be accessed through any browser on the same network, it made real-time monitoring easier for testing or demos. It also helped non-technical team members and viewers understand what the robot was seeing without having to install ROS tools.

Using both RViz2 and Flask gave us flexibility. RViz2 was more technical and helpful for full system debugging. The Flask viewer made the robot's vision more accessible and easier to share with others. Together, these tools helped us verify that the camera system was working and allowed us to monitor the robot's environment clearly during all stages of testing.

3.3 Manual Movement Control and Web Interface Integration

Before we introduced any sensors or autonomous features, we began by testing the robot's basic movement through what is called open loop control. This type of control system sends commands to the robot without checking for any feedback, a diagram shown in Figure 8. In simple terms, the robot is told to move, and it just does it—without asking questions like, "Did I bump into something?" or "Am I still moving in the right direction?"

Open loop control is often used in early-stage robotics because it allows us to isolate and test whether the hardware is functioning correctly. For example, it helps confirm whether the motors are wired properly, the motor driver is receiving power, and the communication between the main computer (in our case, the Qualcomm

RB5) and the motor controller (the mBot Mega) is working. There is no decision-making happening. The robot simply follows each command exactly as given.

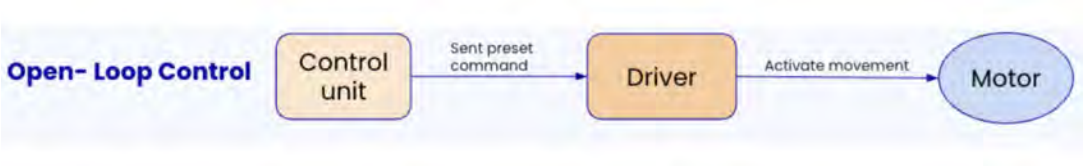

Fig. 8. Open Loop Control Diagram.

To implement this, we connected the RB5 board to the mBot Mega using a USB-to-serial connection. Using ROS 2, we published simple movement commands like "move forward," "turn left," or "stop." Since the robot was not receiving input from any sensors, it could not react to its surroundings or change its behavior based on what it encountered. For instance, if we sent a command to move forward, it would continue to do so until we explicitly told it to stop, even if there was an obstacle in the way.

To make manual control more accessible and user-friendly, we developed a custom web-based interface using Flask, a lightweight Python web framework. This browser interface allowed us to control the robot without needing a joystick or physical connection. From any device on the same network, we could open the control panel and click directional buttons like forward, backward, left, right, stop, or rotate.

Each button triggered a function on the Flask server that published a corresponding message to the ROS 2 topic `/cmd_vel`, which is the standard command velocity topic for movement in ROS-based robots. This allowed us to control the robot's motors in real time just by using a browser. We also included an "Auto" button in the interface. When clicked, this launched the ROS 2 nodes for AprilTag detection, which switched the robot from manual control to autonomous navigation. This gave us the ability to test both control modes without needing to restart the system.

Overall, this manual control system was essential during development. It helped us troubleshoot problems, test motor behavior, and provided a safety net during autonomous testing. Starting with open loop control gave us a simple but reliable base. Adding the Flask interface brought flexibility and visibility, which were both critical for moving into more complex behavior later on.

3.4 Closed loop Control: Obstacle Detection Using AprilTags

After successfully testing manual control, we moved on to autonomous behavior using closed loop control, a diagram shown in Figure 9. Unlike open loop systems, where the robot blindly follows commands, closed loop control uses feedback from sensors to make real-time decisions. This means the robot can see what is happening in its environment, evaluate the data, and adjust its movement on its own. In our case, the sensor feedback came from a vision-based system using AprilTags.

AprilTags are special square markers that look like simple black and white QR codes. Each tag has a unique ID, and when a camera sees one, it can calculate the tag's exact position and orientation in 3D space. This process is known as pose estimation. Pose refers to both the position and the angle of the tag relative to the robot. We used these tags as visual stand-ins for physical obstacles as shown in Figure 10. By placing them in known spots, we trained the robot to recognize when it was getting too close and then take action.

To implement this, we created a custom ROS 2 package called `ros2_april_detection`. This package uses the robot's onboard camera to detect AprilTags in the environment. When a tag was identified within a certain distance (roughly thirty centimeters), the system calculated its pose and published this data to a topic called

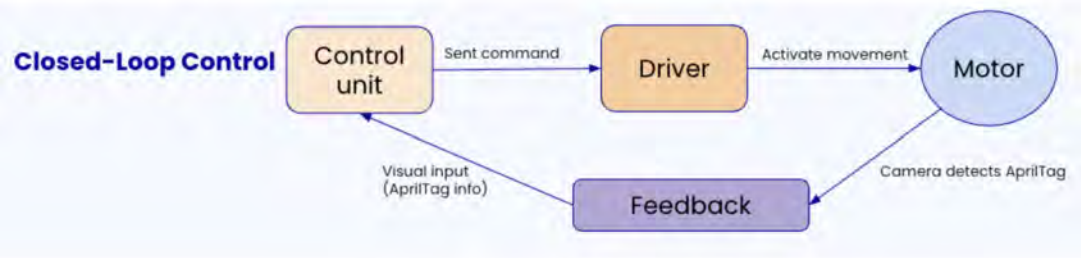

Fig. 9. Closed Loop Control Diagram.

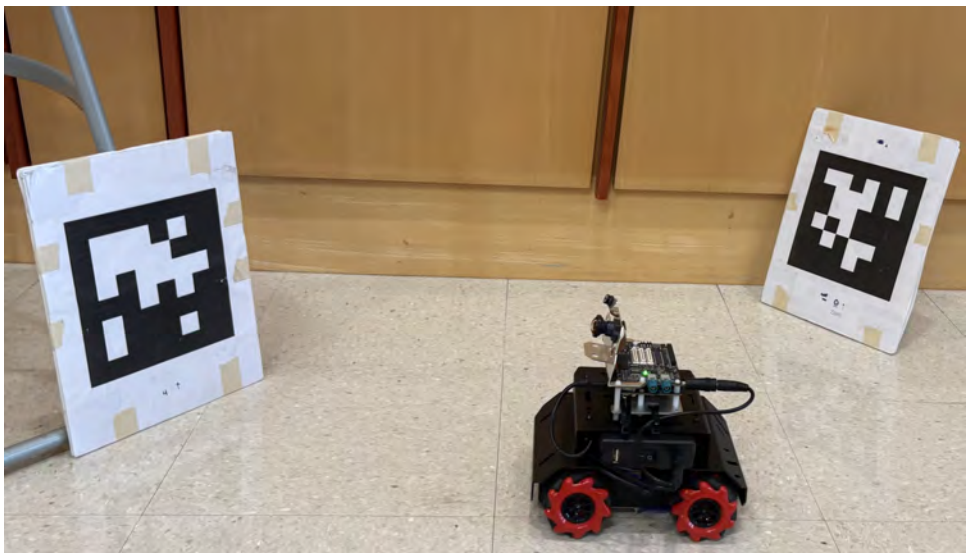

Fig. 10. Obstacle Detection Using AprilTags.

`/april_poses`. In ROS 2, a "topic" is like a communication channel where different parts of the robot's software can share information. Publishing to `/april_poses` made the tag's location available to other parts of the system in real time.

Another node in our system subscribed to `/april_poses` and also listened to the `/tf` topic. The `/tf` topic tracks the positions and orientations of all the robot's coordinate frames over time. Together, these two data sources helped the robot understand exactly where the tag was in relation to its own position. Based on that information, the node made decisions, such as stopping the robot or turning away from the obstacle, and sent commands through the `/cmd_vel` topic to change the robot's motion. This is how the feedback loop was formed: the robot looked at its surroundings, made a decision, and then acted on it.

We tested this system in a two meter by two meter indoor space. AprilTags were placed throughout the area, and the robot successfully detected them, identified them as obstacles, and responded appropriately. The reaction time was typically under half a second, which showed that the system could function in real time

without noticeable delay. This confirmed that a lightweight, vision-only system could be a viable solution for basic obstacle avoidance.

One of the biggest advantages of this approach was that it allowed us to avoid using traditional distance sensors like infrared or ultrasonic. Those types of sensors are often used in robotics to detect how close an object is, but they can be bulky or require extra wiring and hardware. By using only the camera and the robot's internal sensors like the IMU (inertial measurement unit), we kept the system simple and lightweight. This made it easier to run everything on the Qualcomm RB5 board without overloading the processor or requiring additional hardware.

3.5 Testing and System Integration

After we finished building the main parts of the system, including motor control, the camera connection, AprilTag detection, and the manual control interface, we brought everything together into a shared working environment. We used something called a ROS 2 workspace, which is a folder that holds all the software packages and tools the robot needs. This workspace helped us organize the project so that each part could communicate with the others properly.

To run all components at the same time, we used ROS 2 launch files. These are scripts that start multiple parts of the system in a single command. For example, instead of running the camera, the detection program, and the control logic separately, a launch file allowed us to start everything at once. This saved time and helped us test the system as a whole, rather than as disconnected pieces.

We tested the robot inside a small room and went through multiple rounds of adjustments. Each test gave us feedback about how the robot was behaving. We focused on improving how smoothly it moved, how sensitive it was when detecting AprilTags, and how quickly it reacted after detection. Some tests showed that the robot responded too slowly, or that it sometimes missed a tag completely. These issues helped us know what to change.

To observe what was happening, we connected to the robot remotely using SSH, which stands for Secure Shell. This tool let us access the robot's terminal from another laptop, which was helpful during movement tests. At the same time, we used a tool called RViz2. RViz2 gave us a live visual of the robot's surroundings. We could see the camera feed, the AprilTag positions, and how the robot planned to move. This helped us catch problems like delays in movement or missing data between different parts of the system.

We also used GitHub to manage our code. Every update was saved and labeled, which made it easier for the team to work at different times while staying coordinated. We also recorded videos and kept notes of our tests so we could review what worked and what needed fixing.

Testing everything together helped us understand how each part of the system interacted. It showed us where things needed improvement and gave us a clear view of what the robot could do. This process made our system more stable and dependable, and it also created a strong foundation for adding more features in the future, such as mapping the environment or creating a path planning system.

3.6 Future Plans for Mapping and Navigation

While our main goal was to create a working system for real time obstacle avoidance, we also explored more advanced techniques that would allow the robot to build a map of its environment and plan its own path. These features are part of what is known as SLAM, which stands for Simultaneous Localization and Mapping. SLAM is a method that allows robots to understand where they are in space while building a map of their surroundings at the same time.

To begin exploring this, we started testing a tool called ORB-SLAM3. This system uses input from the robot's camera and a built-in sensor called an IMU, which stands for Inertial Measurement Unit. The IMU helps the robot measure its own movement, such as rotation and acceleration. ORB-SLAM3 combines this data to estimate the robot's position using a technique called visual odometry. Visual odometry tracks how the robot moves by

comparing what the camera sees over time. In our early tests, ORB-SLAM3 showed encouraging results, but we found that the real time tracking performance was limited by the processing power available on the RB5 board. The system could estimate position well, but it struggled to keep up when running alongside other tasks.

At the same time, we experimented with a different approach using a package called RTAB-Map. This tool builds a two dimensional map of the environment using data from the camera, and it creates what is known as an occupancy grid. An occupancy grid is a simplified map that shows which areas are free to move through and which ones are blocked by obstacles. Our plan was to eventually connect RTAB-Map with a navigation system called Nav2. Nav2 is a ROS 2 framework that allows robots to plan a path from one location to another while avoiding obstacles in real time. This kind of dynamic path planning is more complex but allows for full autonomous navigation in large or changing environments.

However, due to multiple crashes and hardware limitations (lack of depth camera and embedded sensors), we decided to stay focused on building a simpler and more reliable vision-based system for this project. Our goal was to deliver a fully working robot with strong obstacle avoidance, so we chose not to fully deploy SLAM or Nav2 within our minimum viable product.

Even though we did not integrate these advanced tools completely, we designed our system in a way that makes future upgrades easier. The ROS 2 workspace is organized into modular parts, meaning each function—such as camera input, location tracking, and motion control—is separated into its own section. This makes it possible for future developers or teams to continue building on top of our work without needing to start over or rewrite major parts of the code. Whether the next step involves adding mapping features, creating full navigation plans, or expanding the environment size, the system is ready to grow.

4 Milestones

At the beginning of the quarter, we structured our project around weekly development goals to help us manage progress, respond to challenges, and stay on track. The timeline below summarizes our milestones, what we accomplished, and how we adapted as the project evolved.

4.1 Initial Milestones (Weeks 1 to 4)

- **Weeks 1–2**: Set up the mBot Mega platform and mounted the Qualcomm RB5 development board. Installed Ubuntu 22.04 and ROS 2 Foxy.
- **Week 3**: Verified basic motor control by sending ROS 2 commands through a USB-to-serial interface. Demonstrated open loop movement including forward, turn, and stop actions.
- **Week 4**: Performed motion tests using joystick-based teleoperation. Began setting up the camera stream and explored the feed through RViz2.

4.2 Technical Transition (Weeks 5 to 6)

- **Week 5**: Resolved camera configuration issues on the RB5 board. Used a GStreamer pipeline to stream live video. Switched from source build to binary installation of RTAB-Map due to hardware resource limits.
- **Week 6**: Successfully visualized the live camera feed via RViz2 using the `/camera_0/image` topic. Set up AprilTag detection with the `apriltag_ros` package and began testing real-time visual feedback for obstacle detection.

4.3 Minimum Viable Product Completion (Week 7)

- **Week 7**: Demonstrated the MVP. The robot successfully avoided obstacles using AprilTag detection in a 1.5 m by 1.5 m space, detecting tags within 30 cm and responding in under 500 milliseconds. Verified

closed loop control through live video feed and RViz2 visualization. Split the team into two sub-teams and revised the project schedule based on current progress.

4.4 Unachieved Milestones Due to Hardware Constraints

Despite several adjustments, the following goals were not met due to limitations in processing power on the RB5 platform:

ORB-SLAM3 Integration (Weeks 8–9):

- Failed to achieve stable pose estimation due to high CPU demand.
- Visual odometry experienced over 2 seconds of latency and produced inconsistent results.

RTAB-Map and Nav2 Path Planning (Week 10):

- RTAB-Map generated unstable and fragmented maps.

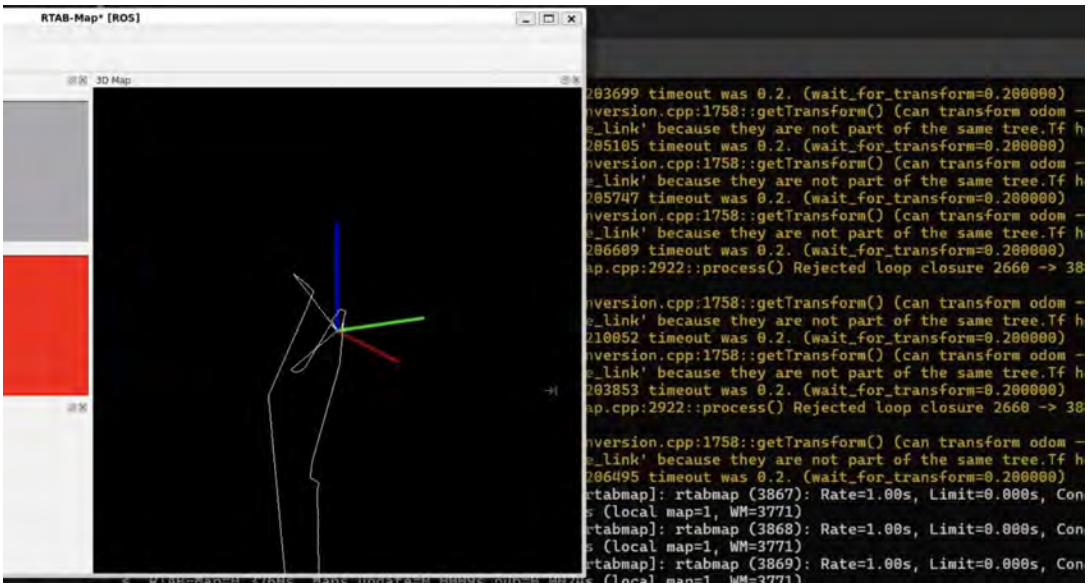

Fig. 11. Incomplete RTAB-Map Trajectory Caused by Pose Drift and Unstable Odometry.

- Nav2 path planning failed due to:
  - Invalid odometry input leading to localization errors.
  - Resource competition between simultaneous mapping and navigation nodes.
- The RB5 platform lacked the resources to run SLAM, mapping, and path planning in parallel.

4.5 Final Phase (Planned Weeks 8 to 10)

- **Week 8**: Refined AprilTag logic by adjusting obstacle detection thresholds using the Z-axis distance in the camera frame. Designed a layout with multiple tags in a 2 m by 2 m space to simulate path following.
- **Week 9**: Prioritized final MVP polishing over continued SLAM testing. Final deliverables included:
  - A demo video showing robust AprilTag-based obstacle avoidance.
  - A ROS 2-compatible browser interface for manual control.
  - The oral final presentation.

  - A GitHub-hosted project website and documentation.
- **Week 10**:
  - Final project video.
  - Final written report submission.

4.6 Challenges Faced and How They Were Addressed

ROS 2 Setup on RB5: Faced compatibility issues and minimal official documentation. Solved through TA support and community forums.

Camera Integration: Required extensive testing of video streaming. Resolved using OpenCV with GStreamer pipelines and ROS 2 topic remapping.

RTAB-Map Installation: Source build failed on RB5 due to limited system resources. Switched to a binary installation using apt.

SLAM Limitations: Full SLAM proved too heavy for the RB5. We chose to use AprilTags as lightweight visual landmarks instead.

Time Constraints: Some advanced features such as dynamic path planning and full web integration were not completed. However, the core MVP was finished, tested, and demonstrated.

4.7 Reflection

This milestone-driven approach helped us stay flexible and focused. By documenting progress, adjusting goals when needed, and working collaboratively, we were able to deliver a functioning autonomous robot. The final result met our course objectives and created a strong foundation for future improvements in mapping, navigation, and system expansion.

5 Conclusion

This project successfully demonstrated that a low-cost, vision-based autonomous robotic car can navigate and avoid obstacles in real time. Using the mBot Mega platform and the Qualcomm RB5 development board, we built a system that runs ROS 2 on Ubuntu 22.04 and processes live camera input through OpenCV to detect AprilTags acting as obstacle markers.

Alongside the autonomous control system, we also developed a remote-control mode through a ROS 2 web-based teleoperation interface. This allowed us to directly control the mBot over Wi-Fi, sending movement commands and receiving real-time feedback using ROS 2 publishers and subscribers. This teleoperation mode played a key role in system testing and debugging as we refined the robot's control logic.

We set a clear Minimum Viable Product (MVP) for the system, the ability to navigate within a 2m × 2m space, detect obstacles within 30cm, and respond within 500 milliseconds. We successfully met this target, achieving reliable closed-loop obstacle avoidance using just the RB5's onboard monocular camera and IMU. The system's performance was validated through live demonstrations, video recordings, and RViz2 visualizations.

Although more advanced features like ORB-SLAM3 mapping, Nav2 path planning, and a web dashboard were not fully integrated by the MVP milestone, the project laid a strong foundation for future improvements. These could include dynamic SLAM integration, multi-goal navigation, and more advanced decision-making using sensor fusion. Overall, this project gave us valuable hands-on experience working with embedded robotics, computer vision, ROS 2 development, and collaborative system design, and showed that effective autonomous behavior can be achieved even with modest hardware and careful, iterative development.

Acknowledgments

We would like to thank UC San Diego CSE 145 for course support and guidance and UC San Diego CSE 276A for providing the RB5 ROS 2 starter code developed by the Autonomous Vehicle Laboratory [3]. We also acknowledge the ROS 2 community for the ROS 2 framework and package ecosystem and the AprilTag Library for enabling the vision-based detection system. We are especially grateful to Rohan Patil, Computer Science PhD Student at UC San Diego, for his continuous support, guidance, and valuable advice throughout the project. We also sincerely thank Julian Raheems, Computer Science and Engineering PhD Candidate at UC San Diego, for his time, effort, and insightful feedback that helped us stay on track and successfully complete this project. Finally, we acknowledge the use of generative AI tools, including ChatGPT (OpenAI), to assist with phrasing, writing suggestions, and language editing during the preparation of this report.

References

[1] R. J. Alitappeh, M. N. Homayouni, M. R. Gholami, and M. Kiani. 2024. Autonomous robot navigation: Deep learning approaches for line following and obstacle avoidance. In *Proceedings of the 20th CSI International Symposium on Artificial Intelligence and Signal Processing (AISP)*. IEEE.

[2] Mohammad Bakhshalipour. 2024. *Bridging Real-Time Robotics and Computer Architecture.* Ph.D. dissertation. Carnegie Mellon University. https://kilthub.cmu.edu/articles/thesis/Bridging_Real-Time_Robotics_and_Computer_Architecture/25705374

[3] Henrik Christensen. 2024. CSE 276A – Introduction to Robotics. UC San Diego Podcast. https://podcast.ucsd.edu/watch/fa24/cse276a_a00

[4] D. Falanga, K. Kleber, and D. Scaramuzza. 2020. Dynamic obstacle avoidance for quadrotors with event cameras. *Science Robotics* 5, 40 (2020), eaaz9712.

[5] W. Jo, J. Kim, R. Wang, J. Pan, R. K. Senthilkumaran, and B.-C. Min. 2022. SMARTmBOT: A ROS2-based low-cost and open-source mobile robot platform. *arXiv preprint arXiv:2203.08903* (2022).

[6] Mathieu Labbé and François Michaud. 2018. RTAB-Map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation. *Journal of Field Robotics* 36, 2 (Oct. 2018), 416–446. doi:10.1002/rob.21831

[7] Thien Ng and LuckyBird. 2019. Indoor non-GPS flight using AprilTags (ROS-based). ArduPilot Discourse. https://discuss.ardupilot.org/t/indoor-non-gps-flight-using-apriltags-ros-based/42878 Accessed: 2025-06-12.

[8] P. Nwokolo and I. N. Vincent. 2019. Autonomous robotic car with obstacle detection and avoidance. *International Journal of Scientific & Engineering Research* 10, 6 (2019), 1055–1060.

[9] Open Robotics. [n. d.]. ROS 2 Documentation: Foxy. https://docs.ros.org/en/foxy/index.html. Accessed: 2025-06-12.

[10] Open Robotics. [n. d.]. ROS 2 Tutorials — Foxy. https://docs.ros.org/en/foxy/Tutorials.html. Accessed: 2025-06-12.

[11] OpenCV Contributors. [n. d.]. Open Source Computer Vision Library. https://opencv.org/. Accessed: 2025-06-12.

[12] S. Pandey, C. Verma, E. Trivedi, and N. Mehendale. 2024. *AprilTag-Based Self-Localization for Drones in Indoor Environments.* Technical Report. SSRN. SSRN 4815151.

[13] Qualcomm Developer. 2023. AI Developer Workflow and Qualcomm® Robotics RB5 Bringup. YouTube. https://www.youtube.com/watch?v=4aIM2wul9co

[14] Tristan Schuler and Greta Studier. 2021. AprilNav: Indoor real-time landmark navigation using AprilTags. GitHub. https://github.com/nasa/AprilNav Accessed: 2025-06-12.

[15] Stephen Stephendade. 2021. AprilMAV: Using a RaspberryPi for Indoor Robot Navigation. rpanion.com blog. https://www.rpanion.com/blog/2021/05/27/aprilmav-using-a-raspberry-pi-for-indoor-robot-navigation/ Accessed: 2025-06-12.

[16] TeamGantt. [n. d.]. TeamGantt Online Project Management Tool. Online. https://app.teamgantt.com Accessed: 2025-06-12.

[17] Thundercomm. [n. d.]. Qualcomm® Robotics RB5 Development Kit. https://www.thundercomm.com/product/qualcomm-robotics-rb5-development-kit/. Accessed: 2025-06-12.

[18] Ø. Volden, A. Stahl, and T. I. Fossen. 2023. Development and experimental validation of visual-inertial navigation for auto-docking of unmanned surface vehicles. *IEEE Access* 11 (2023), 45688–45710.

[19] E. Yılmaz and S. T. Özyer. 2019. Remote and autonomous controlled robotic car based on Arduino with real-time obstacle detection and avoidance. In *Proceedings of the 7th International Conference on Electrical and Electronics Engineering (ELECO)*.

[20] Roberto Zegers. 2023. apriltag_robot_pose: Robot localization using AprilTag markers. GitHub. https://github.com/rfzeg/apriltag_robot_pose Accessed: 2025-06-12.