TORI: Tour-Optimized Robotic Intelligence

A Major Qualifying Project (MQP) Report
Submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements
for the Degree of Bachelor of Science in

Robotics Engineering,
Computer Science

By:

Jacob Ellington
Aashi Goel
Sukriti Kushwaha
Vivek Voleti

Project Advisors:

Professor Nitin Sanket
Professor Greg Lewin
Professor Jing Xiao
Professor Fiona Yuan

Date: April 2025

Abstract

Tori is an intelligent robotic agent designed to be an autonomous tour guide robot of WPI's Unity Hall. Using its multi-modal sensing and advanced reasoning capabilities, Tori enhances WPI visitor engagement by guiding users to their desired destinations in Unity Hall on request. The navigation system dynamically adjusts the robot's path to avoid obstacles and plan safe routes. A voice command system and touchscreen GUI allow for seamless and engaging human-robot interaction methods. In addition, an on-board Large Language Model (LLM) allows users to ask Tori for information about WPI's campus or programs and receive accurate and informative responses. Tori traverses all five floors of Unity Hall, complete with custom-trained vision pipelines to allow her to use the elevators.

Acknowledgements

- Our advisors: Professor Sanket, Professor Xiao, Professor Lewin, and Professor Yuan
- FRC Team 716
- PEAR lab
- WPI Department of Robotics Engineering
- Mike Ellington and Andy Brockway
- Unity Hall custodial staff
- Joana Ripa, Fatimah Daffaie, and Thea Caplan

Contents

List of Figures

1 Introduction and Background

Tori (Tour-Optimized Robotic Intelligence) is an autonomous tour guide robot designed to enhance visitor engagement within Worcester Polytechnic Institute's Unity Hall. Drawing inspiration from a 2019 MQP that explored similar objectives in Gateway Park, our team aimed to modernize the concept using state-of-the-art hardware and software. Tori leverages multi-modal sensing, speech and touchscreen interfaces, computer vision, and advanced reasoning to provide intelligent, human-friendly interactions. She is capable of navigating all five floors of Unity Hall autonomously, including riding elevators and responding to verbal commands or queries about WPI.

This report documents the methodology, implementation, and testing of Tori's subsystems, including the navigation stack, computer vision pipelines, human-robot interaction interfaces, control systems, and mechanical and electrical components. Our approach prioritized modularity, safety, and extensibility, with an emphasis on using open-source frameworks such as ROS 2 and Micro-ROS. Through the integration of vision-based button detection, multi-floor map management, and a local LLM-backed question-answering system, we demonstrate a practical, deployable solution for autonomous indoor guidance.

1.1 Project Inspiration

Our project was heavily inspired by the 2019 Robot Tour Guide MQP completed in collaboration with Ava Robotics. In their project, the team created a friendly tour guide robot designed to assist visitors in Gateway Park, one of WPI's academic buildings. Their robot, standing nearly four feet in height, used facial expressions to convey its emotions to users, such as happiness, confusion, and sadness. It communicated to users if it needed assistance and created the best path to navigate to different locations to provide an efficient tour of the building. For hardware, the team used a NVIDIA Jetson TX2 Module, touchscreen display, Logitech C270 Webcam, and more. Our goal was to improve and "modernize" their project by using newer technology and hardware, as well as working in Unity Hall, WPI's newest academic building, for greater exposure to students and visitors.

By creating an accessible navigation assistant, we enhanced the visitor experience in Unity Hall and made the space more welcoming for everyone. Our robot named Tori, short for "tour guide," stands 4.5 feet in height and uses an FRC robot base with a display near the top to act as a "head." Similar to the previous MQP team, we incorporated concepts of speech recognition, navigation, human-robot interaction, and computer vision. We further expanded the project by adding button-pushing and elevator capabilities

using a lift that moves up and down to reach different buttons on the panel. Our hardware included two NVIDIA Jetson Nanos, a Slamtec RPLIDAR S1 360° laser scanner, Logitech C270 Webcam, Lrtzcbi 14 inch Touch Screen Monitor, amongst others. In the future, we hope this project can be deployed to WPI on a larger scale.

1.2 Robot Operating System 2 (ROS 2)

ROS 2 [1] is a collection of open-source robotic middleware and tools. ROS allows access to algorithms widely used in robotics as well as the ability to tailor existing solutions for custom applications. A ROS Graph is a network of nodes in a given ROS system and the method of connection by which they communicate. Messages are passed between ROS processes using publish and subscribe mechanisms. Topics are used to share data streams known as messages between nodes. Messages are of a pre-defined type.

A node is a member of the ROS graph that uses a client library to communicate to other nodes. Each node is typically responsible for one task. Nodes can publish or subscribe to topics and may reside in the same process, different processes, or on different machines. They can also function as a client that requests services from other nodes, as shown in Figure 1. Conversely, they may function as a server to perform computations for other nodes. For longer running tasks, a node may act as an action client to request another node to perform computations on their behalf. They may also serve as an action server to provide service to client nodes.

Nodes are automatically connected through a distributed discovery process. When a ROS node is started, it advertises its availability to other nodes on the network with the same domain. Nodes respond to the advertisement with information necessary to connect to the new node. ROS nodes continuously advertise their information to discover other new entities that may connect on the network. ROS software is shared using packages. One package is typically developed for a singular task and may contain one or more nodes.

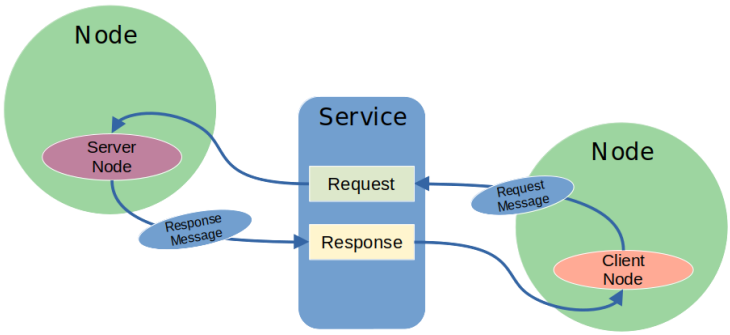

Figure 1: ROS 2 architecture

1.3 Adaptive Monte Carlo Localization (AMCL)

Adaptive Monte Carlo Localization (AMCL) is an algorithm to localize a robot using a particle filter on a 2d map [2]. Given a map of the environment, the algorithm estimates the position and orientation (pose) of the robot with respect to the map. The particle filter represents a distribution of predicted states, with each particle representing a potential state. The algorithm initializes the particles uniformly over the configuration space. As the robot moves around and receives laser scans of the environment, particles are sampled using Bayesian prediction. Particles that match more closely with the robot scans are weighted higher. Eventually, the particles converge to the robot's true pose. In our project, we were tasked with continually updating Tori's pose as she traversed across floors in Unity Hall.

1.4 Vision System

Intelligent robotic agents like Tori typically contain RGB cameras as a part of their sensor suite, allowing the robot to perform frame-by-frame vision processing on its environment. Tori, in particular, needs to be able to visually interpret the Unity Hall elevator buttons and doors to enable multi-floor navigation. While there are a multitude of methods for computer vision (CV), in this section, we explore a few relevant strategies for the robot to quickly, accurately, and efficiently interpret its environment.

1.4.1 Basic Computer Vision Pipeline Techniques

In situations where the desired vision pipeline is not particularly complex and should not be very compute intensive, manually designed rudimentary computer vision algorithms can be useful, since they are quick to implement and have a short runtime using only CPU compute. OpenCV [3], a widely used open-source CV library, provides tools for developers to construct their own custom vision pipelines.

Color thresholding and morphological transformations (e.g. erosion and dilation) are examples of fundamental CV techniques that allow for efficient processing in scenarios where real-time performance is required. For example, if the general color of a desired target object is known (and is consistent), color thresholding is useful for "blob detection", which helps to identify and segment distinct objects or regions based on their color [4]. By defining upper and lower bounds for a specific color range in an image, the developer can find regions of interest (i.e. "blobs"), whereby any pixel falling within the defined range is marked as part of the object of interest, while others are discarded. In a scenario where an agent needs to detect red balls on a green field, properly-tuned upper and lower color vector bounds could identify groups

of sufficiently "red" pixels as a detected object and filter out everything else.

Often with blob detection, a common issue arises whereby non-contiguous regions of pixels are detected as separate objects, even though they are a part of the same ground-truth object. Once blob detection is completed, a customary next step is to apply morphological operations to refine the results, specifically erosion and dilation [5]. Erosion removes the outer pixel layers of detected blobs, shrinking their size. This can be useful for eliminating small noise or separating objects that might have been detected as a single blob due to close proximity. Conversely, dilation adds layers of pixels to the outside of detected blobs, making them larger. This is helpful when objects are fragmented or partially obscured, as it helps to merge these disconnected regions into a single object.

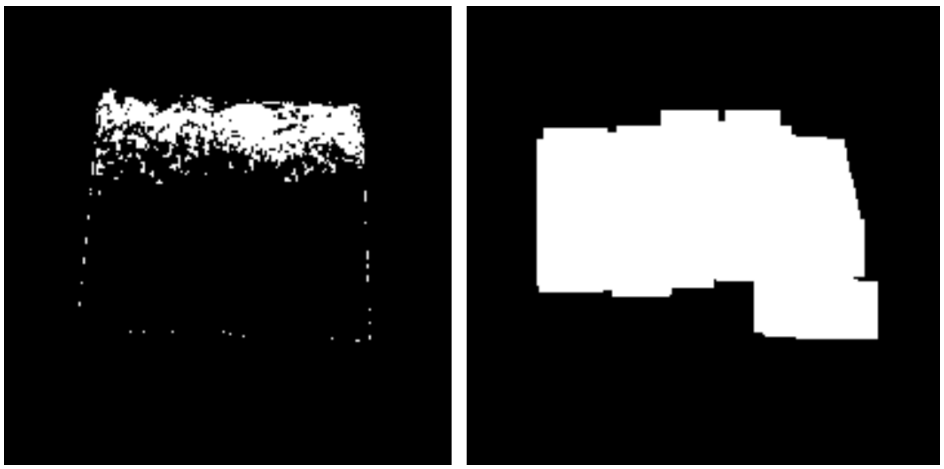

Figure 2: Simple color thresholding on an object without any morphological transformations (left) and with erosion & dilation applied (right)

Figure 2 shows the effects of applying erosion and dilation during post processing of an image. The left image shows a mask of a color-thresholded object without any morphological transformations. Many small, non-contiguous groups of pixels are detected by the model, which are registered as separate objects. Consequently, the image on the right shows the effects of eroding each blob by 1 pixel layer, followed by a dilation of 6 pixel layers. With these changes, the object is registered as one contiguous blob by the vision model despite the results of the initial color thresholding. This ensures one object is not detected as multiple smaller objects, that only the significant objects are retained (i.e. less false positives from small blobs of similar color are likely to arise). This is particularly useful when dealing with cluttered or low-contrast images where some objects might be erroneously detected due to sensor noise or poor lighting conditions.

While generally quick and resource efficient, the primary drawback of basic vision pipelines is their lack of robustness. While we can build some resilience into the pipeline to environmental changes (e.g.

changes in lighting, object occlusion, etc.), excessive changes can cause the system to fail or require more extensive tuning beyond what is feasible. For more dynamic use-cases, more complex CV pipelines are needed.

1.4.2 Machine Learning for Object Detection

To detect objects that are too complex for rudimentary color thresholding, developers typically employ Machine Learning (ML) models. For vision-based tasks, Convolutional Neural Networks (CNNs) [6] can model complex objects with higher accuracy, and they can develop resilience to environmental changes (if trained correctly). These models are typically trained on large datasets and can generalize to different lighting conditions, camera angles, object obfuscation, or noise in the image. The computational resources needed for training are more intensive, often requiring GPUs at runtime to achieve real-time processing. However, the accuracy and adaptability of ML-based vision systems make them preferable for applications where precision is important.

1.4.2.1 YOLO Architecture

One of the most popular deep learning models for real-time object detection is the YOLO [7] ("You Only Look Once") series of models. YOLO is an architecture designed to predict the bounding boxes and class labels of objects in an image in a single forward pass, making it efficient and thereby ideal for real-time applications.

YOLO operates by first downsampling the input image through convolution and pooling layers, which help detect patterns that are indicative of notable objects in the image, and ultimately obtain the class name(s) of any detected objects. YOLO then upsamples the transformed input, which recovers the spacial information used to predict a bounding box location around the detected object. The final output is an array of predictions, each containing a bounding box and a class label. The architecture is designed to run in real-time, making it ideal for dynamic environments. More recent versions of YOLO (e.g. YOLOv9 [8]) have iterated on previous YOLO versions to improve both accuracy and efficiency.

1.4.2.2 Generating Labeled Datasets via Data Augmentation

A successful ML model depends heavily on the quality and quantity of data used during training. For object detection tasks, particularly when dealing with specific environments like Unity Hall, acquiring a diverse and representative dataset is a significant challenge. In the case of elevator buttons, labeled datasets

exist [9], but do not contain sufficient data for a model to generalize to the Unity Hall elevator buttons.

To address this, one can create their own dataset tailored to their needs. Creating a big enough dataset (10,000 to 100,000 images with labeled bounding boxes) is highly unfeasible to do manually. However, techniques exist to synthetically generate large quantities of data on which to train a model. Homography [10] describes a linear transformation that relates the perspective of some image to another perspective (position & orientation). By applying a linear transformation to the pixels of a source image, it is possible to generate another version of the original image where the camera's (x, y, z) position and (γ, θ, ϕ) orientation appear to change, programmatically producing a new image that simulates a different viewpoint of the same object.

1.5 Human-Robot Interaction

Human-Robot Interaction (HRI) is a common discipline in robotics that explores how robotic systems can be used by or work with humans. It has become a significant research topic in recent years as its application has been used for a variety of different purposes. For example, HRI has been used in industrial settings, such as in assemblies and production lines, where robots and humans have collaborated to improve efficiency and human safety. Similarly, some robots safely help humans carry out agricultural tasks, such as harvesting, seeding, fertilizing, and more. In medical settings, HRI plays a large role in assistive robotics and rehabilitative robotics. The discipline continues expanding into additional sectors including space exploration, mining operations, and educational environments, each presenting unique interaction challenges and requirements. [11]

1.5.1 Multimodal Interfaces

In recent years, the integration of speech recognition technologies has transformed human-robot interaction capabilities. These advanced technologies make it possible for computers to understand and process human voices. [12] However, despite these significant advances, there is still plenty of opportunity for speech interaction between humans and robots to improve, particularly in non-ideal circumstances. Background noise, varying accents, and specialized vocabulary continue to impact the accuracy and reliability of recognition. To address these limitations, some experts have highlighted the importance of multimodal interfaces that combine speech with visual elements, such as touchscreen displays. This hybrid approach accommodates challenging settings and diverse user needs and preferences. It also provides flexibility in the circumstance that one modality becomes compromised, allowing for a more resilient and adaptable system

overall. [13]

1.5.2 Retrieval Augmented Generation and Fine-Tuning

Retrieval Augmented Generation (RAG) is a technique that improves the capabilities of a pre-trained Large Language Model (LLM) by integrating its knowledge bases with external data. As shown in Figure 3, when RAG receives a prompt, it retrieves relevant documents that contain information related to the prompt. These retrieved documents provide additional context to the LLM before it generates a response. An alternative way of providing a pre-trained LLM with specific data is fine-tuning. This process requires training datasets and potentially multiple rounds of training to tune the LLM's performance to a satisfactory level. Fine-tuning involves changing the parameters of the LLM model. Full Fine-Tuning updates all model weights for comprehensive adaptation, while Parameter-Efficient Fine-Tuning (PEFT) methods like LoRA, adapter tuning, and prefix tuning strategically modify select parameters to achieve similar results with significantly reduced computational overhead. The choice between these approaches depends on available resources and desired performance outcomes.

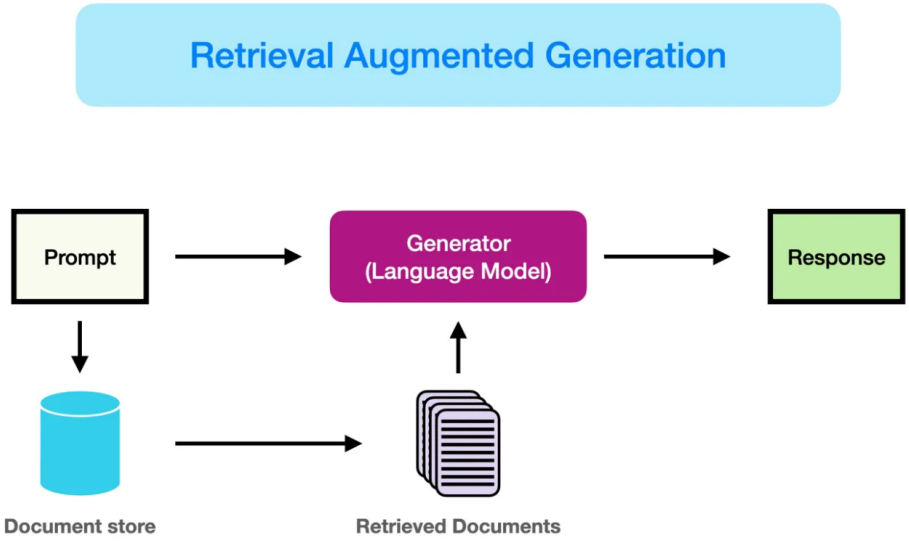

Figure 3: RAG Workflow Diagram

RAG and fine tuning have their tradeoffs depending on the user's purposes and requirements. One benefit of fine-tuning is faster inference than RAG since the training data would be in the model weights. However, fine-tuning is significantly more computationally expensive than RAG. Another downside of fine-tuning is that if the LLM goes through multiple rounds of training, it can be difficult to restore a better version of the model if a newer round worsens the generated responses. Also, the LLM would need to be

fine-tuned again if the training data becomes outdated or new data needs to be added. Alternatively, RAG allows documents to easily be updated to correct inaccurate information or add new data without modifying the model itself. Some advanced implementations balance these trade-offs by taking a hybrid approach of combining fine-tuning and RAG for optimal performance.

2 Methodology

2.1 Project Goals

The goal of this project was to design and develop a cost-friendly autonomous tour guide robot capable of navigating different floors of Unity Hall and interacting with visitors. With our project, we met the following objectives:

1. Autonomous multi-floor navigation
2. Button-pushing and elevator interaction
3. Touchscreen GUI
4. Verbal navigation commands
5. Question-answering system

2.2 Navigation Stack

WPI's newest academic building, Unity Hall, has 5 floors. Our goal was to design a robot capable of guiding users to a given destination on any floor. One unique challenge faced in this building was the abundance of transparent walls. Many of the walls throughout the building contained fritted glass towards the bottom and completely transparent glass towards the top. Laser scans travel through transparent structures, therefore the LiDAR was placed in a manner such that the majority of the readings would intersect the fritted glass. The ROS 2 Navigation Stack is a collection of open-source packages that enables robots to localize and navigate in an environment. Packages used in Tori's navigation stack included `slam_toolbox`, `nav2_amcl`, and `bt_navigator`.

2.2.1 Mapping

Tori is equipped with a Slam Tech RP LiDAR S1. Tori's maps were generated using SLAM Toolbox [14]. The `slam_toolbox` package uses data from multiple sources to mapping the robot's environment: The `LaserScan` message from the LiDAR sensor is used to differentiate between free space and walls. The TF transform `odom` → `base_link` calculates robot odometry, `base_link` → `laser` accounts for physical LiDAR offset on the robot, and `map` → `odom` provides global position of the robot in the map frame. SLAM

toolbox's asynchronous launch file was used to perform mapping. This configuration file allows for dynamic mapping while the robot is moving. The resolution of the maps generated were 0.05 meters per cell. To save the maps, the `map_saver` tool was used in the `nav2_map_server` package. `map_saver` saves two files:

- PGM: Contains the occupancy grid
- YAML: Contains map metadata (resolution, origin, etc.)

To account for the physical offset between the robot's `base_link` and the LiDAR sensor frame `laser`, the following static transform was published:

```
ros2 run tf2_ros static_transform_publisher 0.14 0 1.12 0 0 0 base_link laser
```

- `0.14 0 1.12`: Translation in meters along the x, y, and z axes.
- `0 0 0`: Rotation in radians (roll, pitch, yaw).
- `base_link`: Parent frame (robot base).
- `laser`: Child frame (LiDAR sensor).

These values can be updated as needed to account for offsets between sensors. Due to the height of the LiDAR, Figure 4 shows dynamic obstacles that were captured and recorded as occupied space in the generated maps that would not otherwise be present in the same environment.

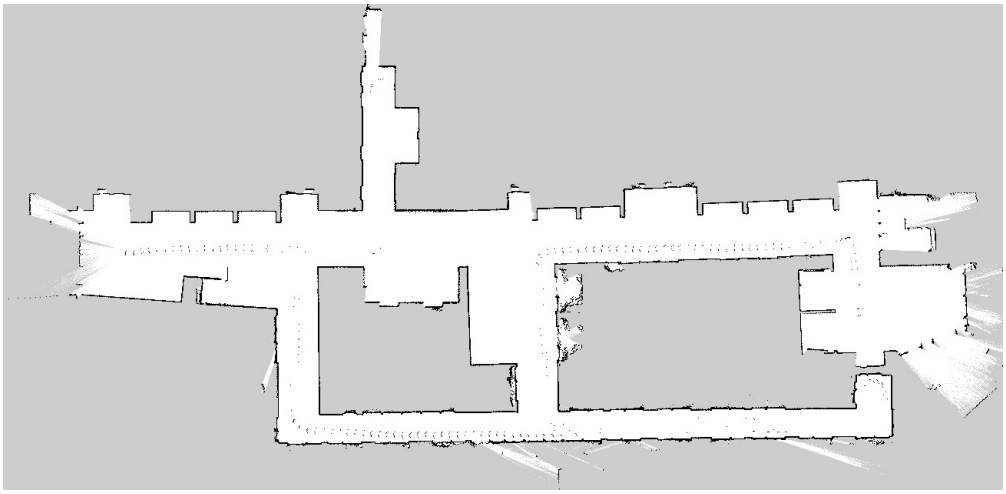

Figure 4: Map generated of floor 4 using SLAM Toolbox before cleanup

These readings were removed from the pgm files using an editor to produce clean maps shown in Figure 5 where the only occupied spaces were the walls of the building.

Figure 5: Map generated of floor 4 using SLAM Toolbox after cleanup

As seen in the maps above, the regions of the map where the free space extends far beyond the walls indicated regions where the laser scans traveled through transparent glass. To minimize the distance the scans traveled, the maximum laser distance was set to five meters during mapping. Our team members manually moved Tori across each floor to generate maps. For better accuracy, Tori was pushed slowly and with minimal external jitter. This setup enabled Tori to generate accurate maps of each floor, despite the challenges posed by the transparent walls.

2.2.2 AMCL

The `nav2_amcl` package [15] was used to perform AMCL within Unity Hall. This package localizes the robot and publishes the `map` → `odom` transformation. The parameters for AMCL were configured in a local copy of the `nav2_params.yaml` file found within the `nav2_bringup` package. The parameter `use_sim_time` was set to `false` since Tori was operating in real-time. Additionally, the `robot_model_type` parameter was set to `"nav2_amcl::DifferentialMotionModel"`. An initial pose estimate was provided to assist Tori in localizing within the environment. In order to exclusively perform localization, all that is needed is to launch the `localization_launch.py` file. The path to the map yaml file was passed as a launch argument to the launch file in order to bring up the map in which Tori localizes. To visualize Tori localizing in the map, the launch file was launched after RVIZ. An initial pose can be set through the command line by publishing a message to the `inital_pose` topic or manually through the RVIZ interface. To apply changes to the local `nav2_params.yaml` file in Tori's directory, a custom script named `deploy.sh` was used to manually overwrite the existing parameters in the installed `nav2_params.yaml` file within the ROS 2 installation directory.

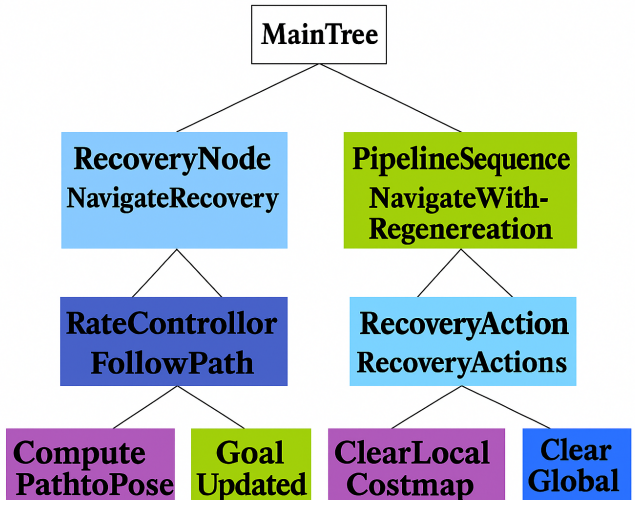

Figure 6: Visualization of navigate to pose with planning and recovery XML

2.2.3 Navigation

2.2.3.1 Behavior Tree (BT) Navigator

The `BT_Navigator` node [15] was used to navigate Tori within Unity Hall. This node receives a goal pose and navigates Tori to the desired destination. The parameters for navigation were configured in the local copy of the `nav2_params.yaml` file found within the `nav2_bringup` package. The behavior of the navigator is defined within an XML file that is pointed by `default_nav_to_pose_bt_xml`. Tori's behavior tree continuously plans the global path at 1 Hz and consists of recovery actions in the case no progress is made. Figure 6 depicts the flow of states for Tori's behavior tree.

It is important to note that the backup and spin behaviors were removed from the default Navigate to Pose with Planning and Recovery Behavior Tree as the desired fallback behavior for Tori was to clear the local and global costmaps to replan a path. The backup and spin behaviors were undesirable for our purpose as they occurred regardless of whether an obstacle obstructed Tori's path.

2.2.3.2 Controller Server

The controller server hosts various plugins, such as trajectory generation, progress checker, and goal checker. The default controller specified in the FollowPath plugin is `DWBLocalPlanner`. The Dynamic Window Based Planner generates multiple trajectories which are then evaluated by critics such as `PathAlign`, `GoalDist`, and `ObstacleFootprint`. The trajectory with the highest score sends its corresponding command velocities. The default progress checker plugin is `SimpleProgressChecker`, which ensures the robot moves

a minimum distance within a required time period. The default goal checker plugin is `SimpleGoalChecker`, which ensures the robot is aligned within a certain x, y, and θ tolerance within the goal. Due to Tori's size, it was important that she stayed closer to the center of the hallways to avoid catching on any corners while turning.

The optimal parameters were found by tweaking the parameters specified in the plugins and testing the robot's change in behavior while navigating. The final value set for `xy_goal_tolerance` was 0.45 meters. The final value set for `yaw_goal_tolerance` was .75 radians. The FollowPath plugin was also switched from `DWBLocalPlanner` to `RegulatedPurePursuitController` since this controller consistently generated feasible paths in comparison to the default. The pure pursuit algorithm finds a point in front of the robot and calculates the linear and angular velocities required to drive towards it. Once it reaches this point, a new point is selected, and the process continues until the robot has reached the goal pose. The distance at which a new point is selected is specified by the `lookahead_dist` which was set to 0.7 meters. The regulated pure pursuit algorithm implementation in Nav 2 comes with built-in collision detection. The distance at which the robot looks ahead to collision detect is also specified by `lookahead_dist`. The `cost_scaling_dist` also decreases the robot's speed based on its proximity to obstacles. Our primary goal was safety, therefore this value was set to 3.0 to drastically slow the robot when navigating close to obstacles. The linear velocity for the robot was set to 0.9 meters per second.

2.2.3.3 Global and Local Costmaps

The Costmap 2D package uses a 2D grid-based costmap to represent a robot's environment. It is used by the planner and controller servers to generate a trajectory that avoids obstacles and stays clear of high-risk areas. Although Tori was not a holonomic robot, her shape was approximated as a circle for the navigation stack. In the global costmap, `robot_radius` was set to 0.45 meters. A `footprint_padding` of 0.1 meters adds padding to Tori's base to ensure she completely maneuvers around obstacles. Three plugins were utilized for the global costmap: `static_layer`, `obstacle_layer`, and `inflation_layer`. The `static_layer` uses the map provided by the `map_server` to place on the costmap and indicates the location of static obstacles, such as walls. The `obstacle_layer` uses 2D raycasting to mark obstacles within the LiDAR's specified range. The `inflation_layer` uses an exponentially decaying function around obstacles to deter from navigating in their close proximity. The `inflation_radius` was set to 0.65 meters, which added a padding of 0.65 meters around obstacles. The planner is unable to generate a plan that crosses within this padding space. The global costmap was updated and published at a frequency of 1 Hz. The cells in the occupancy grid contain values from 0 to 254, which correspond to a cost to travel through each cell.

A cost of 0 indicates a free cell, whereas a lethally occupied cell is indicated with a cost of 254. The local costmap was set to 3 meters by 3 meters. Its update frequency was at 5 Hz, with a publish frequency of 2 Hz.

2.2.3.4 Planner Server

The planner server hosts the server that responds to path planning requests. It takes a goal as input and uses the planner defined in the plugin to output a feasible path to the goal. The default plugin `NavfnPlanner` was used to compute paths to the desired destination. `allow_unknown` was set to true, which allowed for planning in unknown but not necessarily occupied spaces. Figure 7 shows how the servers work in tandem to navigate the robot to the goal pose.

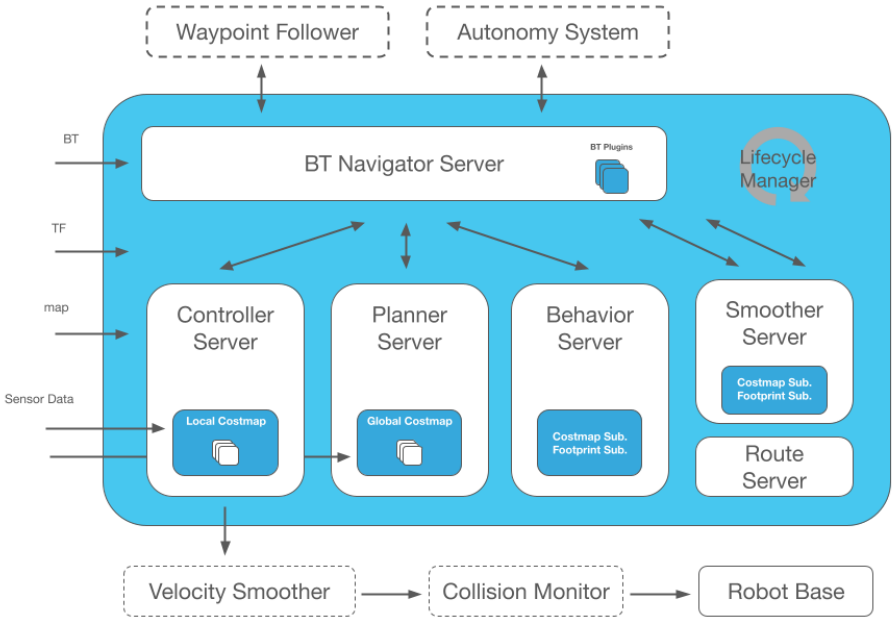

Figure 7: Nav 2 Architecture

2.2.4 Multi-floor navigation

To implement multi-floor navigation, two custom nodes were created to dynamically switch between maps and keep track of the floor Tori was currently on. The `Dynamic Map Loader` node uses the service client `/map_server/load_map` to dynamically update the map when a message of type `std_msgs/String` is passed to the `elevator_button` topic. If the button for floor 4 is pressed, for instance, a string formatted as `Floor4` should be published to this topic. The `StateTracker` node maintains the current floor of the robot. It subscribes to the `current_floor` topic. Ideally, a message indicating the current floor is published to

this topic after pressing an elevator button. That publisher is not included within this node. Once the user makes a request for a specific floor, the node will publish the requested floor to the `requested_floor` topic. The intermediary topic that stores the requested goal pose is called `filtered_goal_pose`. The current and requested floors are compared. If they match, the robot publishes the user's goal pose to `goal_pose` and leads the user to their destination. If the floors do not match, the robot publishes the goal pose for the elevator on the current floor in preparation to enter the elevator. Once Tori has exited the elevator on the correct floor, she navigates to the goal pose that was originally passed to `filtered_goal_pose`. The coordinates for locations corresponding to specific classrooms and areas for each floor are specified in `Unity_coords.json` in the `Unity-Coordinates` package.

2.3 Vision System

In this section, we describe the various Computer Vision (CV) pipelines in place to help detect various objects needed for Tori to accurately push elevator buttons, and autonomously navigate into the Unity Hall elevator.

2.3.1 Dataset Generation

We created custom YOLO datasets to identify two main sets of objects: elevator buttons, and the elevator door (more specifically "open door" and "closed door" objects). In order to create big enough labeled datasets, we created a workflow to hand-label bounding boxes in some few dozen "source" images with a simple GUI Python application.

2.3.1.1 Deriving the Transformation

After labeling and formatting source images, we constructed a set of data generation scripts programmatically produced the rest of the dataset based on the manually-labeled source images. From a single source image, we were able to alter the perceived camera pose in small increments across any translation or rotation axis (or combination thereof). To do each transformation, we defined four standard homogeneous transformation matrices, each representing a basic geometric operation in 3D space [16].

Rotation about the x-axis by angle θ

$$T_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation about the y-axis by angle ϕ

$$T_y(\phi) = \begin{bmatrix} \cos\phi & 0 & -\sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ \sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation about the z-axis by angle γ

$$T_z(\gamma) = \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 & 0 \\ \sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translation in 3D space by $(\Delta x, \Delta y, \Delta z)$

$$T_{trans}(\Delta x, \Delta y, \Delta z) = \begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Composed 3D Extrinsic Transformation

The composed 3D transformation matrix T_{total}, representing the system's "extrinsic parameters" [17] is obtained by multiplying the individual transformation matrices in the desired order. For example, if we first rotate about the x, y, and z axes, and translate by $(\Delta x, \Delta y, \Delta z)$, we have:

$$T_{ext} = T_{trans}(\Delta x, \Delta y, \Delta z) \cdot T_z(\gamma) \cdot T_y(\phi) \cdot T_x(\theta)$$

If given a vector $p_0 = (x_0, y_0, z_0)$ in 3D space, $T \cdot (x_0, y_0, z_0, 1)^{\mathrm{T}}$ gives the resulting point $(x_1, y_1, z_1, 1)$ as if p_0 was translated or rotated as specified by the parameters to T. Each of these transformation matrices represents an operation in 3D, but since image pixel vectors only contain (x_0, y_0) pixel information, we add two more transformation matrices to project the image pixel space from 2D to 3D and vice versa.

Projection from 2D to 3D

Given image width W and height H, we want the camera (0, 0) to point to the center pixel of the image, corresponding to $(-W/2, -H/2)$. We additionally want to add a dimension to the 2D image vector to make a 3D, so we define a 4x3 matrix:

$$T_{2\to3} = \begin{bmatrix} 1 & 0 & -W/2 \\ 0 & 1 & -H/2 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

Projection from 3D to 2D

To project back to 2D image coordinates, we construct a 3x4 "intrinsic parameters" matrix [17]. Since $T_{2\to3}$ performs a translation by $(-W/2, -H/2)$ to define the camera center, we now perform the opposite translation by $(W/2, -H/2)$. When doing world-to-image calculations with a real camera, this matrix is also dependent on physical properties of the camera (namely focal length). However, since we are only simulating a camera, we can custom define our focal length. Based on existing suggestions from users on StackOverflow [18], we define our focal length to be some quantity that scales with image size. With

$F = \sqrt{W^2 + H^2}$,

$$T_{3\to 2} = \begin{bmatrix} F & 0 & W/2 & 0 \\ 0 & F & H/2 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Overall Composed Transformation Matrix

Finally calculating the overall transformation matrix including projections from 2D into 3D and vice versa:

$$T_{overall} = T_{2\to 3} \cdot T_{ext} \cdot T_{3\to 2}$$

This framework enabled us to translate or rotate an image along or about any axis to simulate the camera taking a picture from a different pose. We mainly needed to simulate different yaw angles (y-axis rotation) and distances away from the objects (z-axis translation), so we applied this transformation in small increments (1-pixel increments in the z-axis; 1-degree increments about yaw axis) with experimentally determined boundaries (-150 pixels to 300 pixels for the z-axis; -45° to 45° about yaw axis).

By applying a transformation for every possible combination of camera pose values given these constraints, we were able to generate tens of thousands of transformed versions of the source image and find the bounding box location by transforming the corners of the source bounding box under the same linear transformation. Using this method on multiple labeled source images, we systematically created large datasets, providing the model with bulk amounts of examples to train on. This technique also ensured that our models could generalize better by being exposed to a wider range of simulated conditions - namely different orientations and lower resolution (i.e. simulating being further away from the object). To reduce dataset generation times by several orders of magnitude, we multi-threaded the synthesis scripts and used WPI-provided compute clusters to split the task across as much hardware as possible.

2.3.1.2 Examples

Figure 8 shows a few select examples of our transformation pipeline. The leftmost image shows the manually labeled source image, where the top-left and bottom-right corners of each bounding box are shown in green. The center image shows the source image transformed with $\phi = 30°$, and the right image shows the

source image transformed with $\phi = -30°$, $\gamma = 30°$ and $\Delta z = 300$ pixels. In each image, the bounding box of each labeled object was also transformed, creating a separate training example for the model. Figures 9 and 10 show similar examples used to train a model to identify closed and open elevator doors, respectively.

Figure 8: Left: manually-labeled source image of elevator buttons with button bounding boxes. Center: rotated 30° about the y-axis. Right: rotated -30° about the y-axis, 30° about the z-axis, and translated 300 pixels backwards in the z-axis.

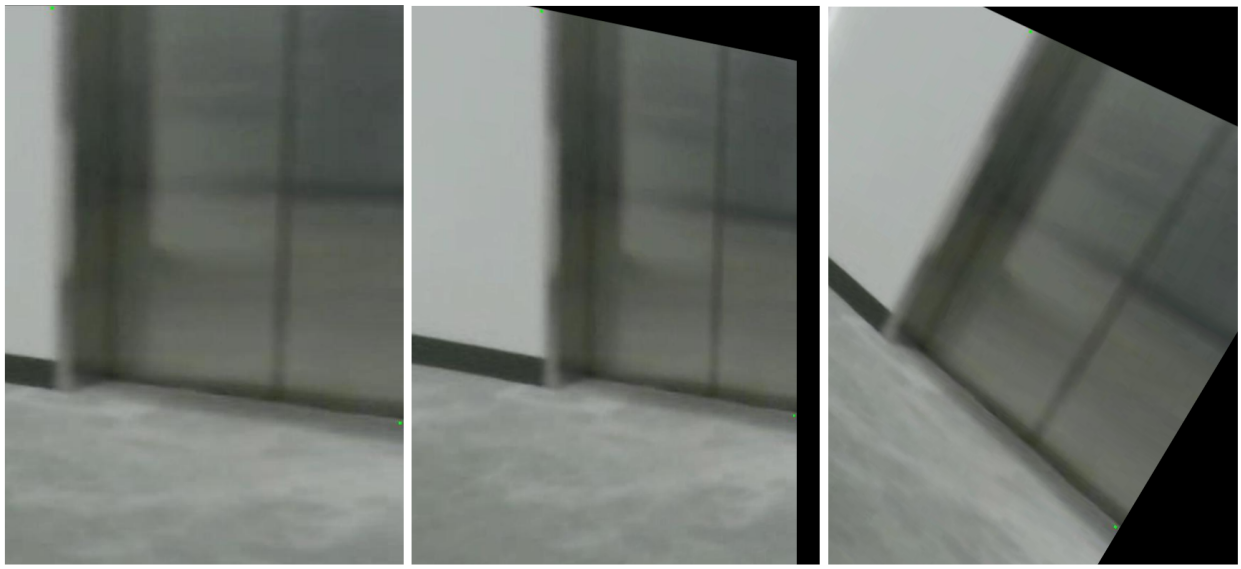

Figure 9: Left: manually-labeled source image of a closed elevator door with button bounding boxes. Center: rotated 30° about the y-axis. Right: rotated -30° about the y-axis, 30° about the z-axis, and translated 300 pixels backwards in the z-axis.

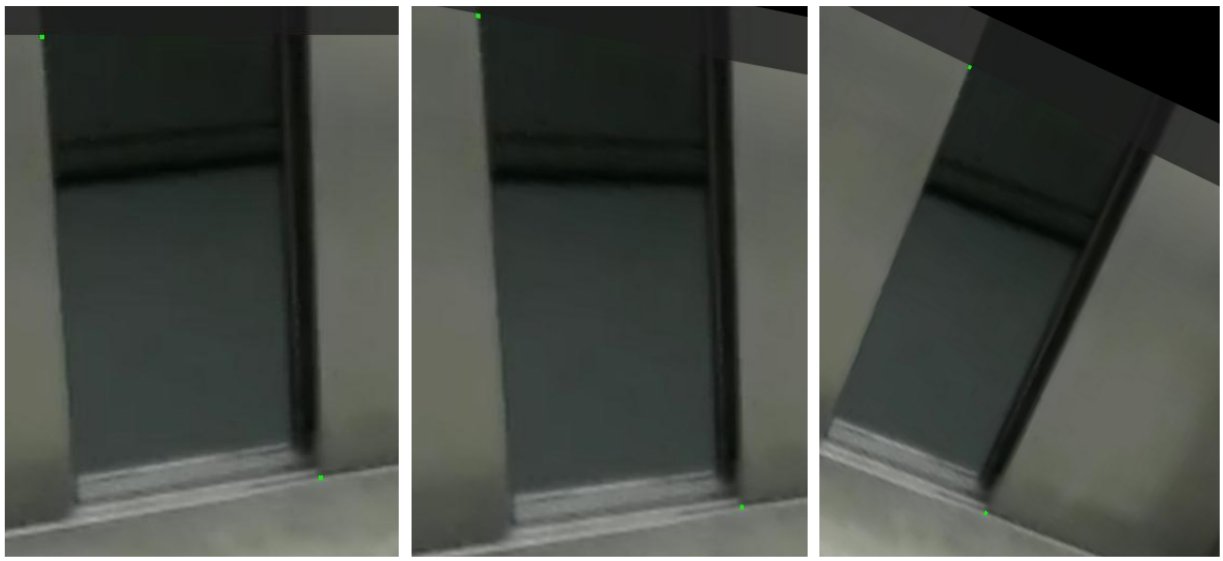

Figure 10: Left: manually-labeled source image of an open elevator door with button bounding boxes. Center: rotated 30° about the y-axis. Right: rotated -30° about the y-axis, 30° about the z-axis, and translated 300 pixels backwards in the z-axis.

2.3.2 Raw OpenCV Processing

For certain computer vision tasks, we opted to use simple OpenCV pipelines instead of a full YOLO model. This approach is advantageous because these pipelines are significantly less computationally intensive and can run entirely on the Jetson's CPU, preserving GPU resources for more demanding tasks.

The first OpenCV pipeline on Tori detects the manipulator arm. This detection is essential for performing pixel-space PID control [19] to align the pusher with the button, based on its position in the camera frame relative to the YOLO-detected button. OpenCV was well-suited for this task not only due to its efficiency, but also because the pusher underwent frequent design and appearance changes. These changes affected its appearance and would have required time- and resource-intensive retraining of a YOLO model. Instead, we wrapped the pusher in purple felt and used blob detection with color thresholding, dilation, erosion, and contour detection (see Section 1.4.1) to reliably locate it, regardless of mechanical changes.

The second OpenCV application verifies whether an elevator button was successfully pressed. In Unity Hall, the elevator indicates a successful press by lighting an orange LED at the center of the button. Detecting this bright orange illumination is trivial and best handled with a simple RGB or HSV threshold, rather than a machine learning approach. During the elevator sequence, once the pusher's limit switch registers contact, the robot backs up and checks if the targeted button is lit. It uses the YOLO model to locate buttons in the image, applies an HSV mask to the region, and calculates the ratio of orange pixels to

the total button area. If this ratio exceeds a configured threshold, the press is considered successful.

2.4 Human-Robot Interaction

Our goal was to make human interaction with the robot as intuitive and seamless as possible. To accomplish this goal, we incorporated two primary user interaction methods:

1. **Graphical User Interface**: A touch screen monitor where users can indicate their desired destination on the display.
2. **Verbal System**: A microphone that allows a user to provide Tori verbal navigation commands or ask questions about Unity Hall and WPI.

2.4.1 Graphical User Interface (GUI)

An application was created in React to host the Graphical User Interface(GUI). The GUI was designed with the goal of creating a simple yet effective user experience. The GUI is displayed on a 14-inch touch screen monitor with a 16:10 aspect ratio. The `react-router-dom` library was used to navigate to different pages within the app. The following is a list of the different paths used by the router:

- `/` — Renders the `Start` component
- `/Path/:room` — Renders the `Path` component with a dynamic room parameter
- `/explore-unity` — Renders the `ExploreUnity` component
- `/floor1` to `/floor5` — Render respective `Floor1` through `Floor5` components for floor navigation
- `/Arrived` — Renders the `Arrived` component indicating arrival at the goal destination

By default, the user begins on the `Start` page shown in Figure 11. This page features a photo of Unity Hall in the background. The colors used on this page and throughout the rest of the application reflect WPI's official logo colors.

Choosing the option to Explore Unity Hall routes to the `ExploreUnity` component where the user may select their desired floor from the floor listing. Upon selecting a floor, the user is routed to the respective `/floor` where they may select a destination from the options provided 12. From either of these pages, the user may navigate back to the home page by selecting the button with the home icon or navigate back to the

Figure 11: GUI home page

previous page by selecting the back icon. Since the option to navigate to the restrooms, stairs, and elevator are common to each floor, they are displayed in the top right corner with image icons. When designing the GUI, it was important that the text was large enough for the average user to read.

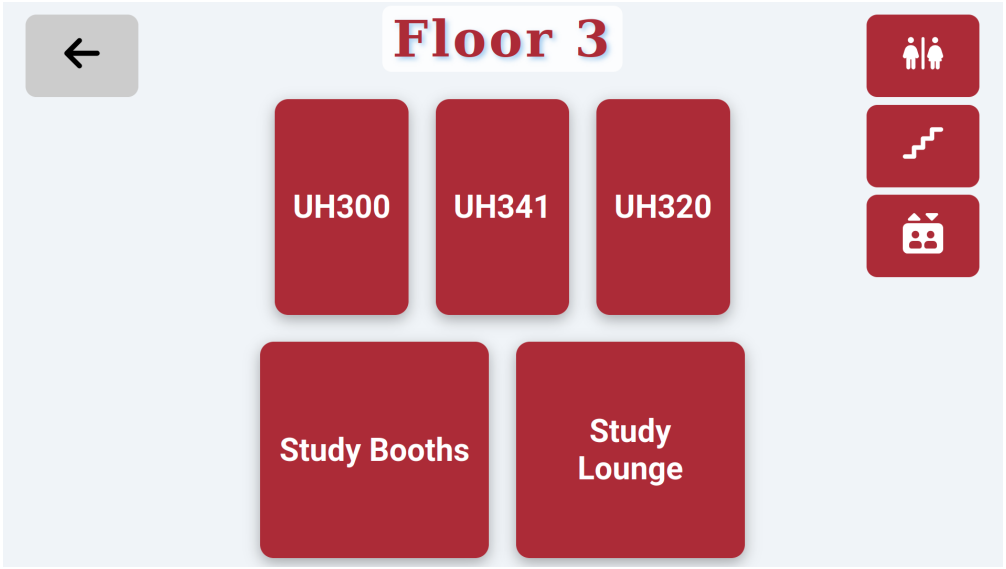

Figure 12: Options presented for selected floor

Once a destination is selected, the `Path` component is rendered which displays a face depicting Tori as shown in Figure 13. On the backend, each button corresponds to a goal destination listed in `Unity_coords.json`. The `rosbridge_server` allows the React application to communicate with ROS using the rosbridge protocol. The server creates a WebSocket connection and allows the `rosbridge_library` to

convert the JSON string corresponding to the goal coordinate into a ROS message of type `PoseStamped`.

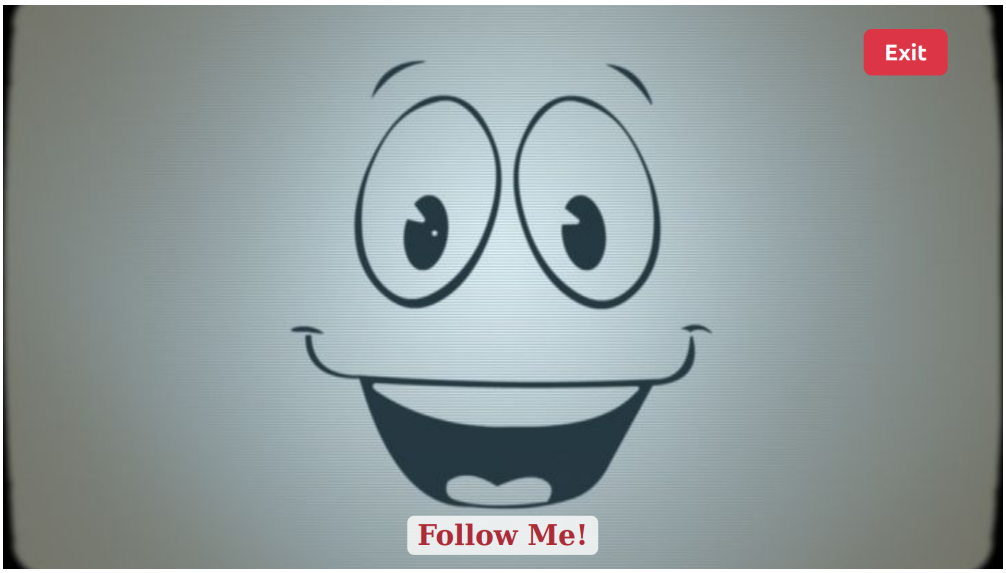

Figure 13: GUI indicates that navigation has begun

Once navigation has concluded, the GUI is routed to the `Arrived` component that provides visual feedback to the user of their arrival. The `react-confetti` library was used to celebrate Tori's hard work as shown in Figure 14.

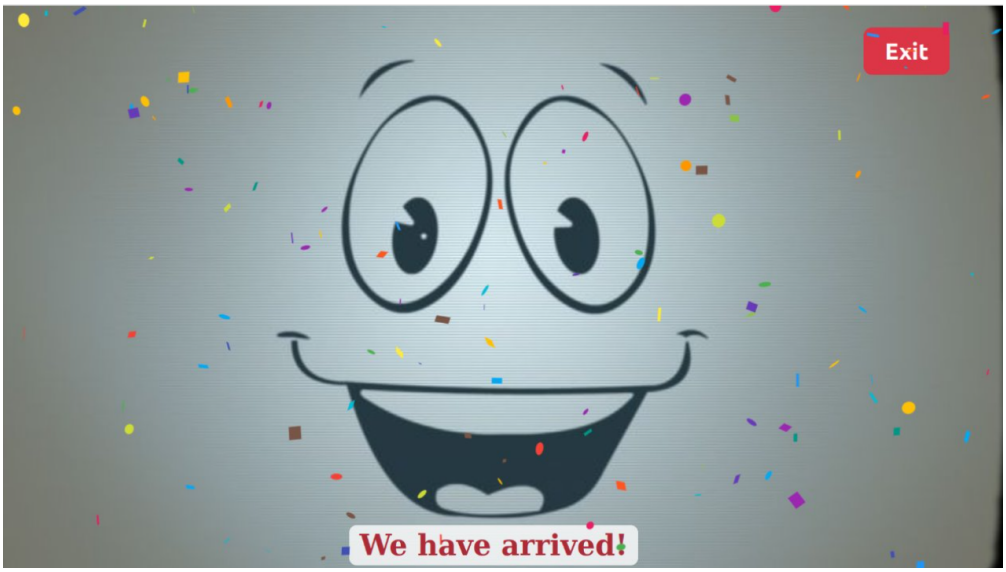

Figure 14: GUI indicates that user has arrived to their destination

2.4.2 Speech Technology

To add more human-robot interaction options, we created a verbal command system in Python that gives Tori the capability to engage in conversation with users. They are able to verbally provide Tori navigation commands, as an alternative to the touchscreen buttons, or ask questions about Unity Hall using a microphone attached to the robot.

The first step to creating this system was finding a speech recognition library that could convert the user's words to text.We ultimately settled on an offline speech recognition API called Vosk [20] that can run on the Jetson Nano. Luckily, Vosk has the ability to turn the audio stream on and off, which is helpful when Tori is speaking so that the speech-to-text library does not accidentally pick up Tori's responses to the user as the user's responses.

For Tori to speak to users, we found an advanced text-to-speech library called TTS [21] that had many pre-created voice options that we could pick from. Since many speech-to-text libraries are computationally heavy and we already had space concerns on the Jetson Nano, we opted to use a library called pyt2s [22] that makes API calls to TTS to avoid downloading the model directly onto the Jetson Nano.

2.4.3 Verbal Navigation Commands

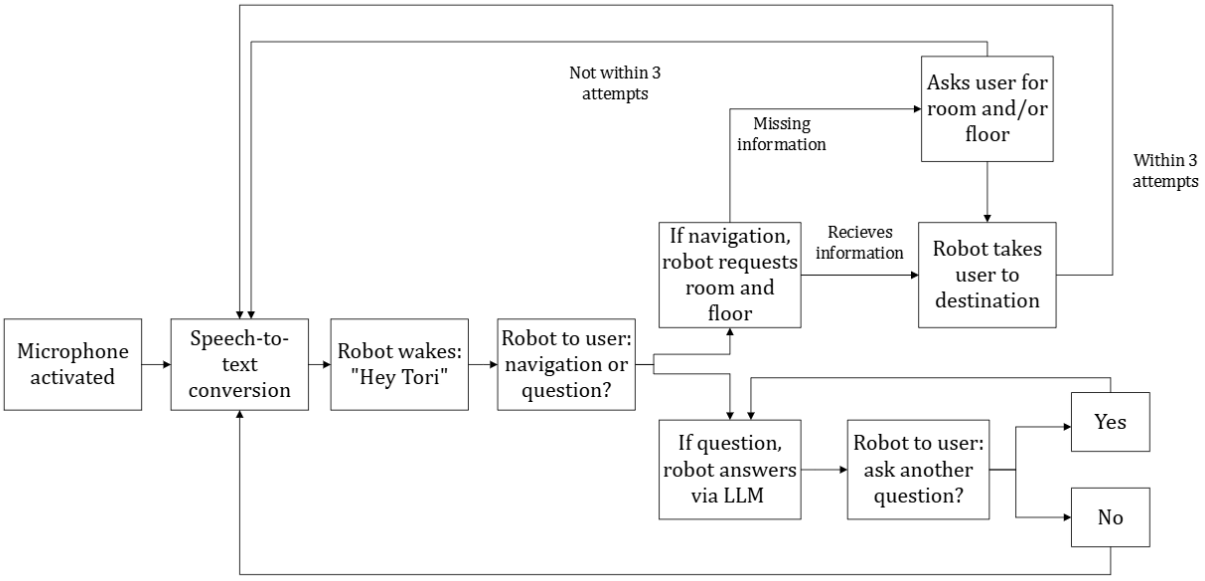

Figure 15: Verbal Command System Workflow

When the React frontend launches and the GUI boots up, it triggers the voice system through a

Flask API call. The Flask backend serves as a bridge between the React frontend and the Python-based speech processing system. When activated, Flask executes the speech recognition pipeline and Tori listens for "Hey Tori." Once the wake phrase is detected, she responds to the user with an introduction and asks the user if they want to say a navigation command or ask a question. If the user indicates that they want to say a navigation command, Tori asks users for both a room and floor number to effectively guide them to their desired location. Figure 15 details the step-by-step workflow.

For Tori to understand verbal navigation commands, we created a system that is based on the repetition in Unity Hall between floors. The classrooms are numbers with the first digit being the floor number (e.g. UH 400 is on the fourth floor), so the system is able to extract the floor from the room number. This means users don't need to specify the floor separately when giving a room number. Alternatively, if the user requests to be taken to a non-number location and provides a floor (e.g. stairs on the third floor), Tori can navigate there because she has received both a room (stairs) and floor (third floor). If either the floor or room is provided by the user, it is saved to a variable so when Tori asks for missing information, the user just needs to respond with that information to ensure all variables have valid data. For generic requests without the floor number (e.g. elevator), Tori will ask the user for the floor. If the floor is provided without the room (e.g. second floor), Tori will ask the user for the room. The system is designed to allow the user up to three attempts to provide a valid location before resetting back to the wake word state. We also implemented error handling so if the user asks to go to a location but provides the wrong floor, Tori will correct them with the right floor. For example, if the user requests to go to the career development center on the first floor, Tori will inform them that it is on the fifth floor. Since different users might use different words to reference the same location (e.g. elevator and lift), we created a mapping system that connects synonyms to the same location.

To ensure locations are valid, the system checks users' location requests against a JSON file that contains room coordinates. If the room coordinates do not exist for the location the user asks to go to, Tori will inform the user that the location does not exist and request a new room and floor. If the location is valid, the system will send the room's coordinates to the ROS2 navigation stack through a message to the goal pose topic. Then the navigation controller creates a path from the current location to the requested location and follows it.

2.4.4 Question-Answering System

If the user indicates that they would like to ask a question instead of a navigation command, Tori will use an LLM along with Retrieval Augmented Generation (RAG) to effectively respond to inquiries. We

tested various LLMs with the intention of choosing a model that could locally run on the Jetson Nano to allow for offline question-answering capabilities, which would particularly be helpful in the elevator where there is unreliable internet connection. We considered using TinyLlama [23], a 1.1B parameters lightweight chat model LLM that uses the same architecture and tokenizer as Meta's Llama 2. However, due to the small size of the model, it did not perform as well as we had hoped. We later switched to a larger model, Meta Llama 3.2 (3B parameters) [24], when we found a strategy to run it locally using an Ollama API instead of Hugging Face directly. This LLM produced much stronger responses and was able to run relatively fast. Other models we had tested included Gemma 3 [25] and Llama 3.1 [26].

Our original plan was to fine-tune an LLM using a custom dataset. One of the challenges with this is that it takes a lot of computational power to train an LLM and our laptops were not capable of handling the load on their own. We were planning on using WPI's Turing servers due to the available GPUs that would be able to handle the training quickly and efficiently. However, with more research and guidance from our advisors, we found a simpler approach to accomplish our goal: RAG. To make Tori a knowledgeable tour guide, we provided her the entire WPI website as the external knowledge base for responding to questions. This way, Tori was able to search through a knowledge base of WPI information rather than relying solely on the LLM's pre-trained knowledge.

To begin the RAG system, we launch an Ollama docker container to run Meta Llama 3.2 and initialize a ChromaDB [27] vector database to store document embeddings. As documents are indexed through, they are broken into chunks and their embeddings are stored for retrieval. When Tori is asked a question, the system converts the query into embeddings using the nomic-embed-text-v1.5 model [28] in Ollama. Early in the process, we had also tested the all-MiniLM-L6-v2 sentence transformers model [29], which is faster and more lightweight than nomic-embed-text-v1.5. A key difference between the models is that nomic-embed-text-v1.5 model can process up to 8192 tokens, while all-MiniLM-L6-v2 can process about 512 tokens. Since we decided to use RAG on a large context (the entire WPI website), nomic-embed-text-v1.5 model was a better choice for our purposes.

After embedding, it then searches the ChromaDB vector database for document chunks with similar embeddings, retrieving the most relevant pieces of information. In previous testing attempts, we had used Sentence Transformers [30] for embeddings and FAISS [31] for vector similarity search. While they worked well, ChromaDB was an all-in-one solution that removed the need to use separate libraries for embeddings and similarity search.

The relevant chunks are passed to the LLM, along with the query embeddings, to generate an accurate response about Unity Hall or WPI. Using prompt engineering, the LLM is instructed to adopt the

persona of Tori, a helpful tour guide robot, and provide concise answers limited to one or two sentences. When asked a question outside of the context, she is instructed to state that she does not know the answer instead of producing inaccurate information.

The question-answering system was originally designed to generate a response then feed it into the text-to-speech model for Tori to verbalize. However, we found that there was a 15-20 second time gap between the user asking the question and Tori's response. This was caused by the time it took to index through documents and for the LLM to generate a response. To reduce some of that time, we implemented threading and queues to allow for the speech generation to occur simultaneously with the LLM streaming a response real-time. Every time the LLM generated a sentence or reached a phrase limit of 150 characters, it was added to the speech queue and fed to the text-to-speech library to allow for the speech generation to occur without blocking the main process. There was a short delay each time more words were sent to the text-to-speech library so sending it one sentence at a time allowed for appropriate and natural-sounding pauses.

2.5 Control System

Beneath the systems responsible for human interaction, vision, and mapping lies the control infrastructure: the hardware and low-level software that manages sensing, computation, and actuation.

Our robot is built on a repurposed First Robotics Competition (FRC) chassis donated by Team 716. All original control hardware was removed, giving us full freedom to design a system tailored to our requirements. While this increased the project's scope, it enabled us to build an entirely open-source control stack from the ground up.

2.5.1 Goals and Guidelines

We defined the following criteria to guide our hardware and software decisions:

1. **Safe**: Tori is a large robot operating near people—it must never endanger them.
2. **Standardized**: Use proven, off-the-shelf components and libraries wherever practical.
3. **Self-Contained**: Avoid dependencies on wireless connectivity, which is unreliable in some environments (e.g., elevators).
4. **Extensible**: Make it easy to upgrade, swap, or modify components for future teams.

2.5.2 Overview

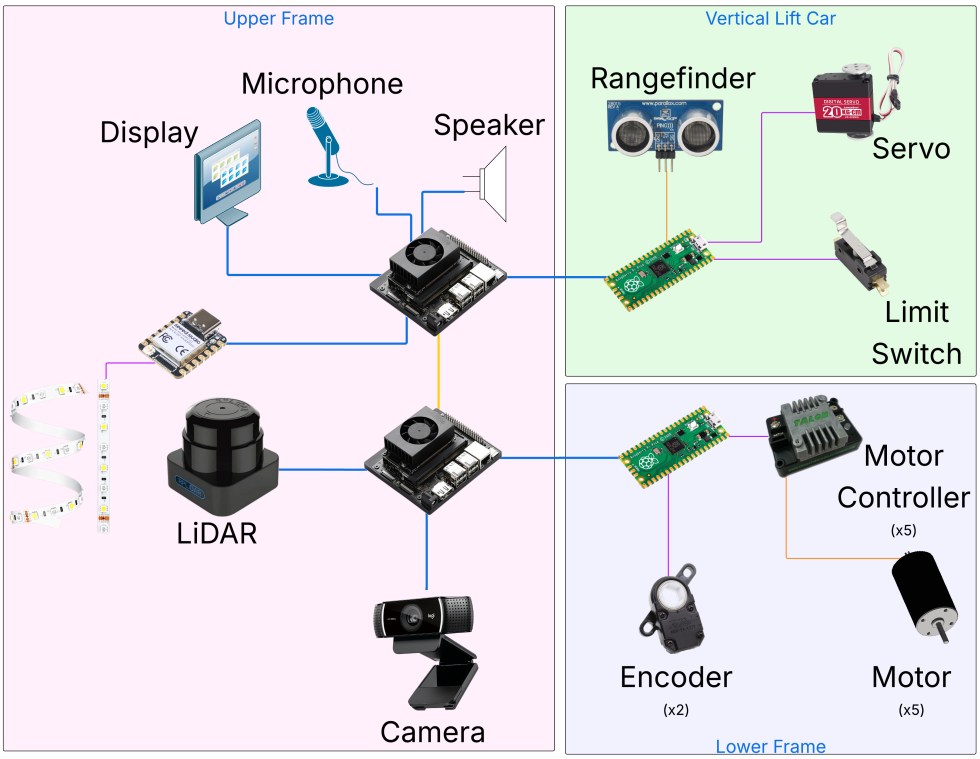

Figure 16: Overview of Robot Onboard Signaling

To meet the above goals, Tori's onboard control architecture incorporates two Nvidia Jetson Orin Nanos, multiple microcontrollers, and a variety of sensors and actuators. An overview of these components and their connections is shown in Figure 16. Each Jetson handles a different portion of the workload and communicates with a dedicated Raspberry Pi Pico microcontroller for low-level control.

2.5.3 Compute

Each Jetson Orin Nano runs Jetpack 6 (Ubuntu 22.04) and is configured with key-only SSH access, ROS 2 Humble, Micro-ROS agents, and various support packages. We selected the Jetson primarily for its CUDA GPU acceleration, which is critical for real-time workloads such as computer vision and large language models (LLMs).

Despite being powerful for embedded systems, the Jetsons are constrained by their 8GB of shared CPU/GPU RAM. Our combined stack—including Nav2, GUI, speech, vision, and LLM—exceeded this limit in practice. While GPU (CUDA) memory can sometimes be swapped to disk [32], it is very expensive and slow, therefore we split the workload across two Jetsons to achieve acceptable performance. Jetson L

(left) runs Nav2, elevator interface nodes, LiDAR, lighting control, and vision. Jetson R (right) handles the GUI, speech I/O, LLM-based question answering, and the display system. While this architecture increases software and networking complexity, it allows the robot to operate multiple demanding subsystems simultaneously.

2.5.4 Communications

The Jetsons connect to WPI's Wi-Fi for development and maintenance, but this connection is not reliable enough for runtime operation due to roaming latency and multicast traffic demands of ROS 2. For reliable communication, the Jetsons are directly connected over a CAT 5e Ethernet cable using static IPs (10.0.0.1 and 10.0.0.2). This supports FastDDS [33] peer discovery with minimal configuration, as indicated by the thick yellow line in Figure16.

Each Jetson also connects to a dedicated Raspberry Pi Pico over USB serial. These Picos run Micro-ROS [34] on top of the Pi Pico C SDK. Micro-ROS provides a ROS 2 Client Library and FastDDS transport implementation in C, enabling the Picos to function as ROS 2 nodes via the micro-ROS serial agent running on the Jetsons.

Jetson L also connects to an ESP32-C3 microcontroller over USB serial. Since its sole task is controlling LED lighting, it uses a simpler protocol: an Arduino sketch on the ESP32 communicates with a PySerial-based ROS node running on Jetson L.

2.5.5 Microcontrollers

Tori includes three microcontrollers: two Raspberry Pi Picos and one ESP32-C3 Xiao. Each serves a distinct purpose.

Primary Pi Pico (Jetson L) This Pico handles all drivetrain motion control, battery monitoring, and basic system safety checks. Each side of the differential drivetrain uses a quadrature encoder mounted on the gearbox. Rather than using CPU-intensive polling or interrupts, we take advantage of the RP2040's Programmable I/O (PIO) peripheral [35] to decode the quadrature signals in hardware. The main CPU then reads these values to calculate wheel velocities and robot odometry. The motion control loop runs a PID algorithm to match actual vs. commanded velocity and adjusts the motor controller's PWM outputs accordingly. This entire control loop runs on core 1, isolating it from less critical tasks. Core 0 handles Micro-ROS communication, battery voltage monitoring, and watchdog logic. If core 1 stalls or crashes, core

0 can restart it. If core 0 itself fails, the RP2040 watchdog will reset the entire board.[36] Battery voltage is read via an analog input. If the voltage drops too low, or if communication from the Jetson times out, the Pico automatically disables all actuators to ensure safe behavior.

Secondary Pi Pico (Jetson R) This Pico is mounted directly on the lift mechanism and handles local sensing and actuation. Its main responsibilities are reading the lift height from a US-016 analog ultrasonic sensor, monitoring a contact switch at the end of the pusher, and driving the pusher servo. Core 1 processes ultrasonic sensor data, applying a running average and converting the raw voltage to height in meters. Core 0 publishes this data to ROS and processes pusher servo commands by generating the appropriate PWM signal. Placing a dedicated microcontroller on the lift reduced the wire length needed for analog signals, improving reliability and avoiding voltage drop issues that would have arisen from running long sensor wires back to the main control board.

ESP32 Light Controller The ESP32-C3 Xiao controls two addressable LED strips using the FastLED Arduino library.[37] It receives a byte stream over USB serial, where each message begins with an ASCII character ('L', 'R', or 'B') indicating the left, right, or both strips, followed by RGB triplets for each LED. This simple firmware, written in the Arduino framework, meets the robot's lighting needs without requiring the overhead of the ESP-IDF SDK or a full ROS integration.

2.6 Electrical Architecture

Tori's electrical system is a modification of the original FRC configuration, retaining the 12V lead-acid battery, power distribution panel, and drive motors. In addition to this core system, Tori incorporates two independent 5V power buses and a 19V bus to support various onboard electronics. All systems draw from the same 12V battery: the 19V and primary 5V buses remain continuously powered, while the 12V and secondary 5V lines are routed through a remote emergency stop (e-stop) relay for safety.

This arrangement enables the use of a COTS remote power cutoff as a reliable e-stop while ensuring that critical control electronics remain powered, thus avoiding issues from sudden shutdowns or data loss. Figure 17 provides an overview of this configuration. Not pictured in the diagram are several local capacitors used on each Raspberry Pi Pico board to stabilize the 5V rail, a voltage monitoring circuit connected to the primary Pico, and auxiliary power supplies for connected USB peripherals.

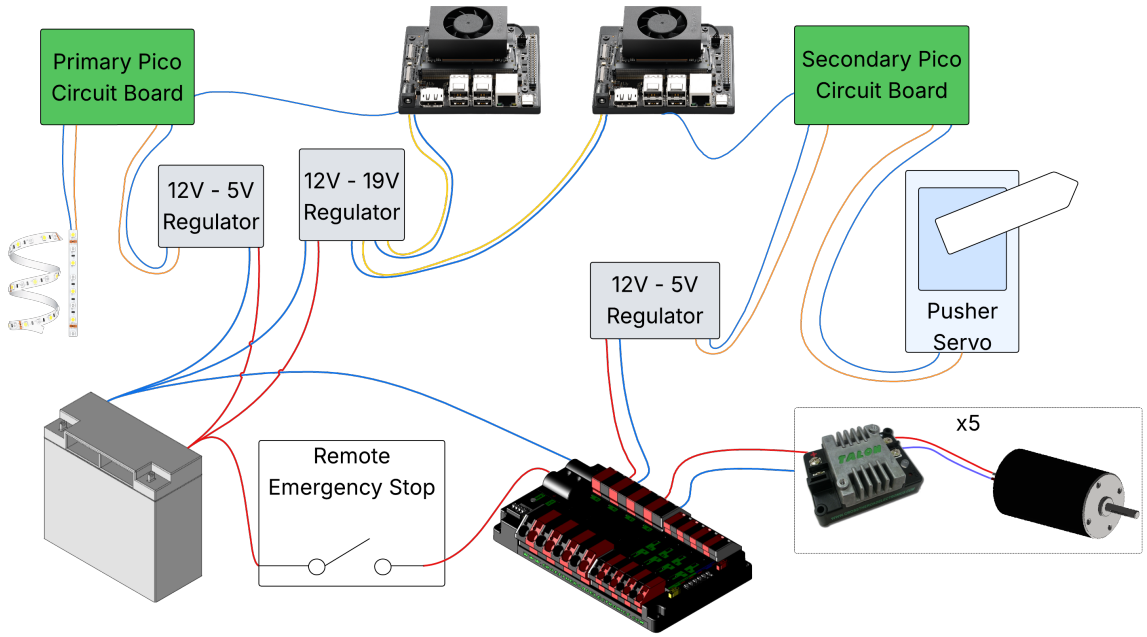

Figure 17: High-level Electrical Diagram

2.7 Mechanical Specifications

As previously noted, Tori originated as the 2019 competition robot for FRC Team 716 [38]. The original chassis was extensively modified for this project. Major removals included: all pneumatics, the FRC control electronics, two drivetrain motors, the primary manipulator arm, and the rear drive chains.

In its current configuration, Tori operates as a front-wheel-drive differential steering robot, with unpowered omni-wheels at the rear to facilitate smooth in-place turning. Each side of the drivetrain is driven by two CIM brushed DC motors connected to the front wheels via a custom gearbox and chain drive. Power electronics and the primary Raspberry Pi Pico are mounted on an angled plexiglass plate above each gearbox.

A rectangular aluminum superstructure rises from the forward half of the chassis, supporting the touchscreen interface, Jetson computers, and a LiDAR sensor. At the front end of this superstructure is a three-stage, cable-driven lift mechanism, which supports the custom-built button-pushing assembly. Figure 18 shows a rear view of the chassis with the decorative shell removed, exposing key mechanical elements. For reference, the robot's "rear" is the user-facing side, opposite its typical direction of travel.

This chassis was chosen primarily for its robust three-stage lift, which provided an ideal foundation for the button-pushing mechanism. While a competition robot may not seem optimal for a tour guide role,

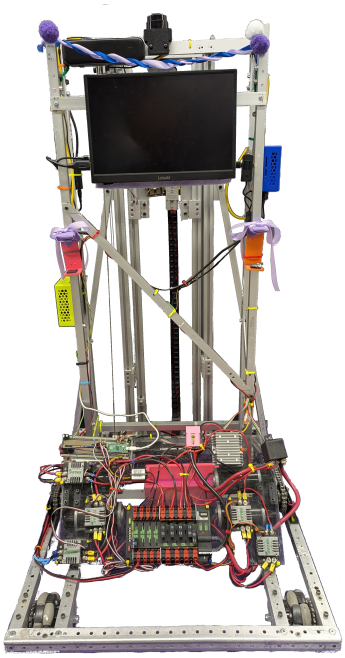

Figure 18: Rear view of Bare Robot Chassis

reusing it allowed our team to focus efforts on autonomy and user interaction. Its modular construction, ample space for computing hardware, and our prior experience with its components made it a practical and efficient choice.

Since Tori is directly interacting with humans, a plexiglass cover was added over the electronics to make it presentable and prevent it from being damaged. Some users mentioned that Tori seemed to be intimating in her original form. To make Tori's appearance more approachable, decorations in a soft lavender color were added to cover up the metal chassis. The decorations that were added include fabric, felted flowers, pom poms, and ribbons. Safety pins were used extensively to ensure that none of the fabric attached to the robot was dragged on the floor or caught in the wheels. The pieces of fabric could be lifted to expose the battery holder in order to safely switch out batteries. In addition, to showcase Tori's name, colorful varsity letters were affixed on top of the fabric on the plexiglass. In general, Tori's final design shown in Figure 19 was eye-catching and welcoming, suited for a tour guide robot that would interact with daily users.

Figure 19: Tori's appearance after adding external decorations

3 Results

3.1 Navigation System

Tori was able to consistently navigate to a given goal destination using the GUI and verbal navigation commands while path planning around people in the hallway. If there was a group of people bloccking her path, she successfully waited until her path cleared to begin moving again. However, she tended to drive in slanted paths and sometimes over-corrected her turns when avoiding obstacles. Additionally, if Tori was stuck inside the lethal space near a wall, she was not able to recover since the wall is stationary. This situation typically lead to a failure in obtaining the goal pose. The `localization_launch` and `navigation_launch` files were included in one common launch file to bring up the navigation stack.

When conducting a user study, we gave participants a brief overview of our project and asked them to interact with our robot with no prior instructions. Following their experience, we asked them to rank how safe they felt using Tori on a scale of 1-10 with 1 being a low satisfaction score and 10 being a high satisfaction score. With a sample size of nine users, the average rated safety experienced was relatively high at 8.3/10. Overall, users noted they felt safe while Tori was navigating. Suggestions included improving responsiveness at intersections and clearly reporting any struggles to the user.

3.2 Vision & Elevator Interaction

After assembling large custom custom datasets, we trained two YOLO-based object detection models: one to identify open and closed elevator doors (around 175,000 training images), and another to identify elevator button locations (around 75,000 training images). Both models were trained for five epochs. Throughout training, we monitored both training and validation loss curves. As shown in Figure 20, the box loss for both models continued to decrease by the end of the fifth epoch, indicating that the models had not yet fully converged. This suggests that additional training epochs could have further improved performance. However, due to time and computational resource constraints, and given the already strong inference-time performance we observed, we opted not to extend training further unless needed.

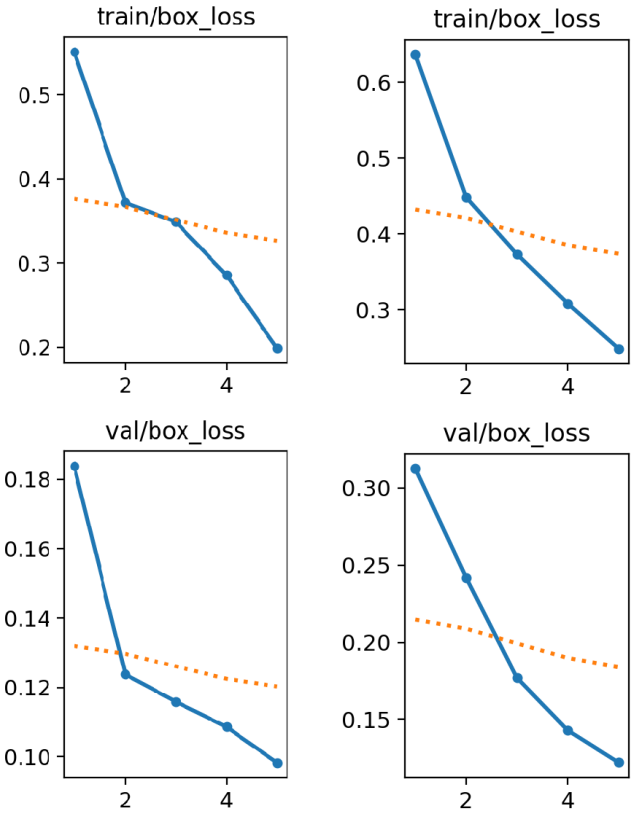

Figure 20: Left: training (top) and validation (bottom) box loss for the elevator door detection model. Right: training (top) and validation (bottom) box loss for the button detection model.

3.2.1 Qualitative Inferencing: Button Detection model & Color-based Blob Detection

The elevator button detection model demonstrated high accuracy on the validation set, reliably identifying buttons across a range of scenarios. Qualitative results (see Figures 21 and 22) show that the

model accurately detected buttons at close, mid-range, and far distances. Even in challenging cases-such as when buttons were partially occluded, viewed from sharp angles, or under varied lighting conditions, the model maintained robust detection. Further, figure 23 shows sequential annotated frames of the model running in real-time, also displaying the color-based blob detection of the button pusher and its real-time alignment with the elevator button. Since there were sufficient negative training examples containing no buttons, the final iteration of the model detected no false positives during its deployment.

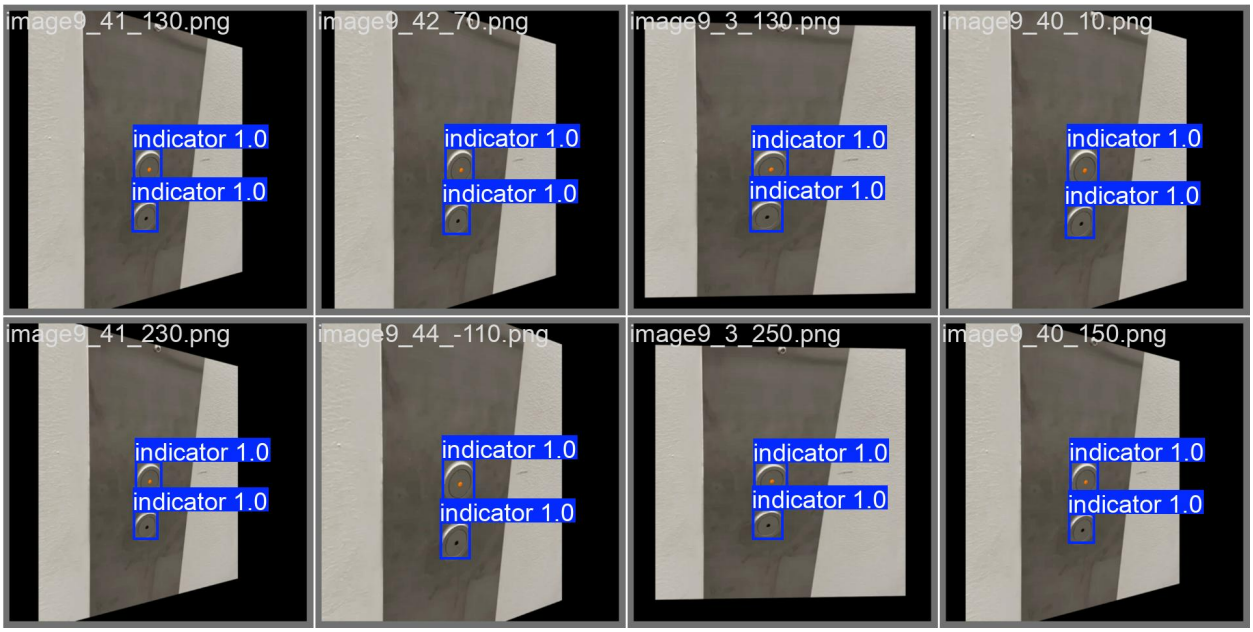

Figure 21: Model inferencing on elevator buttons generated by the YOLO training & validation process. Confidence level ranges from 0.0 to 1.0, with 1.0 being maximum confidence.

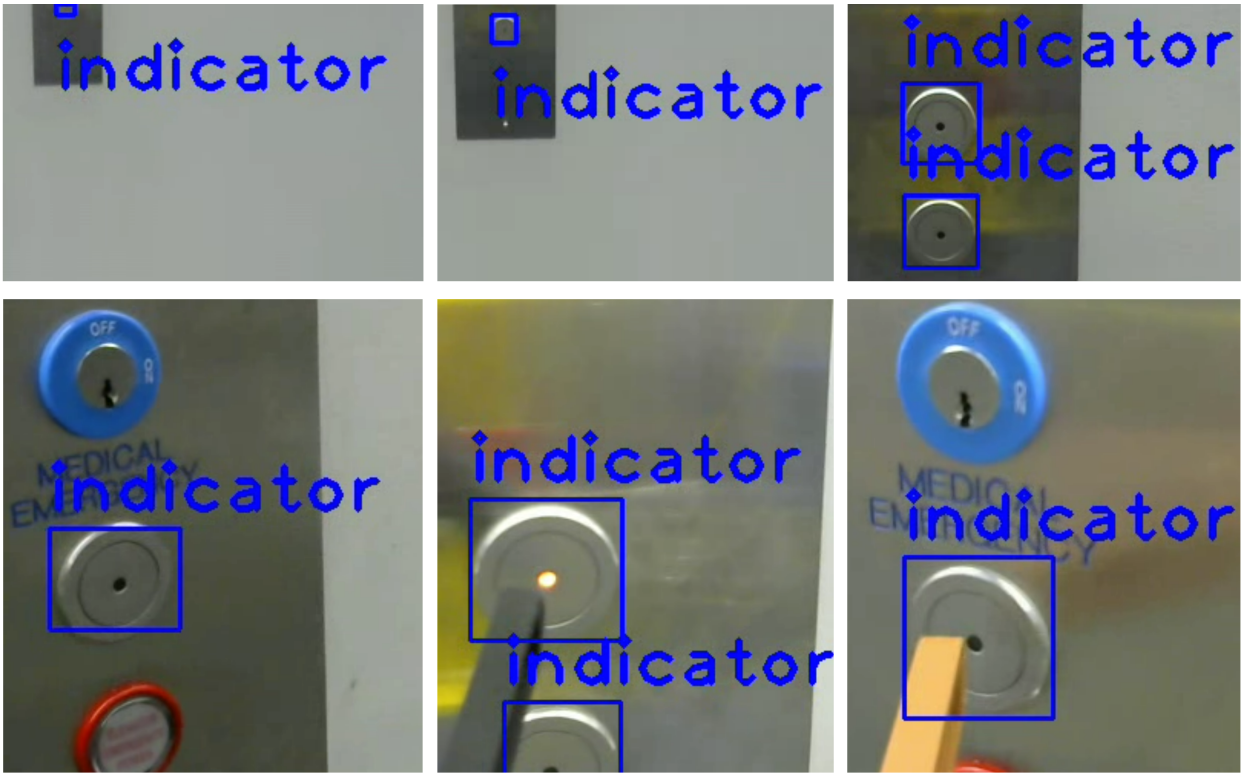

Figure 22: Selections of elevator buttons annotated by the button detection model at various angles and with occlusions

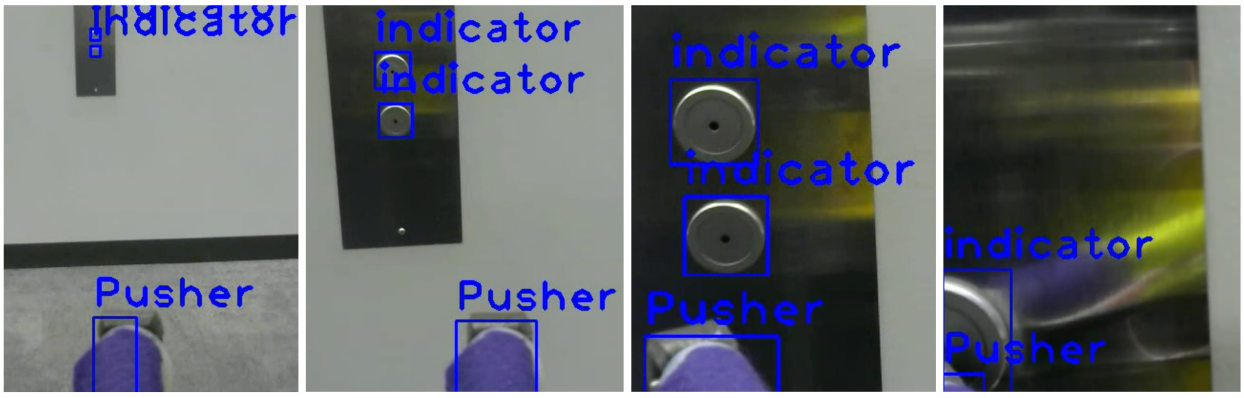

Figure 23: Sequence of annotated frames of Tori autonomously aligning its pusher with the elevator buttons. Tori successfully pushed the button in last frame, despite it being out of the camera's field-of-view.

3.2.2 Qualitative Inferencing: Elevator Door Detection Model

Similarly to the button detection model, the elevator door detection model effectively distinguished between open and closed doors. Qualitative results (see Figures 24 and 25) again show detected doors in varying states of occlusion, lighting, and angles. In all these scenarios, the model's predictions remained

consistent and reliable. Additionally, the model was able to track a dynamic "open elevator" object as the elevator doors opened and closed, allowing the robot to know exactly when the elevator starts opening and helping it gauge how close it is or how far inside of the elevator it is (supplementary to LiDAR data). Figure 26 shows this model running in real-time as the robot guides itself into the elevator using PID entirely in the pixel space. Since there were sufficient negative training examples containing no elevator doors, the final iteration of this model also detected no false positives during its deployment.

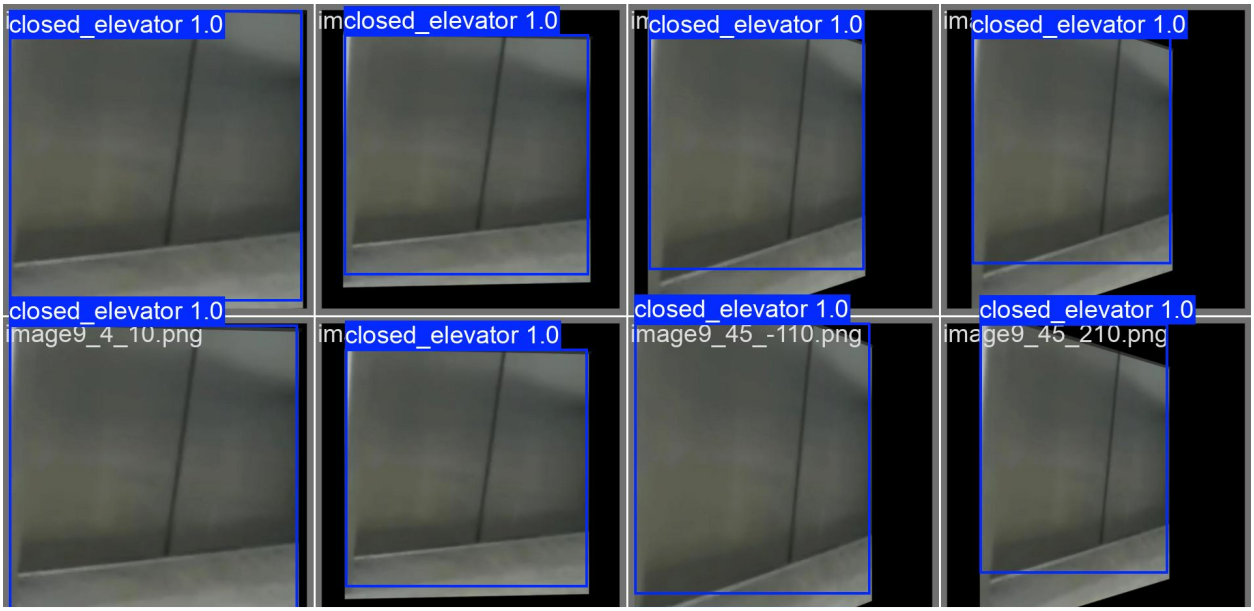

Figure 24: Model inferencing on closed elevator doors generated by the YOLO training & validation process. Confidence level ranges from 0.0 to 1.0, with 1.0 being maximum confidence.

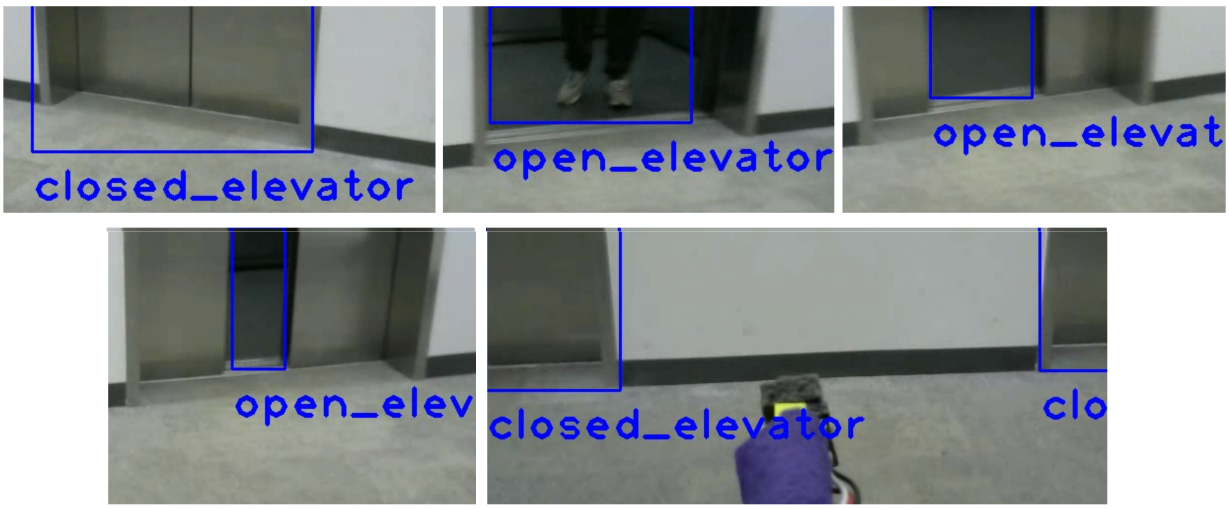

Figure 25: Selections of elevator doors annotated by the door detection model at various angles and with occlusions

Figure 26: Sequence of annotated frames of Tori detecting elevator doors and autonomously entering the open elevator. The last frame shows Tori fully inside the elevator.

3.3 Human-Robot Interaction

Our implementation of the verbal system allowed Tori to successfully be able to respond to wake phrase activation ("Hey Tori"), process verbal navigation commands, extract location and floor information from natural language requests, communicate with users when they need more information, navigate to requested destinations using a ROS2 node, and respond effectively to user queries.

We took a number of steps to improve the user experience once the technical groundwork was set. To improve Tori's communication with the user, we gave her extra phrases to say before she begins navigating and after she arrives at her destination. This was done by creating a goal proximity node that subscribes to the goal proximity topic and waits for it to return "1", which indicates that Tori has arrived at her destination. We also added LED lights that visualize the different modes of the robot. The lights are green with a moving pattern when Tori is navigating towards a goal. They turn white when Tori has arrived at her goal or is waiting for a navigation command. The LEDs turn red when the goal has failed and are an orange marquis pattern when in elevator mode. We also created a loop that brings users back to wake phrase state after Tori fulfills a navigation request or answers questions. This removed the need to use a button on the GUI to restart the system, as we had initially planned.

When conducting a user study, we gave participants a basic overview of our project and asked them to engage with our robot with no instructions. Afterwards, we asked them to rank how easy Tori was to use on a scale of 1-10 with 1 being a low satisfaction score and 10 being a high satisfaction score. The response was generally positive with an average score of 7.4 and a sample size of 8 participants. We observed that users were more likely to press the GUI buttons than to say the wake phrase, so understanding how

to improve the users' engagement with the vocal functionality could be further investigated in the future. Based on testing and observation, the verbal navigation system and the question-answering system worked well individually, but robot integration and human usability could be improved.

We also asked participants to rank the LLM's responses on a scale of 1-10 with 1 being a low satisfaction score and 10 being a high satisfaction score. The average was 6.8 with a sample size of 4 participants. Given more time, we would have been able to increase our sample size, but we were still able to get helpful feedback from the users. One point of feedback was that the workflow was designed in a way that did not allow for very natural conversation with users. For that reason, Tori would sometimes detect questions incorrectly and therefore answer the wrong questions. If the user experience was improved based on this feedback, the average score for LLM responses could be improved in future user studies.

4 Future Work

4.1 Navigation System

Further testing and pruning may allow for considerable improvements to the navigation stack. Due to time constraints, the floors in `Unity_coords.json` include the major rooms on each of the floors, however they exclude Professor's offices. In the future, it may be beneficial to add all the offices and tech suites located on each floor to allow more navigation options. Only one LiDAR was mounted on Tori at a height of 1.12 meters off the ground. This means Tori only detected objects that are at or above this height, which was not ideal. Adding an additional LiDAR source to the `scan` topic may help improve Tori's ability to detect objects outside her current scanning range. In addition, depth sensors or ultrasonic sensors may also be added as observation sources in the navigation stack to enhance safe navigation capabilities.

Tori also drove in slanted paths and sometimes over-corrected her turns when avoiding an obstacle. A different path planner, such as `Smac Planner` may allow for smoother navigation. We also did not have time to tune the `use_cost_regulated_linear_velocity_scaling`, `interpolate_curvature_after_goal`, and `use_fixed_curvature_lookahead`, parameters in the controller server. However, including these parameters may help Tori slow down and approach turns with ease. Furthermore, editing the fallback behaviors in the behavior tree xml file may help provide more helpful recovery behaviors. For instance, if Tori was stuck in lethal space near the wall, she would not be able to navigate further since the wall is stationary. With a backup or rotate in place fallback behavior enabled, she may be able to remove herself from such a situation. However, the default fallback behaviors in Nav2 do not automatically provide collision detection. Therefore,

it may be beneficial to implement a custom node that performs a fallback behavior with dynamic obstacles taken into consideration.

4.2 Vision System & Elevator Interaction

Several avenues of future work can be explored to improve the vision system and the robot's interaction with the elevator. First, improving the non-ML-based vision pipelines would be helpful, especially given the sensitivity of the button pusher to lighting conditions in Unity Hall. This could involve tuning color thresholds, adding extra steps to the raw OpenCV pipeline, or refactoring this process entirely by training a YOLO detection model to perform end effector detection. Additionally, further tuning the PID controllers for the robot when autonomously driving toward the push buttons or open elevator doors would improve the robot's control during elevator operations, and would reduce the risk of hitting something.

Currently, due to various safety concerns, the robot avoids elevator operations when there are people around the buttons or inside the elevator. Otherwise, the robot's large chassis could pose a risk to people, and it is a difficult problem to deduce whether or not the robot has space to do its elevator operations, or whether or not humans are making room for the robot inside the elevator. Future work could focus on developing more advanced reasoning capabilities for these scenarios, though this may require designing a more maneuverable robot with a smaller chassis to more effectively emulate humans, who are able to contort their bodies to fit.

Another possible avenue is to implement Visual Question Answering (VQA) capabilities to improve the robot's perception and cognition skills [39]. By enabling the robot to query a powerful model with questions about its environment (e.g. "What is this sign?", or "What does this symbol indicate?" accompanied by an image), VQA would provide a means for Tori to better understand its environment, allowing the robot to generalize better to other environments outside of Unity Hall.

4.3 Human-Robot Interaction

4.3.1 Graphical User Interface

To improve the user experience with the GUI, it may be beneficial to design and insert a visual layout of the rooms and their locations on each floor. Some users mentioned that the icons user to indicate the stairs and elevators were new to them, so a clickable layout with the room locations could help solve this issue as well. Due to the limited screen size, all of the rooms for each floor were unable to fit on one

page. A scroll bar on the side of the screen allowed the user to scroll to view all the rooms available to them, however some users found the scroll bar difficult to find and use. Additionally, although the GUI provides visual feedback on arrival, it may be beneficial to incorporate speech synthesis to provide verbal feedback when the user interacts with the GUI. The GUI can also be improved to provide different faces for Tori depending on her state. For instance if she is stuck behind an obstacle, having a confused or thinking face may help indicate to the user that she is waiting for the obstacle to move.

4.3.2 Verbal System

Given more time, we would have been able to improve the verbal system. Using a better tuned speech-to-text library could have improved reliability and performance since there were times when off-the-shelf Vosk did not work well with transcribing certain pronunciations or accents. Occasionally, this influenced Tori's ability to accurately understand words and subsequently navigate to rooms or answer questions properly. A solution would be to train the Vosk model to handle noise better. Another issue we had during testing was background noises getting picked up by the microphone. We attempted using a noise-cancelling microphone, but that did not fully block out all unnecessary noise. While the quality of the microphone could have been improved, using additional techniques, such as Voice Activity Detection (VAD) to focus on speech segments, could have created more reliable transcriptions. Also, the use of speech diarization to track individual voices and assign each speaker an "identity" could significantly improve the user experience and give Tori more abilities as a social robot.

One potential idea for improving human-robot interaction is integrating visual feedback with Tori's vocal capabilities, such as adding real-time subtitles of the user and Tori's conversation to the GUI. This approach would improve interaction in noisy environments and allow users to immediately identify when their speech was incorrectly transcribed. Additionally, adding visual or auditory cues on Tori, such as changing the LED lights and patterns or adding a noise to notify the speaker when Tori starts and stops talking, would allow for more seamless interaction between the user and Tori.

4.4 Control System

The low-level control system for Tori is currently stable, so future work should focus on improving reliability and reducing resource usage rather than expanding functionality. During testing, we encountered recurring issues with the Micro-ROS agent. Specifically, Fast DDS automatic discovery would occasionally fail, preventing topic propagation between nodes, and the agent itself would sometimes become unresponsive,

requiring a manual restart. These issues were not isolated to the microcontroller nodes (Picos) but affected all nodes onboard the robot. While switching to a different DDS (Data Distribution Service) implementation could improve reliability, the Micro-ROS agent currently only supports Fast DDS.

Furthermore, the Micro-ROS agent imposes a surprisingly high CPU and memory load on the Jetson devices. To address both performance and reliability concerns, we recommend migrating the control system to either a custom serial transport or a fully switched Ethernet-based network. This change would enable the use of alternative DDS backends, improving communication robustness across the robot's systems.

5 Conclusion

This project successfully produced Tori, an autonomous robotic tour guide capable of navigating the five floors of Worcester Polytechnic Institute's Unity Hall and interacting naturally with users. By combining ROS 2 navigation, YOLO-based object detection, OpenCV pipelines, and a responsive human-robot interaction system, Tori demonstrates a cohesive integration of perception, planning, and communication. The robot reliably moves between floors using elevator interaction, detects and presses elevator buttons, and communicates using a touchscreen interface and onboard voice recognition backed by a local large language model.

The technical design prioritized modularity, real-time performance, and safe operation. The use of Micro-ROS enabled distributed control between high-level logic and embedded microcontrollers, though some instability with the Fast DDS transport layer was observed during deployment. Similarly, while the visual button detection and manipulator alignment performed well in practice, performance could be improved by incorporating depth sensing or retraining models with more environment-specific data. Additionally, the resource usage of the Micro-ROS agent on Jetson platforms presents a future optimization opportunity.

Tori validates the viability of a low-cost, extensible tour guide robot in semi-structured indoor spaces using open-source tools and commercial hardware. The system provides a foundation for future enhancements such as improved reliability, dynamic content generation, and support for localization in dynamic environments. Tori's success demonstrates not only the technical feasibility of the concept but also its potential to serve as a blueprint for similar autonomous service robots in educational, public, or commercial facilities.

References

[1] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022.

[2] Y. Liu, S. Zhang, Y. Zhao, and R. Li, "Amcl: Attention-based multi-scale context learning for semantic segmentation," in *2021 IEEE/CVF International Conference on Computer Vision Workshops (ICCVW)*, pp. 3021–3030, IEEE, 2021.

[3] O. S. V. Foundation, "Opencv: Open computer vision library," 2025. Accessed: 2025-05-02.

[4] A. Ming and H. Ma, "A blob detector in color images," in *Proceedings of the 6th ACM International Conference on Image and Video Retrieval*, CIVR '07, (New York, NY, USA), p. 364–370, Association for Computing Machinery, 2007.

[5] K. A. M. Said and A. B. Jambek, "Analysis of image processing using morphological erosion and dilation," *Journal of Physics: Conference Series*, vol. 2071, p. 012033, oct 2021.

[6] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," 2015.

[7] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," 2016.

[8] C.-Y. Wang, I.-H. Yeh, and H.-Y. M. Liao, "Yolov9: Learning what you want to learn using programmable gradient information," 2024.

[9] S. M. University, "Elevator button recognition." https://universe.roboflow.com/sun-moon-university/elevator-button-recognition, 2022.

[10] O. S. V. Foundation, "Basic concepts of the homography explained with code," 2025.

[11] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," *arXiv preprint arXiv:2102.00928*, 2021.

[12] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, "Understanding deep learning requires rethinking generalization," *arXiv preprint arXiv:1611.03530*, 2017.

[13] M. Marge, C. Espy-Wilson, N. G. Ward, A. Alwan, Y. Artzi, M. Bansal, G. Blankenship, J. Chai, H. Daumé, D. Dey, M. Harper, T. Howard, C. Kennington, I. Kruijff-Korbayová, D. Manocha, C. Matuszek, R. Mead, R. Mooney, R. K. Moore, M. Ostendorf, H. Pon-Barry, A. I. Rudnicky, M. Scheutz, R. S. Amant, T. Sun, S. Tellex, D. Traum, and Z. Yu, "Spoken language interaction with robots: Recommendations for future research," *Computer Speech and Language*, vol. 71, p. 101255, 2022.

[14] S. Macenski and I. Jambrecic, "SLAM Toolbox: SLAM for the dynamic world," *Journal of Open Source Software*, vol. 6, no. 61, p. 2783, 2021.

[15] S. Macenski, F. Martin, R. White, and J. Ginés Clavero, "The marathon 2: A navigation system," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020.

[16] U. of Ottawa, "What is camera calibration?." https://www.site.uottawa.ca/~petriu/HomTransfMatrix.pdf.

[17] Mathworks, "What is camera calibration?." https://www.mathworks.com/help/vision/ug/camera-calibration.html, 2025.

[18] "Stack overflow," 2013.

[19] C. Knospe, "Pid control," *IEEE Control Systems Magazine*, vol. 26, no. 1, pp. 30–31, 2006.

[20] N. Platonov and N. Dunin-Barkowski, "Vosk: Offline speech recognition toolkit," 2020. Alpha Cephei Inc.

[21] E. Gölge, S. Majumdar, A. Nauwelaerts, J. O'Keeffe, E. Casanova, and D. Le, "Tts: Text-to-speech for all," 2023. Coqui AI.

[22] S. Su, "pyt2s: Python script to convert text to speech," 2023. GitHub repository.

[23] Z. Zhang, J. Li, A. Albalak, X. Song, J. Cao, M. Mahdavi, A. Rawat, Y. Chen, S. Zhuang, S. Yadlowsky, A. Menon, D. Card, and M. Rathkopf, "Tinyllama: An open-source small language model," *arXiv preprint arXiv:2401.02385*, January 2024.

[24] P. Dubey, H. Touvron, L. Barrault, T. Lavril, T. Scialom, R. Pasunuru, A. Raju, P. Robion, M. Labeau, *et al.*, "Llama 3.2: Advancing open language models with enhanced reasoning," *arXiv preprint arXiv:2407.21783*, July 2024.

[25] G. Team, Google, and DeepMind, "Gemma 3: An open family of reasoning models," *arXiv preprint arXiv:2410.07176*, October 2024.

[26] M. AI, L. Team, H. Touvron, T. Lavril, *et al.*, "Llama 3.1: Meta's latest open language models," *arXiv preprint arXiv:2407.35174*, July 2024.

[27] C. Feinberg, A. Troynikov, and J. Huber, "Chromadb: An open-source embedding database," 2023. Chroma.

[28] N. AI, "Nomic embed: Text embeddings for production ai," 2024.

[29] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*, 2019. Model: all-MiniLM-L6-v2.

[30] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 3982–3992, Association for Computational Linguistics, November 2019.

[31] J. Johnson, M. Douze, and H. Jégou, "Faiss: A library for efficient similarity search and clustering of dense vectors," in *International Conference on Learning Representations*, 2019.

[32] NVIDIA Corporation, *CUDA Runtime API*, 2025. Version 12.9.

[33] eProsima, "Fast dds: The most complete dds." https://github.com/eProsima/Fast-DDS, 2025. Accessed: 2025-05-02.

[34] micro-ROS Development Team, "micro-ros: Embedded ros 2 for microcontrollers." https://micro.ros.org/, 2025.

[35] Raspberry Pi Ltd., *Pico SDK: Hardware APIs – PIO (Programmable I/O)*, 2025. Accessed: 2025-05-02.

[36] Raspberry Pi Ltd., *Pico SDK: Hardware APIs*, 2025. Accessed: 2025-05-02.

[37] D. Garcia and M. Kriegsman, "Fastled: Fast, efficient, easy-to-use arduino led library." https://fastled.io/, 2025. Version 3.9.16, Accessed: 2025-05-02.

[38] FRC Team 716, "Bill of materials for destination deep space competition robot." Unpublished internal document, 2019. Excel spreadsheet, accessed 2025-05-02.

[39] S. Antol, A. Agrawal, J. Lu, M. Mitchell, D. Batra, C. L. Zitnick, and D. Parikh, "Vqa: Visual question answering," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, December 2015.