ChatGPT

Production-Ready ROS 2 Humble Navigation Stack for Vstone 4WDS Rover X40A

1. Navigation Architecture (Mapping, Localization, Planning, Control, Perception, Safety)

Overall Design: For reliable indoor deliveries, we use the ROS 2 Navigation2 stack (Nav2) on Humble, which is a **production-grade** framework used by many companies [1]. The Rover X40A (a four-wheel independent steering base) is equipped with a 360° Livox MID-360 LiDAR for environment perception, plus wheel encoders (and an IMU). Our architecture follows best practices: - **Mapping & Localization:** We employ **SLAM Toolbox** for 2D SLAM to build an occupancy grid map [2]. SLAM Toolbox is the officially recommended SLAM solution in ROS 2 Nav2 [2], capable of **online mapping** and later **localization-only mode**. Initially, the robot will **explore to map** the ~10×10 m area (since SLAM was preferred over pre-built maps). Once a map is created, we switch to **localization mode** for routine operation using Adaptive Monte Carlo Localization (AMCL) on the saved map [3]. This two-phase approach ensures accurate localization without continual map distortion by moving people or changes. (It's possible to navigate while SLAM updates the map, but using a fixed map with AMCL is generally more robust for repetitive deliveries.) - **Global Path Planning:** We use Nav2's **Smac Planner**, specifically the **Hybrid-A** *plugin, which is optimized for car-like and non-holonomic robots* [4] [5]*. The Rover X40A can move omnidirectionally (all wheels steer)* [6]*, but it behaves mostly like an Ackermann/differential drive for path planning. Smac Hybrid-A* considers the robot's turning radius and produces smooth, feasible paths without oscillations [5]. (Alternative global planners like NavFn or 2D grid A *are less optimal; Smac's Hybrid-A* and State Lattice planners handle Ackermann kinematics and arbitrary footprints better [4] [7].) - **Local Control:** We select the **Regulated Pure Pursuit (RPP)** controller, a trajectory follower introduced in Nav2 for robust and easier tuning. RPP is well-suited to Ackermann and differential drives, as it follows the global path with adaptive speed control and handles **rotate-in-place** at the goal if needed. This avoids the tuning difficulties and oscillation issues sometimes seen with the DWB (Dynamic Window) trajectory planner. In our indoor, low-speed (< 1 m/s) scenario (payload is light, speed is slow per requirements), RPP provides smooth control to waypoints with safety regulation (slowing down near obstacles). We configure RPP's lookahead distance and goal tolerances so that the robot does not overshoot or oscillate. (For completeness, Nav2 also offers **DWB** and **MPPI** controllers; DWB could be constrained to zero lateral velocity for Ackermann, but RPP's simpler behavior is preferred. MPPI, a model-predictive controller [8], is powerful but adds complexity and is overkill for a slow indoor rover.) - **Perception & Costmaps:** The MID-360 LiDAR provides a dense 3D point cloud (360° horizontal FoV, ~60° vertical [9]). We integrate this into Nav2's 2D **costmap** via a **voxel layer** to handle 3D data [10]. The costmap stack has: a **Static Layer** (for the SLAM-built map), an **Obstacle/Voxel Layer** (for real-time sensor obstacles), and an **Inflation Layer** (to pad obstacles for safety) [10] [11]. We configure the voxel layer to project 3D points into the 2D costmap, marking obstacles and clearing free space by raytracing [10]. **Ground filtering** is crucial on flat floors: we will set a minimum obstacle height (e.g. 5 cm) so that LiDAR returns from the floor are ignored, preventing "floor clutter" in the costmap. Additionally, we downsample the point cloud (using a VoxelGrid filter) to reduce load, and we may convert it to a 2D laser scan for SLAM/localization (e.g. using `pointcloud_to_laserscan`). The costmap's **height clipping** (min_z, max_z) will be tuned to include obstacles roughly within the robot's height (e.g. 0.1 m < points <

0.5 m considered obstacles, ignoring ceiling or table-top points if the robot can pass under). We also expand obstacles with an inflation radius slightly larger than the robot's half-width to maintain a buffer (e.g. inflation radius ~0.3 m for a ~0.5 m wide rover, ensuring ~10 cm clearance on each side as required). These perception filters and costmap parameters (discussed further in Section 3) ensure the rover only treats true obstacles (walls, furniture, people) as lethal and can safely navigate around them. - **Safety and Recovery:** Safety is paramount for deliveries around people. First, the Rover X40A hardware has a **physical E-stop button** standard [12] – we mount it accessibly so human operators can halt the robot immediately if needed. In software, we incorporate Nav2's **Collision Monitor** node as an extra safety layer. The Collision Monitor subscribes to LiDAR data and the outgoing `/cmd_vel` commands and can automatically stop or slow the robot if an obstacle suddenly appears too close [13] [14]. We define a safety "stop zone" polygon immediately around the robot (e.g. a 0.5 m radius) such that any intrusion (like a person stepping in front) triggers an immediate zero-velocity command. We also set conservative speed limits: the base's max translational speed might be limited to ~0.5 m/s and rotational speed ~0.5 rad/s in Nav2, given the requirement for slow, safe motion. Nav2's planner and controller have built-in **recovery behaviors** (like spinning in place or reversing if path is blocked). These can be enabled in the Behavior Tree (BT) so the robot attempts to unstuck itself if it gets stuck or oscillates. We will fine-tune the **goal tolerance** such that the robot does not insist on perfect orientation at waypoints (since "close enough is ok") – for example, allow a positional error of up to 5–10 cm and orientation error ~15° at the final goal. This prevents excessive correction maneuvers and improves goal success rate for deliveries. - **Frame Transform (TF) Setup:** We use a standard ROS TF tree: `map` → `odom` → `base_link` → sensor frames. The SLAM toolbox or AMCL provides the `map→odom` transform as it localizes the robot on the map. The rover's odometry (from wheel encoders/IMU) provides `odom→base_link`. A **robot_state_publisher** will broadcast fixed transforms (e.g. LiDAR frame relative to base_link, which we calibrate precisely). Ensuring consistent frames is vital: misconfigured frames commonly cause drift or costmap issues, so we double-check sensor extrinsic calibration (e.g. LiDAR mounted at known offset). We align `base_link` with the robot's geometric center at floor level, and define the LiDAR's frame according to its mount (e.g. `livox_frame` at ~0.2 m above base_link). All sensor data (LaserScan or PointCloud2) is reported in its frame with a static TF to base_link. Nav2 will use `base_link` as the robot frame and `map` as the global frame. We set the costmap `robot_base_frame = base_link` and `global_frame = map` (for global costmap) / `odom` (for local costmap in rolling mode). This prevents TF inconsistencies. If any TF loop or discrepancy arises (a common pitfall), we will resolve it by having only one source for each transform chain (e.g. disable duplicate odometry TF from any secondary source). - **Integration with Vstone Rover Interface:** The official Vstone ROS interface for X40A needs to mesh with Nav2. The Rover's motor controller (VS-WRC058 board) accepts velocity commands on a topic (in ROS1 this is `/rover_twist`) and provides odometry data. In ROS 2, Vstone provides a sample package `fwdsrover_xna_ros2` [15]. This package likely defines a node that subscribes to a Twist message (geometry_msgs/Twist) to command the wheel motors, and publishes some form of odometry. Indeed, in the ROS1 driver, the hardware subscribed to `/rover_twist` (Twist) and published `/rover_odo` (Twist of wheel speeds), which a helper node converted to `/odom` (nav_msgs/Odometry) [16]. In ROS 2, we expect a similar interface: for example, the sample launch remaps `cmd_vel` to `rover_twist` [17], indicating Nav2's output velocity commands can be sent to the rover by a simple remapping. We will **remap Nav2's `/cmd_vel` output to `/rover_twist`** so that the Vstone driver receives it. The message type is standard geometry_msgs/Twist (the X40A interprets linear x, linear y, angular z – since it has omnidirectional steering, a lateral y velocity is possible if given). The rover's driver in ROS2 likely publishes an odometry topic (`/odom` of type nav_msgs/Odometry) directly, or at least publishes wheel encoder info that we can feed into an odometry node. We will confirm the exact topics from the Vstone repo (since the user has it). If it does not publish a full `Odometry`, we can use **robot_localization** (EKF) to fuse the wheel encoder twist (/rover_odo Twist) and IMU data to produce a

stable /odom. In summary, the Nav2 stack will **subscribe to** `/odom` (for localization and velocity feedback) and **publish** `/cmd_vel` (remapped to `/rover_twist` for the base). **TF**: The base driver or our launch should also broadcast `odom→base_link` continuously, typically done by the odometry node. We ensure the `odom` frame ID matches between Nav2 and the Vstone driver. If the Vstone repo is ROS1-only, we would bridge topics via `ros1_bridge` (e.g. bridging `/rover_twist` and `/odom`), but since an official ROS2 package exists [15], we prefer to use that natively to avoid the complexity of a ROS1 bridge. Compatibility is achieved by **topic remapping and message type alignment**, which fortunately appear standard (geometry_msgs and nav_msgs). - **Summary:** The architecture consists of modular ROS 2 nodes: **Livox Driver** (LiDAR point cloud publisher), **pointcloud filter** (ground removal, etc.), **tf** (static publishers for sensor transforms, plus odometry tf), **SLAM / Localization** (SLAM Toolbox or AMCL), **Nav2 stack** (containing costmaps, planner, controller, BT navigator, recovery behaviors), **Safety nodes** (Collision Monitor, possibly a watchdog on commands), and the **Vstone base driver** (motor controller interface). This yields a layered system where sensors feed into costmaps, costmaps feed into planning, which outputs velocity commands to the base. By adhering to ROS 2 best practices and properly tuning each component, the Rover X40A will reliably navigate to waypoints while avoiding obstacles and ensuring safety.

2. Choice of Libraries & Rationale (SLAM vs Odometry, Planners, Controllers, Filters, Base Control)

Mapping & Localization Libraries: We choose **SLAM Toolbox** for 2D SLAM mapping, instead of alternatives like Cartographer or pure LiDAR odometry. *Trade-offs:* SLAM Toolbox (Humble version) is well-maintained, integrates smoothly with Nav2, and supports large maps and loop closure if needed [2]. It also allows saving the map and resuming in localization mode. **Cartographer** could perform 2D SLAM with loop closure as well, but its ROS2 support is less straightforward and it introduces more complexity (e.g. tuning 3D scan matching if we used the 3D point cloud). **LiDAR-Inertial Odometry (LIO):** Tools like **FAST-LIO2** or **LOAM** can provide real-time 3D SLAM; indeed, there are ROS2 demos of FAST-LIO2 with the Livox MID360 [18]. Those can give very accurate odometry and a point cloud map. However, implementing a 3D LIO pipeline for a 2D navigation task is overkill – it demands more computation and doesn't directly yield a 2D occupancy grid for Nav2. Instead, we stick to **2D SLAM** which is sufficient for a flat indoor environment and far simpler to tune. The MID-360's pointcloud is converted to a 2D **LaserScan** (either by SLAM Toolbox's scan topic subscription or using `pointcloud_to_laserscan`). This approach filters out the floor and ceilings, focusing on horizontal structures for mapping. After mapping, **AMCL** (Adaptive Monte Carlo Localization) is used on the static map for long-term localization [3]. AMCL is robust for indoor use and computationally light, and it frees us from continuously running SLAM (which might incorporate moving persons as fake "walls" in the map). If the user ever needs the robot to **build and navigate on the fly** (e.g. in a new environment), Nav2 can be run in a "SLAM mode" where SLAM Toolbox and Nav2 run concurrently (Nav2 uses the evolving map via the map_topic). This is advanced and requires disabling the static map layer initially – an approach described in the Nav2 tutorials [19]. For production in a known building, the recommended path is: run SLAM first, **verify the map**, and then use that map for navigation with AMCL for repeatability.

Global Planner (Nav2): We use **SmacPlannerHybrid** (Hybrid A*) plugin.* Trade-offs: *The Hybrid-A* planner searches in SE2 (position + heading) space, respecting a minimum turning radius for the rover [4] [5]. This produces feasible paths that a non-holonomic vehicle can follow without frequent re-planning. Given the X40A *can* move laterally (omnidirectional), one might argue a simpler 2D grid planner (like **SmacPlanner2D** or Dijkstra/NavFn) could work, treating the robot as holonomic. However, if we treat it as fully holonomic,

the planner might create paths that assume the robot can strafe freely, which, while theoretically possible (the wheels could be turned for sideways motion), would require complex wheel coordination and likely slow down the motion. We prefer to constrain planning to more car-like paths for reliability. Smac Hybrid-A *has parameters for minimum turn radius and can generate smooth curves; it naturally avoids path solutions that are too tight for the rover. Another option is SmacPlannerLattice, which allows specifying a lattice of motion primitives (including sideways moves) for omnidirectional robots* [20] [21]*. Lattice planning could fully exploit 4WDS's capabilities (e.g. crab driving for fine positioning). The trade-off is complexity: one must provide a motion model/control set. For initial deployment, Hybrid-A* is easier (we can set it to use Dubins or Reeds-Shepp curves for smooth turns). The **NavFn** (Dijkstra) global planner was the old default; it finds shortest paths on the grid but produces jagged paths and doesn't consider orientation. We trade that out for Smac to get better quality paths and to avoid corners that might snag the robot's rectangular footprint. Smac's smoothing module will also post-process paths to avoid unnecessary oscillations [22]. We will tune Smac's costmap downsampling and analytic expansion to ensure planning over the 10×10 m map is fast (< 100 ms). With low speeds and a small map, global re-planning at 1 Hz is feasible so the robot can adapt if dynamic obstacles (people) block the way.

Local Planner (Controller) Choices: We select **Regulated Pure Pursuit (RPP)** as the controller. *Trade-offs:* **DWB (Dynamic Window Approach)** is Nav2's legacy trajectory planner, which samples velocities and scores trajectories. It works for differential drive robots but can struggle for Ackermann steering unless carefully constrained (we would have to set `y_vel_max = 0` to prevent strafing and tune trajectory critics to avoid oscillation). DWB often requires tuning multiple cost weights (path alignment, goal alignment, obstacle costs, etc.) and can get "No valid trajectory" errors if parameters are off [23]. **RPP**, in contrast, is a simpler follower: it computes a target point on the path and uses pure pursuit geometry to steer towards it, regulating speed by curvature and proximity to obstacles. For our slow-moving rover, RPP provides a **more stable and easily interpretable behavior**. RPP includes safety features like slowing down if the path curvature is high (preventing overshoot) and stopping if an obstacle is too close ahead [13]. This fits well with our safety-first approach. Another alternative is the **MPPI controller** (Model Predictive Path Integral) [8]. MPPI can optimize control inputs over a short horizon and handle dynamic constraints elegantly (it could theoretically incorporate the rover's acceleration limits and even four-wheel steering dynamics). However, MPPI is computationally heavier (running many trajectory simulations each cycle) and its benefits show up more at high speeds or with complex dynamics. Since our payload is light and speed is intentionally slow, MPPI's advantages aren't critical; RPP will suffice and is much simpler to configure. RPP also has a parameter `rotate_to_heading` which we'll use to allow the robot to smoothly rotate in place at the end of a trajectory to face the final orientation if needed (though we have loose orientation tolerance, it's nice to stop roughly facing forward for the next segment). We'll set RPP's lookahead distance around 0.5–0.8 m (for a ~0.5 m robot) to balance responsiveness vs. smoothness. We'll also enable its collision stopping feature so it halts if any obstacle enters a safety buffer in the costmap.

Perception Filters: We employ a **voxel grid downsampling** on the LiDAR's raw point cloud (e.g. downsample to ~0.05 m voxels). The Livox MID-360 can produce a high point rate; downsampling reduces costmap update load without sacrificing map accuracy significantly in a sparse indoor environment. **Ground removal** is important: since the LiDAR has a 60° vertical FoV, it will see the floor near the robot. If unfiltered, those points would be marked as obstacles in the costmap's voxel layer. We have multiple strategies: - Using the costmap's **min_obstacle_height** parameter: e.g. set `min_obstacle_height = 0.1` m, and `max_obstacle_height = 1.0` m, assuming the LiDAR is mounted ~0.2 m high. This way any returns very close to the ground (<10 cm) are ignored [24] [25]. This simple method works if the floor is flat and returns are concentrated at 0 height. - Alternatively, use PCL

filters: e.g. a **PassThrough filter** to drop points with z below a threshold, or a plane model segmentation to explicitly remove the ground plane. Given our flat surface, a passthrough might suffice. - Nav2's **voxel layer** by itself doesn't automatically distinguish ground, but it can clear freespace via raytracing. We will tilt the LiDAR if needed so it sees obstacles above ground but not too much floor. However, likely the LiDAR is mounted horizontally. In that case, points hitting the floor might only occur at the extreme near range due to beam divergence. We choose a conservative filtering: drop points with z < ~0.05–0.1 m (just above floor) to eliminate floor hits, and points above ~0.5–1 m (because anything above the robot's top is not relevant for collision). This effectively creates a "slice" of points at robot-relevant heights (like leg level for humans and furniture). We'll verify that crucial obstacles (table edges, etc.) are within that band – if not, we'll raise max height.

After filtering, the point cloud is fed to the costmap's **obstacle layer or voxel layer**. We enable **marking and clearing** so that as the robot moves, cleared space becomes free [26]. The **inflation layer** is set with an inflation radius ~0.2–0.3 m beyond the robot's footprint. The Rover X40A's footprint (approx 0.5 m wide) will be defined in a polygon or circle. We add a small cushion (10 cm) beyond half-width to account for any lateral error and ensure at least 10 cm clearance to obstacles (as required). This inflation radius also helps the planners avoid getting too close to walls, preventing side scrapes. We can tune the **cost scaling factor** of inflation so that the cost decays such that the robot prefers staying, say, >0.1 m away even if it technically can squeeze closer.

For moving obstacles (people), our approach is reactive: the LiDAR will detect the person and mark them in the **local costmap**, causing the local planner (RPP) to slow or reroute around them. The global planner can replan if the way is blocked longer-term. We set Nav2's **obstacle persistence** and **decay** such that once a person moves, the costmap frees up that space after a short period (e.g. obstacle layer observation persistence 2–3 seconds). If needed, we could incorporate a **Spatio-Temporal Voxel Layer (STVL)** plugin [27] for dynamic obstacle handling, which retains a short memory of obstacles and can account for their velocities. But given moderate foot traffic, the default approach suffices: the robot will either wait or slowly navigate around a person using the local planner and then resume its global path.

Base Control (Hardware Interface): The Rover X40A's drive system is **4-wheel drive with independent steering** on each wheel [6]. The Vstone control board likely abstracts this and takes a simple Twist command. We confirm that the `/rover_twist` message uses **geometry_msgs/Twist** with `linear.x` (forward/back), `linear.y` (strafe), and `angular.z` (rotation) in the base frame [28]. This is analogous to a holonomic base. We do not need a custom ros2_control plugin if we use Vstone's provided node – it internally handles the kinematics (possibly using an Arduino library on the VS-WRC058 board, as mentioned by Vstone [29]). However, it's worth noting how **odometry** is computed. In Vstone's ROS1 sample, the board did **not directly provide pose**; it provided wheel distances which the ROS node integrated to produce odom [30]. In ROS2, the `fwdsrover_xna_ros2` package likely includes an odometry node (perhaps named `pub_odom` similarly) that subscribes to the wheel encoder info and publishes `nav_msgs/Odometry` on `/odom`. This odometry is differential (no absolute localization, just relative motion), which is fine because Nav2's AMCL (or SLAM) will correct any drift by adjusting the `map→odom` transform. We will **use the provided odometry** for Nav2's state estimation. If needed, we can fuse the **IMU** from the Livox (the MID-360 includes a built-in IMU [31]) to stabilize yaw during slow rotations – the **robot_localization** EKF node could fuse `/imu/data` with wheel odometry for a smoother odom frame. This might improve localization during temporary LiDAR occlusions or slippage. It's an optional enhancement; for initial simplicity, wheel odometry alone (with periodic LiDAR re-localization from AMCL) should keep the robot within the 10 cm accuracy requirement.

If we wanted a more direct low-level control via ros2_control, we'd need to model the steering joints and wheel joints. That's complex since 4WDS isn't a standard controller in ros2_control (it's neither pure diff nor pure omnidirectional without special plugins). Vstone's driver saves us from implementing this by providing a higher-level interface. We just ensure our **Nav2 controller outputs** obey the rover's limits (we will set `max_vel_x`, `max_vel_y`, `max_rotational_vel` in Nav2 params to within what the hardware supports – presumably the X40A can do ~1 m/s max per specs, but we'll cap at ~0.5 m/s for safety). Also, because the rover can move omnidirectionally, we allow Nav2's local planner to command `y` velocity for minor path adjustments (this can help avoid having to rotate the entire base just to move a few centimeters laterally). The RPP controller doesn't naturally produce lateral moves (it will tend to rotate then move forward). If we did want the robot to sometimes execute a sideways correction, we could consider using a specialized **omnidirectional controller** or switch to a holonomic planning mode for final approach as hinted by some users [32]. For now, we prioritize simplicity: the robot will mostly face the direction of travel (which is efficient and reliable).

Finally, we'll integrate the **official Vstone communication repository** (which the user has). For ROS2, it likely provides launch files to bring up the motor interface and maybe a teleop. We will incorporate or reference those in our launch (Section 5). If, hypothetically, the user's repo was ROS1 (older), we would add a **ROS1–ROS2 bridge**. That bridge would handle at least two topics: `/rover_twist` (geometry_msgs/Twist) and `/odom` (nav_msgs/Odometry). The standard dynamic_bridge can bridge those message types automatically. But running a bridge adds latency and complexity, so we prefer the ROS2 driver. Since Vstone announced **ROS2 firmware availability in 2023**, we will proceed with ROS2 native control.

In summary, our chosen libraries/components and their trade-offs: - **SLAM Toolbox** (2D SLAM) – pros: officially recommended [2], easy ROS2 integration, good for static maps; cons: limited to 2D. - **AMCL** – pros: simple and reliable localization on known map; cons: requires pre-mapping. - **Smac Planner (Hybrid-A*)** – pros: kinematically feasible paths for 4WDS [5]; cons: more params than NavFn, but better path quality. - **Regulated Pure Pursuit** – pros: simple tuning, safe regulation, good for Ackermann; cons: may not handle very complex maneuvers that require path recomputation (which is mitigated by letting global planner replan if needed). - **Voxel/Obstacle Layer** – pros: handles 3D LiDAR, can clear 3D obstacles; cons: slightly higher CPU and needs tuning for ground filter. - **Inflation Layer** – pros: adds safety buffer; cons: too large inflation can restrict narrow passage navigation (we tune it moderately). - **Nav2 BT & Recovery** – we will use the default **Behavior Tree** (Navigate to Pose) which includes retries and resets costmaps. We ensure the **spin-in-place** behavior is enabled so if the rover gets stuck it can spin to look for alternatives. - **Collision Monitor** – pros: extra safety net; cons: needs careful polygon tuning to avoid false positives. We will define a forward-facing polygon (like a virtual bumper) of perhaps 0.3–0.5 m that triggers a stop if an obstacle enters it suddenly.

All these chosen libraries are **compatible with ROS 2 Humble**. Nav2 on Humble (version 1.1.x) supports these plugins out-of-the-box, and Vstone's ROS2 package is designed for Humble as well (Ubuntu 22.04). In Section 8, we detail the exact versions and licensing (spoiler: mostly Apache 2.0, except SLAM Toolbox which is LGPL [33]).

3. Workspace Setup and Example Configuration (Humble)

We now provide a **runnable Minimum Working Example (MWE)** for the Rover's navigation stack. The ROS 2 workspace is organized into a few packages for clarity:

```
rover_nav_ws/            # Colcon workspace
├── src/
│   ├── rover_description/        # URDF/Xacro files and meshes
│   ├── rover_bringup/            # Launch files and runtime configs
│   ├── rover_navigation/         # Nav2 configuration (behavior tree, params)
│   └── rover_waypoints/          # Example Python script for waypoint following
├── maps/                        # Saved map files (pgm + yaml)
└── rviz/                        # RViz config (for visualization)
```

URDF/Xacro (rover_description): We create a URDF model of the Rover X40A including the LiDAR. The URDF is simplified (since we do not need full mechanical simulation detail for Nav2). Key frames: - `base_link`: origin at the center of the rover's wheelbase on the ground. We assume the rover is roughly 0.5 m long/wide and ~0.2 m tall (X40A is smaller than X120A's 519×476 mm footprint [34], and carries ~40 kg payload [35]). - Four wheel links (optional for visualization) at the corners. We can attach them with fixed joints since steering is not simulated here. - `livox_frame`: attached to base_link via a fixed transform (e.g. `(0.0, 0.0, 0.2)` if LiDAR is mounted at the center top). The MID-360 provides a 360° horizontal scan, so we assume it's mounted upright, spinning around vertical axis. We orient its frame with X-axis forward, Y-axis left, Z-axis up, as is ROS standard for sensors. If the Livox driver expects a different frame orientation, we adjust accordingly. - In URDF, we include an `<inertial>` and `<collision>` for base_link (a box roughly covering the robot footprint) – this helps Gazebo simulation for collisions. - The URDF defines the **footprint** either via `<geometry>` shapes or we will specify a footprint separately in Nav2 costmap params. (Nav2 can use the robot's collision geometry from URDF if `use_polygon_footprint` is set, or we can input a footprint in YAML.)

Minimal URDF snippet (in Xacro for reuse):

```
<robot name="rover_x40a" xmlns:xacro="http://ros.org/wiki/xacro">
  <!-- Base Link -->
  <link name="base_link">
    <inertial>...</inertial>
    <visual>
      <!-- a simple box visual for the chassis -->
      <geometry><box size="0.5 0.4 0.2"/></geometry>
      <material name="grey"><color rgba="0.6 0.6 0.6 1.0"/></material>
    </visual>
    <collision>
      <geometry><box size="0.5 0.4 0.2"/></geometry>
    </collision>
  </link>

  <!-- LiDAR sensor (Livox MID360) -->
  <link name="livox_link">
    <visual>
      <geometry><cylinder length="0.1" radius="0.05"/></geometry>
      <material name="black"><color rgba="0 0 0 1"/></material>
```

```
    </visual>
  </link>
  <joint name="lidar_mount" type="fixed">
    <parent link="base_link"/>
    <child link="livox_link"/>
    <origin xyz="0 0 0.2" rpy="0 0 0"/>  <!-- LiDAR mounted 20cm above base
center -->
  </joint>

  <!-- Wheels (visual only, no steering joints modeled) -->
  <link name="wheel_front_left"><visual>...</visual></link>
  <joint name="fixed_wheel_fl" type="fixed"><parent link="base_link"/><child
link="wheel_front_left"/>
    <origin xyz="0.25 0.2 0" rpy="0 0 0"/></joint>
  <!-- Similarly define wheel_front_right, wheel_rear_left, wheel_rear_right
with fixed joints -->

  <!-- Sensors -->
  <gazebo reference="livox_link">
    <sensor type="gpu_ray" name="gpu_laser">
      <update_rate>10.0</update_rate>
      <range>
        <min>0.1</min><max>20.0</max><resolution>0.01</resolution>
      </range>
      <scan>
        <horizontal>
          <samples>720</samples><resolution>1</resolution>
          <min_angle> -3.14159 </min_angle><max_angle> 3.14159 </max_angle>
        </horizontal>
        <vertical>
          <samples>16</samples><resolution>1</resolution>
          <min_angle>-0.5236</min_angle> <!-- -30 deg -->
          <max_angle>0.5236</max_angle>  <!-- +30 deg -->
        </vertical>
      </scan>
      <plugin name="gazebo_ros_gpu_laser_controller"
filename="libgazebo_ros_gpu_laser.so">
        <ros>
          <namespace>rover</namespace>
          <remapping><remap from="scan" to="scan"/></remapping>
        </ros>
        <frameName>livox_link</frameName>
      </plugin>
    </sensor>
  </gazebo>
</robot>
```

This URDF: - Defines a LiDAR sensor for Gazebo simulation (`gpu_ray` with 720 horizontal beams and 16 vertical layers, ~10 Hz). It publishes to `/scan` topic in the `rover` namespace. (We simulate a simplified LiDAR – 16-layer is just an example; the actual MID-360 has a non-repetitive scan pattern, but that detail can be abstracted as a 16-line 3D scan for simulation.) - The LiDAR plugin's `<frameName>` is set to `livox_link`, which means the `sensor_msgs/LaserScan` will have `frame_id = "livox_link"`. We ensure our Nav2 and SLAM config uses that (or we remap frame names).

Launch Files (rover_bringup): We create modular launch files and a top-level **"one-command"** launch.

Key launch files: - `vstone_base.launch.py`: Launches the Vstone Rover base driver. For example, if the package `fwdsrover_xna_bringup` has a launch file, we include it. We also use `ros2 param` or remappings to ensure it publishes odometry on `/odom` and expects commands on `/rover_twist`. If needed, we start a small bridge or converter node here (e.g. if the base only publishes a custom message, convert to Odometry). - `livox_driver.launch.py`: Launches the Livox ROS2 driver. Using **livox_ros_driver2** (version 1.2.4 recommended for Humble [9] [36]). We provide a config JSON for the MID-360 (device ID, IP/port or serial, etc.). The driver will publish `sensor_msgs/PointCloud2` on a topic (by default maybe `/livox/lidar` or `/livox/point_cloud`). We can remap it to `/points`. - `tf_static.launch.py`: We publish static transforms such as (`base_link` to `livox_link`) if not already handled by robot_state_publisher. Actually, we will run **robot_state_publisher** with the URDF, so static links (like LiDAR mount) will be published automatically. We also might need a static transform from `base_link` to a `base_footprint` if using one (sometimes `base_footprint` is on ground and `base_link` at center, but we set base_link at ground already, so not needed). - `nav2.launch.py`: Launches the Nav2 stack. We use `nav2_bringup::bringup_launch.py` or our custom version. It will start **nav2_amcl** (if localization mode) or **slam_toolbox** (if mapping mode) depending on a parameter. We pass our `nav2_params.yaml` (detailed below). We also start **rviz2** here for convenience (with a preconfigured view showing map, laser, path). - `waypoint_follower.launch.py`: Launches the example Python node that sends waypoints (this could also be run separately).

One-Command Bringup: We create `rover_navigation.launch.py` that **includes** all the above:

```
# rover_navigation.launch.py
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription, ExecuteProcess
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch_ros.actions import Node

def generate_launch_description():
    # Path to packages
    pkg_bringup = get_package_share_directory('rover_bringup')
    pkg_desc    = get_package_share_directory('rover_description')
    pkg_nav     = get_package_share_directory('rover_navigation')

    use_slam = LaunchConfiguration('use_slam', default='false')  # set true for
mapping mode

    # Load robot description
```

```
    rsp = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(os.path.join(pkg_desc, 'launch',
'rsp.launch.py'))
    )
    # Base driver (if needed)
    base = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(os.path.join(pkg_bringup, 'launch',
'vstone_base.launch.py'))
    )
    # Livox driver
    livox = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(os.path.join(pkg_bringup, 'launch',
'livox_driver.launch.py'))
    )
    # Nav2 (with slam or localization based on use_slam)
    nav2 = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(os.path.join(pkg_nav, 'launch',
'nav2.launch.py')),
        launch_arguments={'use_slam': use_slam}.items()
    )
    # (Optional) Waypoint follower
    waypoints = Node(
        package='rover_waypoints',
        executable='waypoint_follower.py',
        output='screen'
    )

    return LaunchDescription([
        rsp, base, livox, nav2, waypoints
    ])
```

When the user runs:

```
ros2 launch rover_bringup rover_navigation.launch.py use_slam:=false
```

it will start everything for normal navigation (assuming a map is already available on disk). Setting `use_slam:=true` would instead launch SLAM Toolbox in mapping mode (we implement that logic inside `nav2.launch.py` by conditionally including either the AMCL node or the SLAM node and configuring Nav2's map usage accordingly).

nav2_params.yaml (rover_navigation/config/nav2_params.yaml): This YAML contains tuned parameters for our scenario: - **Costmap Parameters:** - `global_costmap` is **static**, not rolling, covering the whole area (e.g. 15×15 m size with 0.05 m resolution). Static layer subscribes to `/map`. We disable the global costmap's obstacle layer if we rely on static map only for global planning (dynamic obstacles will be handled in local costmap). Alternatively, we can enable a global obstacle layer with a very low update frequency

(0.5 Hz) and large horizon, so that if a permanent obstacle appears it could replan globally. - `local_costmap` is **rolling_window: true**, with width/height ~3 m, centered on the robot, resolution 0.05 m. It has no static layer (since we want it to react to sensor data only), but has obstacle/voxel and inflation layers. We set `publish_frequency` to e.g. 5 Hz so RViz visualization is smooth. - Both costmaps use `robot_radius` or `footprint` polygons. For a rectangular footprint 0.5 × 0.4 m, we can list footprint vertices (e.g. `footprint: [[0.25, 0.2], [0.25,-0.2], [-0.25,-0.2], [-0.25,0.2]]`). We enable `footprint_clearing_enabled: true` in voxel layer so the robot's own footprint is cleared of obstacles (important if LiDAR sees parts of the robot) [37]. - **Obstacle Layer** (or Voxel Layer) config: - `observation_sources: lidar` (single source). - `lidar.sensor_type: PointCloud2` (since we feed point cloud). - `lidar.topic: /points` (assuming we remapped the Livox output to `/points`). - `lidar.data_type: PointCloud2` and if voxel layer, we specify `lidar.obstacle_range` (e.g. 10 m) and `lidar.raytrace_range` (maybe 15 m) so that beyond obstacle_range points are ignored, and raytrace_range is how far to clear. - `lidar.min_obstacle_height: 0.1` m, `lidar.max_obstacle_height: 1.0` m (tuning to ignore ground and low obstacles above 1m since the robot is <0.3 m tall). - `voxel_layer.z_voxels: 16` (to split the 0–1 m height into bins, though for 2D usage this might not be critical). - `inflation_layer.inflation_radius: 0.25` m, `cost_scaling_factor: 3.0` (this means cost decays exponentially – a higher factor = steeper cost increase near obstacles, encouraging more clearance). - We also set `inflation_layer.hysteresis: false` (no inflation hysteresis needed) and could set `footprint_clearing_enabled: true` on static layer (to allow clearing unknown space inside footprint, though static layer being just the map means that's usually not needed). - **Planner Server:** - `planner_plugin: "SmacPlannerHybrid"` (the plugin library is `nav2_smac_planner/SmacPlannerHybrid`). We configure `allow_reversing: false` (to keep the robot from planning backward motions, unless we want it – we can allow reversing if the rover can drive backward as easily as forward, which it can). - `minimum_turning_radius: 0.5` m (example value, tune based on wheelbase – the Rover can turn in place, effectively zero radius if it rotates wheels, but we may put a small value to encourage smoother arcs). - `motion_model: "ackermann"` (Smac Hybrid supports Dubin, ReedShepp, Ackermann models – ackermann allows in-place rotation as separate actions). - We reduce `planner_frequency: 1.0` Hz (global replan once per second is enough). - **Controller Server:** - `controller_plugin: "FollowPath"` (assuming RPP is set as default plugin in nav2_params, sometimes it's called `FollowPath` or specifically `nav2_regulated_pure_pursuit_controller/RegulatedPurePursuit`). - Set RPP parameters: `desired_linear_vel: 0.4` m/s (max cruise speed), `lookahead_dist: 0.6` m, `min_lookahead_dist: 0.3` m, `max_lookahead_dist: 1.0` m, `lookahead_time: 1.5` s. These define how far ahead on the path the robot aims. We also set `rotate_to_heading_angular_vel: 0.3` rad/s (speed for in-place rotate if needed). - `use_velocity_scaled_lookahead_dist: true` (so when moving slow, lookahead is shorter, which helps precision). - `transform_tolerance: 0.2` s (tolerate small TF delays). - **Goal checker:** use `goal_checker: "goal_checker/XYTolerances"` with `xy_goal_tolerance: 0.05` m and `yaw_goal_tolerance: 0.1` rad (about 5.7°) to satisfy "close enough" stopping. We enable `stateful: true` if using RPP so that once the robot is within tolerance, it stops trying to correct further [38]. - We disable DWB critics since we're not using DWB. - **Behavior Tree & Recovery:** We use the default `navigate_w_replanning_and_recovery.xml` BT. We set parameters like `failure_tolerance` if needed (e.g. allow 3 consecutive controller failures before considering the mission failed). We also configure the **Spin** recovery to use a small angular speed (since the LiDAR is 360°, spinning in place is mostly to find new paths rather than to see obstacles, but still we include it). - **AMCL (if in localization mode):** - `min_particles: 500, max_particles: 2000` (for a 10×10 m area, this is plenty). - `initial_pose` – we either set via RViz or have it in launch (0,0 if starting near origin). - `laser_model_type:`

`likelihood_field`, `update_min_d: 0.2` m, `update_min_a: 0.1` rad, etc. (Tune so AMCL updates frequently as the robot moves, given slow speed we can allow small movements to trigger updates). - AMCL's `z_hit, z_short, z_max, z_rand` and `sigma_hit` we use defaults typically, as those are sensor model params. - **SLAM Toolbox (if in mapping mode):** - Use `sync_slam: true` (sync mode ensures the map updates in real-time). - `max_laser_range: 15.0` m, `resolution: 0.05` m, `mode: mapping` (if want to continue to add to map). - We might set `loop_closure_interval: 30` s and `loop_closure_threshold: 0.8` to enable loop closures if the robot revisits an area. - Once map is built, we'll save it (with `map_saver_cli`) to a file in `maps/` directory.

Below is a **condensed nav2_params.yaml** highlighting important parts:

```
amcl:
  ros__parameters:
    use_sim_time: false
    motion_model_type: "diff"  # Rover is actually omni, but diff is fine for
slow
    min_particles: 500
    max_particles: 2000
    xy_sigma: 0.2
    update_min_d: 0.2
    update_min_a: 0.1
    laser_min_range: 0.1
    laser_max_range: 15.0
    initial_pose: [0.0, 0.0, 0.0]  # to be set from launch if needed
global_costmap:
  ros__parameters:
    update_frequency: 1.0
    publish_frequency: 1.0
    global_frame: "map"
    robot_base_frame: "base_link"
    rolling_window: false
    width: 15.0
    height: 15.0
    resolution: 0.05
    footprint: [[0.25,0.2],[0.25,-0.2],[-0.25,-0.2],[-0.25,0.2]]
    plugins: ["static_layer", "inflation_layer"]  # no obstacle layer here if
using static map only
    static_layer:
      map_subscribe_transient_local: true
      subscribe_to_updates: true
    inflation_layer:
      inflation_radius: 0.25
      cost_scaling_factor: 3.0
local_costmap:
  ros__parameters:
    update_frequency: 5.0
```

```
    publish_frequency: 2.0
    global_frame: "odom"
    robot_base_frame: "base_link"
    rolling_window: true
    width: 3.0
    height: 3.0
    resolution: 0.05
    footprint: "polygon"
    footprint_polygon: [[0.25,0.2],[0.25,-0.2],[-0.25,-0.2],[-0.25,0.2]]
    footprint_clearing_enabled: true
    plugins: ["voxel_layer", "inflation_layer"]
    voxel_layer:
      enabled: true
      voxel_decay: 0  # use static clearing
      origin_z: 0.0
      z_voxels: 10
      z_resolution: 0.1
      obstacle_range: 10.0
      raytrace_range: 12.0
      max_obstacle_height: 1.0
      min_obstacle_height: 0.1
      clearing: true
      marking: true
      observation_sources: "lidar"
      lidar:
        data_type: "PointCloud2"
        topic: "/points"
        marking: true
        clearing: true
        min_obstacle_height: 0.1
        max_obstacle_height: 1.0
    inflation_layer:
      inflation_radius: 0.25
      cost_scaling_factor: 3.0
TrajectoryPlanner:
  # not used (DWB), ensure controller_server uses pure_pursuit instead.
ControllerServer:
  ros__parameters:
    use_sim_time: false
    controller_plugins: ["FollowPath"]
    FollowPath:
      plugin: "nav2_regulated_pure_pursuit_controller/
RegulatedPurePursuitController"
      desired_linear_vel: 0.4
      max_linear_accel: 0.5
      lookahead_dist: 0.6
      min_lookahead_dist: 0.3
      max_lookahead_dist: 1.0
```

```
      lookahead_time: 1.5
      rotate_to_heading_angular_vel: 0.3
      use_velocity_scaled_lookahead_dist: true
      transform_timeout: 0.2
      regulated_linear_scaling_min_radius: 1.0
      use_regulated_linear_velocity_scaling: true
      use_rotate_to_heading: true
      rotate_to_heading_min_angle: 1.57  # if path requires >90° change, rotate
in place first
GoalChecker:
  ros__parameters:
    goal_checker_plugins: ["goal_checker"]
    goal_checker:
      plugin: "nav2_controller/GoalChecker"
      xy_goal_tolerance: 0.08    # 8 cm tolerance
      yaw_goal_tolerance: 0.10   # ~5.7 degrees
      stateful: true             # once in goal, consider succeeded
PlannerServer:
  ros__parameters:
    planner_plugins: ["SmacPlanner"]
    SmacPlanner:
      plugin: "nav2_smac_planner/SmacPlannerHybrid"
      tolerance: 0.5
      minimum_turning_radius: 0.3
      motion_model_for_search: "Ackermann"
      allow_reversing: false
      # Optimize for smoother paths
      angle_quantization_bins: 72  # 5° resolution
      smoothing_navfn: true
      # costmap_downsampling: 1 (no downsampling for small map)
BehaviorTreeManager:
  ros__parameters:
    use_sim_time: false
    # Use default BT XML from nav2_bringup package, or our customized tree
    default_server_timeout: 20.0
    default_bt_xml_filename: "navigate_w_replanning_and_recovery.xml"
RecoveryServer:
  ros__parameters:
    recovery_plugins: ["spin", "backup"]
    spin:
      plugin: "nav2_recoveries/Spin"
      max_rotational_vel: 0.4
      tolerance: 0.03
    backup:
      plugin: "nav2_recoveries/BackUp"
```

Note: The above is a representative configuration. In practice, we would adjust values after testing (especially tolerances and acceleration limits to avoid jerky motion). We set `use_sim_time: false` for real robot, but in simulation we'll override that to true.

Waypoint Following Example (rover_waypoints package): We implement a Python node using the `nav2_simple_commander` API. This high-level API allows sending multiple waypoints easily [39]. The script might be called `waypoint_follower.py`:

```
#!/usr/bin/env python3
from nav2_simple_commander.robot_navigator import BasicNavigator,
NavigationResult
import rclpy
from geometry_msgs.msg import PoseStamped

def main():
    rclpy.init()
    navigator = BasicNavigator()

    # Wait for Nav2 stack to be active
    navigator.waitUntilNav2Active()

    # If map is known, you could set initial pose
    # initial_pose = PoseStamped()
    # initial_pose.pose.position.x = 0.0
    # initial_pose.pose.position.y = 0.0
    # initial_pose.pose.orientation.w = 1.0
    # navigator.setInitialPose(initial_pose)

    # Define a sequence of waypoints (could also load from a file or parameter)
    waypoints = []
    def make_pose(x, y, yaw=0.0):
        p = PoseStamped()
        p.header.frame_id = 'map'
        p.pose.position.x = x
        p.pose.position.y = y
        # Convert yaw to quaternion
        import math
        qz = math.sin(yaw/2.0)
        qw = math.cos(yaw/2.0)
        p.pose.orientation.z = qz
        p.pose.orientation.w = qw
        return p
    waypoints.append(make_pose(2.0, 0.0, 0.0))
    waypoints.append(make_pose(2.0, 2.0, 1.57))
    waypoints.append(make_pose(0.0, 2.0, 3.14))
```

```
    navigator.get_logger().info(f"Starting navigation through {len(waypoints)}
waypoints...")
    result = navigator.followWaypoints(waypoints)  # This will send a
NavigateThroughPoses request

    # Monitor progress
    while not navigator.isTaskComplete():
        feedback = navigator.getFeedback()
        if feedback and feedback.distance_remaining < 0.3:
            navigator.get_logger().info("Near waypoint, distance remaining: %.
2f" % feedback.distance_remaining)
        rclpy.spin_once(navigator)  # yield control

    # Check result
    if navigator.getResult() == NavigationResult.SUCCEEDED:
        navigator.get_logger().info("Waypoint following succeeded!")
    else:
        navigator.get_logger().error("Waypoint following failed: result code %s"
% str(navigator.getResult()))

    navigator.lifecycleShutdown()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

This script: - Initializes the Nav2 Simple Commander and waits for the Nav2 servers to be ready. - Defines a list of waypoints (in `map` frame) to visit. We can specify approximate orientations or use `NavigateThroughPoses` which doesn't require stopping at each intermediate orientation. - Calls `navigator.followWaypoints(waypoints)`, which uses the **NavigateThroughPoses** action under the hood to go through each pose in sequence. The Behavior Tree in Nav2 will treat intermediate poses as waypoints (the default BT has a `NavigateThroughPoses` node that goes to each pose one by one, or alternatively directly plans a single path through all if possible). - We spin and provide some feedback logs. At the end, we log success or failure.

This example will cause the robot to navigate a triangular route (0,0) -> (2,0) -> (2,2) -> (0,2) and back to start, just as a demo. In a real scenario, the waypoints could be loading docks, delivery points, etc., and one could integrate with a higher-level scheduler to trigger this node with different queues.

Workspace Build: All packages will have `package.xml` and `CMakeLists.txt` properly defined. `rover_description` installs the URDF and meshes. `rover_bringup` exports launch files. `rover_navigation` exports the param YAML and BT XML (if custom). `rover_waypoints` depends on `nav2_simple_commander` (which is part of `nav2_navfn_planner` or its own package as of Humble) – we add that in package.xml.

We then do:

```
colcon build --symlink-install
```

and source the workspace to use the launch files.

4. Simulation Setup (Gazebo)

Before testing on the real robot, we set up a Gazebo simulation to validate the navigation stack. We use **Gazebo Classic** (gazebo11, which is fully supported on ROS 2 Humble via `gazebo_ros` packages). Alternatively, Gazebo Fortress/Ignition could be used, but we'll use Classic for familiarity.

Gazebo World: We create a simple indoor world (e.g. a 10 m × 10 m room with some walls or obstacles). Gazebo offers building editor or we can place primitives: - A rectangular room with walls, leaving a doorway or open side. - Insert a few obstacle models: e.g. a table, chairs, or simply cubic pillars to represent static obstacles. - The floor is flat. We ensure the LiDAR's ray can hit the walls/obstacles (i.e. give obstacles some height like 0.5 m or more within LiDAR's vertical range).

We might create `house.world` or use an existing world (like TurtleBot3's apartment world, scaled to our needs). For simplicity, a world file snippet:

```
<world name="simple_room">
  <include file="sun.world"/>  <!-- lighting etc. -->
  <model name="floor">
    <static>true</static>
    <link name="floor_link">
      <collision name="floor_col"><geometry><plane><normal>0 0 1</
normal><size>20 20</size></plane></geometry></collision>
      <visual name="floor_vis"><geometry><plane><normal>0 0 1</normal><size>20
20</size></plane><material><script>Gazebo/StoneTiles</script></material></
visual>
    </link>
  </model>
  <!-- Walls -->
  <model name="wall1">
    <static>true</static>
    <link name="link"><collision name="col"><geometry><box><size>10 0.1 1</
size></box></geometry></collision>
      <visual name="vis"><geometry><box><size>10 0.1 1</size></
box><material><ambient>0.8 0.8 0.8 1</ambient></material></visual>
    </link>
    <pose>0 5 0.5 0 0 0</pose>
  </model>
  <!-- other walls ... -->
  <!-- Table obstacle -->
  <model name="table">
    <static>true</static>
```

```
    <link name="link">
      <collision name="col"><geometry><box><size>0.5 0.5 0.75</size></box></
geometry></collision>
      <visual name="vis"><geometry><box><size>0.5 0.5 0.75</size></
box><material><color rgba="0.7 0.4 0.3 1"/></material></visual>
    </link>
    <pose>2 2 0.375 0 0 0</pose>  <!-- a box table 75cm high -->
  </model>
  <!-- Robot -->
  <include>
    <uri>model://rover_x40a</uri>
    <name>rover</name>
    <pose>0 0 0.1 0 0 0</pose>
  </include>
</world>
```

We spawn the **robot model** in Gazebo either via `<include>` in world as above (if we saved it as a model), or by ROS spawn:

```
ros2 run gazebo_ros spawn_entity.py -entity rover -topic /robot_description -x
0 -y 0 -z 0.1
```

We can integrate spawn into a launch file for convenience.

Using Simulation Time: In simulation, we set `use_sim_time = true` for all nodes (Nav2, SLAM, etc.) so they use Gazebo's clock.

Launching simulation: We create a launch file `sim_launch.py`: - Launch Gazebo with our world: `ExecuteProcess(cmd=["gazebo", "--verbose", "<path_to_world>"])`. - Launch the robot_state_publisher with the URDF (to publish /tf for robot). - Use a Gazebo plugin or `spawn_entity.py` to insert the robot. - Launch Nav2 (we can reuse `rover_navigation.launch.py` with `use_sim_time:=true`). For simulation, we might start in SLAM mode to build the map from scratch.

Building a Map in Simulation: With SLAM Toolbox running and the robot in an unknown environment, we can drive the robot manually or use an exploration node: - One method: use `teleop_twist_keyboard` or an RViz "2D Nav Goal" to manually drive and cover the area. The LiDAR will map the walls/obstacles. We watch RViz to see the map forming. - Alternatively, use the Nav2 **Explore Lite** node or similar exploration package to autonomously cover the space. But manual teleop is fine for a small area. - Once we have a complete map (the occupancy grid displayed in RViz covers the environment and no large unknown gaps remain), we save it: `ros2 run nav2_map_server map_saver_cli -f ~/map_sim` which generates `map_sim.yaml` and `map_sim.pgm`. - We then can switch Nav2 to **localization mode**: kill SLAM, launch AMCL with the saved map. We provide the saved `map_sim.yaml` via `nav2.launch.py map:=<path_to_yaml>`. - Test localization: set initial pose in RViz near the robot's true pose, ensure `amcl_pose` converges. The robot should localize accurately (the map and odometry will align). - Now test **navigation**: use RViz "Nav2 Goal" tool or our waypoint script to send goals. The robot in Gazebo should plan

and move to them. We verify it avoids the table and walls. If we place a dynamic obstacle (e.g. a person model or simply stand in front via a virtual obstacle insertion), we see if the local planner stops or detours. We adjust parameters as needed until simulation runs smoothly (goal success rate high, no significant oscillations or collisions).

We specifically check: - The robot stops within ~0.1 m of goal (or within our tolerance). - It doesn't overshoot corners. - Costmaps in RViz show inflation clearly around obstacles and no "speckle" obstacles on the floor (if we see scattered noise where floor should be free, we tweak ground filter). - Timing: CPU usage is within limits (with ~720 beams, 16 layers LiDAR, costmap updates at 5 Hz, our PC should handle it; if not, reduce resolution or update rates).

Simulated Delivery Run: As an acceptance test in sim, we could programmatically send a series of delivery waypoints using the Python script. For example, simulate picking up item at (0,0), delivering to (2,2), returning to base. The path might involve going around the table. We measure that it succeeds consistently. We gather statistics like **path length** vs direct distance (to ensure not overly detouring) and any oscillation events.

With simulation successful, we proceed to the real robot deployment.

5. Real Robot Bring-up (Networking, Sync, Hardware Integration)

Bringing up the system on the actual Rover X40A involves additional considerations: - **Networking:** If using an external laptop to monitor or send commands while the robot's onboard PC runs the stack, we must configure ROS 2 networking (DDS). In ROS 2, by default, it uses multicast – if on the same Wi-Fi network, it should auto-discover. We ensure the laptop and robot share the same **ROS_DOMAIN_ID** if isolating them from other robots. If there are multiple network interfaces, sometimes FastDDS picks the wrong one; we might set environment variables to specify the interface. Time synchronization between devices is important if we have multiple compute nodes: - We run the **chrony** time sync service on the robot PC and any remote PC, with one as NTP server. Chrony can synchronize clocks over Wi-Fi within <1 ms difference, which is important for TF timestamps and message ordering if, say, RViz or remote nodes are running offboard. - Alternatively, if only one PC (on the robot) is running all navstack and we just use remote desktop or teleop, then sim time sync is not an issue. - **Udev Rules:** To reliably identify hardware devices: - The Livox MID-360 likely connects via Ethernet or USB. If Ethernet (using UDP packets), we ensure the correct network configuration (maybe a static IP). If USB (perhaps through a converter), we set a udev rule to give it a consistent name if it appears as `/dev/livox` or similar. - The Vstone motor controller might appear as a USB serial (`/dev/ttyACM0` or `/dev/ttyUSB0`). We add a rule matching its vendor:product ID to symlink e.g. `/dev/rover_base`. In the `vstone_base.launch.py`, we then refer to that device path. This prevents issues if Linux enumerates devices in a different order on reboot. - Example udev rule (based on VS-WRC board if known): `SUBSYSTEM=="tty", ATTRS{idProduct}=="***", ATTRS{idVendor}=="***", SYMLINK+="rover_base"`. - **Sensor Extrinsics Calibration:** We physically measure the LiDAR position on the Rover. For accurate mapping, the LiDAR's position and orientation relative to base_link must be correct. Even a few centimeters error can cause mapping inconsistencies (e.g., robot thinks LiDAR is more forward and thus walls appear shifted). We use a tape measure or CAD data from Vstone to set the X-Y offset of the `livox_link`. If available, we could do a quick calibration by observing the robot against a wall: drive straight along a wall and see in RViz if the wall in the map stays straight. If the LiDAR was offset, the wall might look curved or angled – then adjust the offset. Given the environment is flat, this should be straightforward. - Additionally, if the IMU is used, calibrate its orientation (the MID-360's IMU might need

alignment to LiDAR frame; the driver usually handles that). - **Vstone Communication Interface:** The user has the official repo, so presumably the firmware and ROS node are all set. We integrate by: - Running the base driver node (maybe something like `ros2 run fwdsrover_xna fwdsrover_xna_node` with appropriate params). The documentation or README (likely in Japanese) should list topics. From Qiita and context: - The node probably publishes `odom` as `nav_msgs/Odometry`. We'll echo that to verify. If it doesn't, but instead publishes a custom /rover_odo twist, we'll use a small node to convert it. Possibly the ROS2 package improved this by directly publishing Odometry with covariance (the ROS1 had a `pub_odom` node doing essentially: take delta distance from encoders, update x,y,theta, publish /odom and broadcast tf). - There might also be a topic for **battery or voltage** (the Qiita snippet shows a subscriber for "voltage" and such [40]). We should ensure those don't conflict with Nav2 (they likely don't). - The **emergency stop** on the robot – we need to see how it's exposed. Possibly the base driver might publish a topic or service when E-stop is pressed. Or it might just cut motor power. To be safe, we implement a software e-stop: a subscriber to (perhaps) a GPIO message or we configure that pressing E-stop triggers a script that calls `ros2 lifecycle set nav2_bt_navigator deactivate` (pausing Nav2). This can be looked at if needed. But since it's standard, likely pressing E-stop physically stops the robot's motors regardless of ROS commands (the motor controller likely ignores commands when E-stop circuit is open). - **Remapping in launch:** As mentioned, we remap Nav2's `/cmd_vel` -> `/rover_twist`. We can do this in the `nav2.launch.py` by setting an argument in `BringupLaunch` or adding a `Node` remapper. Alternatively, we keep Nav2 on `/cmd_vel` and use a ROS2 `topic_tools relay` to forward `/cmd_vel` to `/rover_twist` if we prefer not to touch the Nav2 config. But remap is simpler and was shown in Vstone's example [17]. - We also ensure the base driver's odometry `frame_id` is `odom` and child_frame `base_link` to be consistent. If not, we might remap those frame IDs via parameter or use the `robot_localization` EKF to subscribe to its data and republish correct frames. - **Bridging (if required):** In case the Vstone provided interface was only ROS1 (say they haven't finished ROS2 for some reason), we set up a ROS1 bridge: - Run `ros2 run ros1_bridge dynamic_bridge` on the robot PC (which also has ROS1 melodic/noetic environment). - The Rover's ROS1 node publishes on `/rover_odo` (Twist) and we need that in ROS2. The bridge will create a topic if message definitions match (geometry_msgs/Twist is common interface). It will also bridge `/rover_twist` commands from ROS2 to ROS1. - We then would use an **Odometry converter**: since Nav2 needs `nav_msgs/Odometry`, and bridging a Twist as Odometry is not automatic (different type), we'd run a small ROS2 node that subscribes to bridged `/rover_odo` (Twist) and integrates or converts it to `/odom` (Odometry). But since Qiita shows the ROS1 stack had a node doing that, perhaps bridging the final `/odom` (if it existed) is easier. We would check the ROS1 launch – they had a `pub_odom` node that published `/odom` in ROS1 [41]. If we include that in ROS1, we can bridge the Odometry directly. All this is only if ROS1 is involved. Given ROS2 package exists, likely unnecessary.

- **Calibration and Tuning on real robot:** Once the stack is up on the real rover, we perform some tests:
- **Stationary Localization:** Start the robot at a known spot, give it the saved map from simulation or a real map if available. Set initial pose, see if AMCL converges. Drive the robot around via teleop a bit to ensure odometry and localization remain consistent (the AMCL pose in RViz should follow the robot closely and loop closure jumps should be small if any).
- **Sensor Checks:** Ensure LiDAR data in RViz lines up with map features (walls etc.). If not, check extrinsics or time sync issues.
- **Basic Navigation:** Send a short goal (1 m forward). The robot should plan a straight line and move forward. Check that it drives straight and stops at the goal. If there's oscillation or overshoot, adjust controller parameters (maybe lower `Kp` if it were a PID, but RPP doesn't have a direct Kp; instead, check acceleration limits).

- **Obstacle Avoidance:** Place an obstacle (e.g. a cardboard box or have a person stand) in the path and send a goal past it. The robot should detect and either go around or stop if no path. If it tries to collide, increase inflation or check costmap observation settings. If it oscillates in front of obstacle (a known issue if goals are very close behind an obstacle), we might enable Nav2's **oscillation prevention**: e.g. DWB had `oscillation_reset_dist`, but for RPP, ensuring `use_rotate_to_heading` helps if it's stuck toggling orientations. The **Collision Monitor** if configured will preempt and stop it as well.
- **Multi-point Mission:** Finally, run the waypoint script for a multi-stop route similar to simulation. Time it, and verify all KPIs (see Section 6).

Time synchronization on the robot: If LiDAR and main PC have different clocks (some LiDARs time-stamp data internally), we might want to configure the driver to use system time for header stamps. The Livox driver likely offers an option to sync time (perhaps via PTP or just uses ROS time on receipt). We'll ensure the driver outputs stamped PointCloud2 that aligns with Nav2's clock (if not, AMCL or costmap could throw warnings about timestamp skew).

ROS 2 QoS Settings: We pay attention to QoS: - LiDAR data: we set the costmap's subscription QoS to "Sensor Data" (best-effort) to match the driver's best-effort publishing (Livox driver likely uses default reliable for point cloud; if high bandwidth, some choose best-effort). We can configure costmap's `observation_sources` QoS via parameters if needed to avoid drops. - Odometry: ensure Nav2's localizer (AMCL or the state estimation in Nav2) uses reliable and that the odometry publisher is reliable (usually nav_msgs/Odometry is default reliable). - For Wi-Fi remote control, we might compress topics or limit bandwidth. Running RViz remotely is fine for testing, but in production we likely run headless.

System Startup: We set the robot to launch everything on boot for autonomous operation: - Possibly use systemd to launch a script that sources the workspace and runs `ros2 launch rover_bringup rover_navigation.launch.py map:=/path/to/map.yaml`. - Ensure the robot's PC is set to **Ubuntu 22.04, ROS 2 Humble** with all dependencies installed (Nav2, Livox driver, Vstone package, etc.). - We include a script to **check hardware** on startup (e.g. confirm LiDAR is returning data, else alert). The Vstone base likely has some startup delay for the microcontroller, so maybe a short wait or retry mechanism for connecting to it (the ROS node probably handles that by retrying the serial connection). - Logging: We configure ros2 loggers such that important info (like "failed to plan" or "recovery triggered") are saved, to help later debugging.

Environmental Setup: The indoor area might have **lighting or reflective surfaces** that could affect LiDAR (Livox is less sensitive to lighting than vision sensors, but reflective floors or glass could cause weird readings). We test in the actual space and possibly adjust LiDAR angle if needed (sometimes tilting it up or down slightly can help avoid floor reflections directly back to sensor). Given MID-360 is multi-beam, we trust its coverage.

At this point, with all components launched by `rover_navigation.launch.py`, the robot should be fully operational: it listens for Nav2 goals or direct usage of the waypoint API, and it can be monitored in RViz.

6. Testing & Acceptance Criteria

To ensure the system meets requirements, we define Key Performance Indicators (KPIs) and a test plan:

KPIs: - **Navigation Success Rate:** The rover should reach its designated waypoint goals **at least 95% of the time** without human intervention. Goal failures might occur if the path is completely blocked; in semi-static environments, we expect nearly all goals to be reachable after perhaps a short delay or replan. We will measure the **goal reach rate** over, say, 20 autonomous runs. - **Accuracy:** The rover should stop within **10 cm** of the target position and within the allowed orientation tolerance (we set ~5–10° tolerance). We verify this by measuring the final offset (in RViz or physically with tape). Our goal checker is set to 8 cm, so the system will consider anything under that as success. We ensure this tolerance meets the "close enough is ok" criterion (e.g. if delivering an item, being within 10 cm of the table is fine). - **Repeatability:** If the rover performs a route multiple times, the variation in its path and stopping point should be small. We want lateral error < 10 cm each time at key waypoints. This indicates good localization consistency and controller tuning. - **Obstacle Response:** If a new obstacle (e.g. a person or box) is placed in the robot's path: - The robot should **detect and stop** at a safe distance (at least the inflation radius + a bit). We test this by walking in front of the robot; it should pause before contact, ideally >~0.3 m away due to inflation padding. - It should either navigate around the obstacle if space allows or wait and then continue once the obstacle is removed. We can time how quickly the robot resumes once an obstacle disappears (the costmap should clear within a second or two of the person moving, and then the planner continues). - **Speed & Time:** Although speed is slow by requirement, we check if any route can be completed within reasonable time. If a point-to-point distance is, say, 5 m, at 0.4 m/s it should take ~12.5 s plus some rotation time. If we see significant slowdowns or waiting (without obstacles), that indicates an issue. We ensure the **delivery frequency** (deliveries per hour) meets the expectation (with slow speed and presumably short distances, it might do e.g. 10–20 deliveries/hour, but this depends on scenario). - **Localization robustness:** The robot should not get lost. KPI: it maintains localization (AMCL covariance stays bounded) throughout runs. If we manually disturb it (pick it up and move it a bit), does it recover? We might not require that, but good to test re-localization (like use the "initial pose" re-set if it ever lost). - **System Uptime:** The stack should run for hours without crashes or significant memory leaks. We can do a soak test: let the robot idle for an hour and navigate periodically, monitor memory and CPU. Nav2 on Humble is fairly stable (no known major memory leaks in costmap or planner when properly configured). - **CPU and resource usage:** On the X40A's PC (likely an Intel or AMD embedded PC), CPU usage should be manageable. With one 16-beam LiDAR at 10 Hz and Nav2, we expect maybe 50% CPU on one core at peak. We measure CPU/RAM. If CPU > 80% sustained, we might lower update rates to ensure no overload when something extra happens (like map updates or remote monitoring). - **Emergency Behavior:** When the E-stop is pressed: - The robot must stop immediately (this is hardware-enforced, but we also ensure our software stops sending commands). We test by driving and hitting E-stop; it should halt within a very short distance (the mechanical brake is immediate). - After releasing E-stop, the system should be able to resume (perhaps requiring a re-enable command depending on the controller firmware). We outline a procedure: likely twist commands remain zero while E-stop is pressed. Once released, we might need to toggle a service or simply issue a new nav goal. We ensure no unpredictable behavior on re-enable. - **Regulatory safety distances:** If applicable, ensure the robot never exceeds safe speed around obstacles (this is covered by our conservative speed and inflation). But if there's an internal requirement like "stop within 0.5 m of any obstacle if moving at 0.4 m/s" – with our braking distance, 0.4 m/s can stop within ~0.2 m easily. We effectively have that via inflation and collision monitor.

Test Plan:

1. **Static Navigation Test:** Place known static obstacles (boxes, etc.) at various positions. Command the robot to navigate to waypoints that require path planning around them. Observe in RViz that global planner plans around obstacles (costmap shows inflated obstacle and path circumventing). The robot should follow the path without collision. Use a checklist: *Did the robot maintain >~0.1 m*

clearance from obstacle edges as it passed? (We can measure from lidar points how close it got – should be > robot radius).

2. **Dynamic Obstacle Test:** Have a person walk in front of the moving robot. The expected result: robot slows or stops, then continues when clear. We do multiple trials at different encounter distances and angles (head-on vs crossing). KPI: No contact, and minimal "confusion" (no spinning in circles or oscillating).
3. **Long Run Repetition:** In a clear environment, have the robot run a cycle of say 5 waypoints repeatedly (like a delivery loop) for 30 minutes. All goals should be reached. We record any failures or significant deviations. If a failure occurs (e.g. robot got stuck on something or goal not reached in time), investigate and adjust parameters (like increase patience time or add a recovery).
4. **Localization Recovery Test:** Manually push the robot a bit off course (or simulate a wheel slip) and see if AMCL corrects it (the map to odom TF jump). If not auto-correcting, consider adding a routine: e.g. if localization covariance grows, trigger a re-localization by a spin or a service call to AMCL to re-initialize near expected position (not usually needed unless environment is feature-sparse).
5. **Networking & Remote Control Test:** If applicable, test that we can send a goal from a remote UI (RQt or a custom app) and the robot receives it. Also test remote E-stop override if needed (for example, a ROS topic `/emergency_stop` that an operator can publish to and our collision monitor or a simple subscriber will then stop the robot by clearing costmaps or calling `activate(False)` on controller).
6. **Battery and Mission Duration:** Run the robot until low battery to ensure no issues like performance degrading. The Vstone base likely has a battery of 24 V, 288 Wh giving a few hours. We ensure our software handles a low battery gracefully (not directly a nav issue, but possibly the base might publish a warning; we could incorporate that to stop missions when battery low).

We document all test results. Particularly, ensure the **lateral error < 10 cm** requirement: we verify by placing a tape cross at the goal and filming the robot's stop position relative to it multiple times. If it overshoots or stops short more than 10 cm away consistently, we refine the controller parameters (e.g. increase `goal_dist_tol` a bit but then rely on goal checker tolerance to accept it, or improve the braking by adjusting `max_linear_accel` and `slowdown_radius` in RPP). Given slow speeds, stopping accuracy of a few centimeters is achievable.

We also test **fail-safes**: - If the localization were lost (e.g. we deliberately give wrong initial pose), does the robot abort rather than drive crazily? Nav2's BT usually requires localization quality good before proceeding. We can test this by not setting initial pose and sending a goal – Nav2 should fail to plan since `pose` is unknown. - If sensor data is lost (unplug LiDAR), the costmap would stop updating. The Collision Monitor or local planner might stop the robot due to no new data. We see that as acceptable fail-safe (robot stops moving if blind).

Acceptance Criteria Summary: The system is accepted when: - Robot can autonomously navigate to all requested waypoints on the map with required tolerance. - No collision or dangerous behavior observed in a suite of trials (including presence of people). - The system runs reliably for extended periods. - All relevant safety measures (inflation, E-stop, collision monitor) function as intended. We will have the stakeholders witness a demo run: e.g., "The rover starts at the base, goes to Office A, then to Office B delivering items, then returns to base, while we occasionally walk in front of it. It should complete the route and pause for pedestrians appropriately." If it does so, we consider it passed.

7. Troubleshooting & Common Pitfalls

Despite careful setup, issues can arise. Here are common pitfalls and how to address them:

- **TF Configuration Errors:** One frequent problem is TF frames not set correctly. For example, if the LiDAR frame is not linked to base_link or if `odom→base_link` is not published, the costmap or AMCL will complain about missing transforms or the robot may not move (Nav2 will wait for transforms). To fix: use `ros2 run tf2_tools view_frames` to visualize the TF tree. Ensure `map→odom` (from AMCL/SLAM) and `odom→base_link` (from rover odometry) are present and being updated. Check that all sensor data header frames exist in the TF tree. A mismatch (e.g. LiDAR data says frame "lidar_link" but URDF uses "livox_link") will cause no sensor input to costmap. We resolve such issues by consistent naming and adding static transform publishers as needed. Also, be mindful of TF **timing**: If odometry or AMCL is publishing transforms with a timestamp far in the future or past relative to current ROS time, Nav2 might freeze (it has a transform tolerance of 0.5 s by default). We ensure all system clocks are synced (use_sim_time for simulation, chrony for multiple hardware). If we see warnings like "Transform timeout" or "Could not transform map→base_link", likely the transform tolerance is exceeded – possibly odom not updating fast enough or clock sync off. Solution: increase `transform_timeout` in controller, and fix underlying time sync.
- **Odometry Drift or Jumps:** If the odometry reported by the base is inaccurate (e.g., wheel slippage or calibration error in wheel radius), the robot's perceived motion will be off. This can manifest as the robot thinking it moved less or more than reality – causing localization error until the LiDAR corrects it. To mitigate: if drift is rotational, incorporate the IMU yaw (so the robot knows it rotated even if wheels slipped). Use **robot_localization** EKF to fuse encoder and IMU. If linear distance is off, calibrate the wheel parameters on the controller (Vstone's firmware might have a parameter for wheel circumference or encoder ticks per meter – refer to their documentation). Over short indoor distances, AMCL will correct global pose, but for local control, odometry quality affects how well the controller does path tracking. We noticed that especially for path following, if odometry underestimates rotation, the robot might not turn enough. So calibration is important. We can drive a known 1 m distance and see if odom reports ~1 m, adjust scaling if not.
- **Costmap Issues (Ground Clutter & Footprint):** As anticipated, one of the first issues may be the LiDAR seeing the ground or parts of the robot and marking them as obstacles. Symptom: The local costmap shows obstacles all around the robot (like a circular wall) even when in an open area, preventing any movement ("Planner failed to find path" because it thinks it's stuck). This is due to **ground returns** or the LiDAR hitting the rover's own body. Fixes:
- Increase `min_obstacle_height` filter to ignore points near ground.
- Enable `footprint_clearing_enabled` [37] in costmap so that any obstacle within the robot footprint is automatically cleared – this often handles seeing its own body.
- Use a **negative obstacle filter** if needed – not likely needed indoors.
- Worst case, mount the LiDAR higher or angle it slightly upward so it doesn't catch the floor right at the robot's base. But since we have parameter filters, that usually suffices.
- **Planner Oscillation:** Sometimes the local planner can get into an oscillation where it turns left-right repeatedly without progressing (especially in narrow passages or when goal tolerance is too tight). In DWB, this is a known scenario addressed by an **oscillation distance parameter** (robot moves < X meters and keeps changing direction triggers recovery). In our RPP controller, oscillation is less common, but one could still see a slow wiggle if, say, the path requires fine adjustment and the tolerance is small. Our approach: we set a reasonable tolerance and also use Nav2's default Recovery behaviors:

- The BT includes an **Oscillation Check** that should detect if the robot is oscillating (no progress to goal for some time) and then trigger the Spin recovery. We ensure `oscillation_timeout` (if using BT parameter) is set, e.g., 10 s.
- We also allowed our goal checker tolerance to be non-zero so that it doesn't oscillate trying to get exactly on spot.
- If oscillation happens around obstacles, possibly increase the inflation radius or cost penalty so the controller doesn't try to squeeze through too tight gaps causing back-and-forth decisions.
- **"No valid trajectory" / Controller Server timeouts:** If we had used DWB, this error can occur if it cannot find a path through dynamic costs. With RPP, a similar failure would be "Path followed lost" or it might simply slow to zero if it deems path not feasible. If we see the robot stuck while it should move, check local costmap: maybe an obstacle is persistently in front (could be a false-positive obstacle). If false, adjust sensor filters. If real, the robot is doing the right thing. If the robot is stuck because global planner hasn't replanned around a new obstacle, ensure that the BT is using the *replanning* version (we chose navigate with replanning BT, which calls the planner periodically). Confirm `planner_frequency` > 0. If the global planner is too slow (not expected in our small map), it could lag. But Smac Hybrid at 1 Hz on a small grid is fine.
- **Recovery behaviors causing problems:** In some cases, the default Spin or Backup recovery can put the robot in weird positions (e.g. backing into a corner further). If we observe that, we can disable a particular recovery. For example, in a tight space, spinning in place could confuse localization (spinning quickly might blur LiDAR data). We might reduce spin speed. Or if backup is dangerous (no rear sensor), we could disable the BackUp recovery to avoid reversing into unseen obstacles. We rely mostly on the rover's 360 LiDAR to see all around, so backup is safe since LiDAR covers rear as well (so it won't backup into something unseen).
- **Latency or command overshoot:** If commands from Nav2 have delay in reaching motors (e.g., network latency if the controller board is separate computer), the robot might overshoot targets. We prefer the motor controller on the same PC or directly connected, so latency < ~50 ms. If using Wi-Fi teleop, that's slower – but our autonomous nav loop is on the robot itself, so that's fine. We double-check that the Vstone driver is running real-time enough. If, for instance, the rover's motion has a noticeable lag to commands, we might shorten the controller's control cycle to compensate. Nav2's controller publishes at e.g. 20 Hz by default. The Vstone driver likely consumes and applies those immediately. If not, maybe its internal control is at 50 Hz which is fine. We test by sending a velocity command and measuring delay (could log odom vs cmd_vel timestamps).
- **Livox Driver throughput:** Livox MID-360 can produce a high volume of points (~100k+ points per second). The ROS2 driver in default might output full clouds at high frequency. On a resource-constrained PC, this could bog down the costmap updates. If CPU usage is high and we see costmap update lag (e.g., local costmap in RViz updating slowly or not keeping up), we may throttle the point cloud:
- The Livox driver's config JSON might allow selecting a subset of lines or a lower publish rate. Or we could downsample in a separate node.
- Using a **pointcloud filter nodelet** (like voxel grid filter in a CloudNode) could help but in ROS2, maybe use `pointcloud_filters` pipeline.
- Another strategy: since our environment is mostly 2D, we could run a **scan simulator** on the pointcloud – e.g. take the minimum distance in each angular sector to create a fake 2D LaserScan at 10 Hz. This drastically reduces data size. The `pointcloud_to_laserscan` package can do this: we set it to take points within a Z-range (like 0.1–0.3 m height) and produce a 360° scan. This might lose some vertical obstacle info (like if an object's base is out of that slice but a high part is in – e.g., a

tabletop with no legs detected in slice might be invisible). But for safety, we want legs, etc., which are at lower height – so it's fine.

- If using full 3D data via voxel layer, we consider enabling **raytrace clearing** so that moving obstacles don't leave ghost obstacles. The voxel layer's default clearing will handle it as long as the sensor origin is inside costmap and raytrace_range is set.
- **Nav2 Config Errors:** Many times, a parameter name might be mistyped or placed under wrong namespace in YAML, leading to it not actually being applied. If behavior is not as expected, we use the dynamic reconfigure (ROS2 param introspection) or simply log the loaded params (Nav2 prints them on start). For example, if we accidentally wrote `Footprint` vs `footprint` key, it might ignore it and default to a circle. By verifying logs or using `ros2 param get` on the costmap node, we catch such errors.
- **Licensing issues or package version conflicts:** We ensure that all custom code we wrote (like URDF, small scripts) are under a permissive license or what the project requires. The chosen packages have compatible licenses (Nav2 is Apache 2.0, Vstone's ROS2 package is Apache 2.0 [42], Livox driver is Apache 2.0 [43], slam_toolbox is LGPL-2.1 [33] which is fine to use as a library node). If we ever modify SLAM Toolbox source, we'd have to comply with LGPL (provide modifications source, etc.), but we likely use it as-is.
- **Upgrading to new ROS2 versions:** The question asks for best practices as of Aug 2025. ROS 2 Humble is LTS until 2027, but newer ROS 2 (Iron, etc.) exist. We ensure our setup is forward-compatible:
- We avoid deprecated parameters. For instance, the `navigate_through_poses` action interface changed in late Nav2 versions (Humble to Iron). But using nav2_simple_commander abstracts that. Still, keep an eye on the output – it might warn about something if not up-to-date.
- If we needed to patch something (some users reported minor bugs in nav2 or livox driver on Iron if any), we would note them. For example, a community note says livox driver 1.2.4 doesn't work on Iron without a PR [36]. Since we're on Humble, we are fine, but if planning upgrade, use the patched version or ensure maintainers merge fixes.

By systematically addressing these issues, we aim for a smooth deployment. We also maintain a **runbook** for operators: - Steps to start/stop the system. - What to do if the robot gets stuck (e.g. joystick override or manual nudge). - How to reset localization if needed (place robot at a known reference and call `initialpose`). - Monitoring: e.g. use `rqt_robot_monitor` or a dashboard that indicates battery, localization health, etc. – not explicitly requested, but a good practice.

8. Package Versions & Licensing

To ensure compatibility and support, we lock down exact versions of all key packages that we used on ROS 2 Humble (as of August 2025):

- **ROS 2 Distribution:** Ubuntu 22.04 with ROS 2 Humble (ros_base + nav2 packages installed via apt). Nav2 in Humble is around version 1.1.10 (we verify by `apt show ros-humble-nav2-bringup`). Specifically:
- `nav2_core`, `nav2_costmap_2d`, `nav2_smac_planner`, `nav2_regulated_pure_pursuit` etc. – use the latest patch release in Humble. (Nav2 1.1.**X** where X is latest bugfix.) According to docs, Humble to Iron transition added improvements [44] but we stay on Humble for now to match Ubuntu 22.04.

- **License:** Nav2 is **Apache-2.0** licensed [45], meaning we can use and modify freely with attribution. All Nav2 plugins we used (Smac, RPP) are under that license.
- **SLAM Toolbox:** We use version 2.2.6 (or latest in Humble release). On Humble, `ros-humble-slam-toolbox` deb should be installed (commit corresponding to mid-2023 release). **License:** SLAM Toolbox is LGPL-2.1 [33]. We do not modify its code, just use it as a node, which is allowed in our Apache-licensed project (LGPL is compatible as a linked library).
- **Livox ROS2 Driver:** We will use **livox_ros_driver2** version 1.2.4 (as mentioned by users for mid360 on humble [46]). We compile it from source in our workspace (since it may not be on apt). We also include the dependent **Livox-SDK2** (likely it's fetched by the driver's build script or as a submodule). We confirm this version supports MID-360 and has no critical bugs in our use-case. If needed, we apply any community patches (for example, if Iron support patch needed, but we stick to Humble so fine). **License:** Livox driver is Apache-2.0 (per their Gitee and README) [43]. That means we can freely integrate it. Note their note: "not recommended for mass production" [47] – but that just means one might optimize it. If we find performance issues, we could optimize (maybe reduce copies or unnecessary features). But functionally it's fine.
- **Vstone 4WDS Rover XNA ROS2 package:** We use the official repository `vstoneofficial/fwdsrover_xna_ros2` at version 1.2.0 (just assuming based on ROS1 version history). We have to build it from source (or if Vstone provided a deb, install it). We ensure it's targeting Humble (most likely, as the announcement for ROS2 firmware was late 2023, likely built for Humble). We might need to adjust its launch or config for our purposes. If we modify anything in it, we must adhere to its license. **License:** The GitHub shows **Apache-2.0** [42] for this package, meaning we can modify and redistribute. We will keep any modifications minimal and possibly contribute back if relevant.
- **Robot Localization:** If we use it for EKF, `ros-humble-robot-localization` (version ~2.7.X). License: BSD 3-clause (very permissive).
- **pointcloud_to_laserscan:** `ros-humble-pointcloud-to-laserscan` if used. License: BSD. We ensure to install that if needed for filtering.
- **Nav2 Simple Commander:** It's part of Nav2 (nav2_common or nav2_simple_commander package). On Humble, ensure we have the latest (some early Humble versions didn't include it by default, but by 2025 it's there). If not, we can pip install or include `nav2_python`. License: Apache 2.0 following Nav2.
- **Gazebo and Simulation:** Gazebo 11 (gazebo_ros_pkgs 3.5.0 etc). The URDF and world files we wrote are our own (we license them under Apache-2.0 for consistency). Gazebo models might have their own licenses (the ones we used are basic shapes we defined, so no issue).
- **Our Code:** We will license our custom packages (description, bringup, etc.) under a permissive license (Apache-2.0 or BSD) to align with ROS community norms and to avoid license incompatibilities. This also allows integration with the Apache-licensed Vstone code easily. We include headers in our scripts/launch files accordingly.

Patch Requirements: If any of the packages above had known bugs on Humble requiring patches: - For example, if the Regulated Pure Pursuit controller had a bug in early Humble that was fixed later, we'd cherry-pick that fix or use the latest patch release. We'll monitor the Nav2 issue tracker: - There was a mention of RPP improvements up to Nav2 1.2 (Iron) [48]. On Humble, RPP works but maybe lacks some features. We don't strictly need the latest features (like stateful parameter we already set). - Livox driver potential patch: On Iron it needed one, on Humble it's presumably fine as per user feedback [9]. We test it thoroughly (especially that it doesn't crash or leak after continuous use). If a minor fix is needed (for example, some users mention a "bind error" [49] – possibly trying to bind network port twice), if we encounter that, we solve by adjusting the config (maybe ensure only one driver node instance or correct IP). - Vstone package patches: If their ROS2 package is new, it might have updates. E.g., Qiita mentioned version

1.2.0 added X120A support. If we have X40A, initial release presumably supports it fully. We'll ensure that the config file for X40A (maybe a YAML with wheelbase dimensions, etc.) is used. If not present, we might need to create it based on the ROS1 config differences. - We ensure no **incompatible dependencies**: e.g., if we compiled Livox driver, it uses Livox SDK which might link to older Boost or something – but since recommended for Humble, likely it's fine. We compile with C++14 (ROS2 default) and resolve any warnings.

License Checklist: - Apache-2.0 (Nav2, Vstone driver, Livox driver) – we can use and distribute. We will include attributions and not remove their license notices. If we modify those codebases, we keep them Apache and mention changes. - LGPL-2.1 (SLAM Toolbox) – we dynamically link or run as separate node, so no licensing conflict. We just need to credit it. If we did static link (not applicable), we'd need to provide object files for relinking. But we don't do that. - Our code we make Apache-2.0 to be consistent (or BSD). - We must also abide any other third-party licenses: - The Livox SDK might be under BSD or Apache (likely Apache since the ROS driver is). - Standard ROS messages, etc., are BSD/Apache. - We list the licenses in our documentation and ensure any proprietary code (none here, all open) is identified. Vstone's firmware on the board is closed but that's irrelevant to our ROS side.

We also maintain version control (maybe git) for our workspace, tagging the commit that goes into production. This way, if an issue arises, we know exactly which version of each component was used, making debugging easier.

9. Deployment Document, Launch & Final Checklist

Finally, we compile all the above into a single deployment document (which this essentially is), with clear sections (we have done), including shell commands, code blocks (provided), and a checklist for operators/engineers. We've structured this document in Markdown with titles and lists as requested, which should be user-friendly to follow.

One-Command Launch: The `rover_navigation.launch.py` is our one-command bringup. For example, to start in localization mode with a pre-built map:

```
ros2 launch rover_bringup rover_navigation.launch.py map:=$HOME/maps/
office.yaml use_slam:=false
```

This will: - Launch Vstone base driver (connecting to the microcontroller over USB), - Launch Livox LiDAR driver (connecting to sensor, publishing pointcloud), - Launch robot state publisher (broadcasting URDF frames), - Launch Nav2 (bringing up AMCL, costmaps, planners, controllers, BT navigator), - Optionally launch RViz and our waypoint follower if desired (we might keep waypoint node separate in real use to not start automatically). After this, the robot should be ready to receive navigation goals.

Checklist (before running): - Calibration parameters (wheel radius, LiDAR pose) set correctly. - All hardware connections secure (LiDAR connected, E-stop engaged/not engaged as needed, motors powered). - Environmental preparation: initial pose known or set. - Map file in place (if using localization). - For SLAM mode: ensure area is clear enough for mapping, or do mapping after hours if people movement would confuse initial map.

Checklist (after running): - Verify all nodes are up (`ros2 node list` should show nav2 nodes, driver nodes). - Check `/odom` is updating (echo to see if values change). - Check `/points` or `/scan` coming from LiDAR (maybe echo or view in RViz). - Check `/tf` frames – e.g., `ros2 run tf2_ros tf2_echo odom base_link` to see if it's being published. - In RViz, confirm localization: robot model aligns with map (if not, set initial pose). - Send a test goal nearby to verify movement before doing a full mission.

Final Note: We'll also include links to relevant official docs and repositories for reference: - Nav2 documentation for configuration [50] [10]. - Vstone's official repo [15] (for any who need to refer to it or updates). - Livox driver repo and notes [51] [52]. - ROS2 design articles if needed for understanding (like TF or costmap, already cited relevant bits).

By following this comprehensive design and using the cited best-practices from ROS 2 and Nav2, the Rover X40A should achieve reliable autonomous navigation for indoor delivery tasks on ROS 2 Humble.

Sources:

- Nav2 official docs (Humble): Configuration guides and rationale [2] [4] [5] [10].
- Vstone Rover X40A info: Press release confirming 4WDS omni-steering and payload [53].
- Vstone ROS interface: Topic definitions from community example (Qiita) [16] and ROS2 usage notes [17].
- Livox MID-360 usage on ROS2: user feedback confirming it on Humble [9].
- Livox ROS2 driver readme: compatibility and license [47] [43].
- License references: Nav2 (Apache-2.0) [45], slam_toolbox (LGPL-2.1) [33], Vstone ROS2 (Apache-2.0) [42].

---

1 Nav2 — Nav2 1.0.0 documentation
https://docs.nav2.org/

2 3 10 11 19 50 Mapping and Localization — Nav2 1.0.0 documentation
https://docs.nav2.org/setup_guides/sensors/mapping_localization.html

4 5 7 20 21 22 nav2_smac_planner — nav2_smac_planner: Humble 1.1.18 documentation
https://docs.ros.org/en/humble/p/nav2_smac_planner/

6 12 35 53 vstone.co.jp
https://www.vstone.co.jp/news/pdf/press220804.pdf

8 nav2_mppi_controller: Humble 1.1.18 documentation
https://docs.ros.org/en/ros2_packages/humble/api/nav2_mppi_controller/

9 31 36 46 49 Which 3D, ROS Compatible, rotating LiDAR to buy - Computer Vision / Perception - Open Robotics Discourse
https://discourse.openrobotics.org/t/which-3d-ros-compatible-rotating-lidar-to-buy/26370

13 Collision Monitor — Nav2 1.0.0 documentation
https://docs.nav2.org/configuration/packages/configuring-collision-monitor.html

14 nav2_collision_monitor: Humble 1.1.18 documentation
https://docs.ros.org/en/humble/p/nav2_collision_monitor/

15 17 42 vstoneofficial/fwdsrover_xna_ros2 - GitHub
https://github.com/vstoneofficial/fwdsrover_xna_ros2

16 28 30 41 ROS - メガローバー制御 #C++ - Qiita
https://qiita.com/takmot/items/d81eac6b26204dac16b5

18 ROS2使用nav2全教程原创 - CSDN博客
https://blog.csdn.net/mubibaiwhale/article/details/132289239

23 ROS2 Navigation2: Local Plan does not Avoid Obstacle
https://robotics.stackexchange.com/questions/109841/ros2-navigation2-local-plan-does-not-avoid-obstacle

24 25 26 37 ROS 2 Navigation Tuning Guide – Nav2
https://automaticaddison.com/ros-2-navigation-tuning-guide-nav2/

27 (STVL) Using an External Costmap Plugin - Nav2
https://docs.nav2.org/tutorials/docs/navigation2_with_stvl.html

29 34 4-Wheel Independent ROS-Compatible Cart Robot 4WDS Rover X120A ヴイストン | IPROS GMS
https://mono.ipros.com/en/product/detail/2000876368/

32 How to get a Nav2 planner to ignore goal pose orientation : r/ROS
https://www.reddit.com/r/ROS/comments/1ktgv56/how_to_get_a_nav2_planner_to_ignore_goal_pose/

33 LICENSE — slam_toolbox: Humble 2.6.10 documentation
https://docs.ros.org/en/humble/p/slam_toolbox/__LICENSE.html

38 48 (SLAM) Navigating While Mapping — Nav2 1.0.0 documentation
https://docs.nav2.org/tutorials/docs/navigation2_with_slam.html

39 Nav2 Simple Commander - Stretch Docs
https://docs.hello-robot.com/0.3/ros2/navigation_simple_commander/

40 ros_bridge.py - niscode/scripts · GitHub
https://github.com/niscode/scripts/blob/main/ros_bridge.py

43 livox_ros_driver2: Livox最新得ros驱动，支持mid-360
https://gitee.com/xlhou/livox_ros_driver2?skip_mobile=true

44 Humble to Iron — Nav2 1.0.0 documentation
https://docs.nav2.org/migration/Humble.html

45 ROS Package: navigation2
https://index.ros.org/p/navigation2/

47 51 52 GitHub - Livox-SDK/livox_ros_driver2: Livox device driver under Ros(Compatible with ros and ros2), support Lidar HAP and Mid-360.
https://github.com/Livox-SDK/livox_ros_driver2