Mobile Manipulator Intelligence Stack Design

1. Repo Strategy Decision

We recommend a **monorepo** approach for the new manipulation intelligence stack. All new modules (perception, policy, adapter, etc.) will live in a single Git repository as separate ROS 2 packages. This makes it easier for 1–3 developers to coordinate changes and ensures all parts remain compatible. It's common in ROS to have multiple packages in one repo[1], especially when they belong to one robotic system. By grouping components that "deal with the same topic, algorithm, interface with the same machine"[2] (in this case, the mobile manipulator), we simplify development and integration.

Trade-offs: A monorepo centralizes versioning and CI, but it means all modules share the same release cycle. In contrast, a multi-repo setup (each module in its own repository) could allow independent reuse or separate open-sourcing of components, but would introduce overhead in dependency management and integration testing for a small team. Given our team size and the tight coupling between the manipulation modules, the monorepo is the pragmatic choice. We will **treat the existing `ranger-garden-assistant` base repo as an external dependency**, interfacing only through ROS 2 messages/actions (no code-level integration). This clean separation means the base navigation + Piper driver stack can run on its own (perhaps launched on the robot as `nav_stack`), and our new repo (call it `manipulation_stack`) can be launched separately to provide the "brain" for manipulation. The interface boundary between repos will be the **ROS 2 API surface** – i.e. published/subscribed topics, services, and actions. This ROS interface contract decouples the two codebases, so updates to one won't require modifying the other as long as the agreed message/topic schema remains consistent. We will clearly define these interfaces (see sections 4 and 5) as the "bridge" between the navigation base and the manipulation intelligence.

By not vendoring the Piper SDK or base code, we avoid duplicating source and instead rely on the base repo to provide hardware drivers. Our new stack will use ROS 2 topics/actions to command the base/arm, meaning we adhere to a **loosely coupled, distributed architecture** consistent with ROS best practices. In summary, **Mono-repo (for new modules) + ROS API interface to external base** is our strategy, balancing ease of development with modularity.

2. Suggested New Repo Name

We brainstormed several candidate names that convey mobile manipulation intelligence and generality:

- **MobileManipulationCore** – Emphasizes that this is the core "brain" for mobile manipulation (final choice).
- **ManipulationMaestro** – Connotes orchestrating arm and base in harmony, like a maestro conducting motion.
- **DexNav** – Suggests dexterous manipulation combined with navigation capabilities.
- **UniManipAI** – Short for "Universal Manipulation AI," highlighting general-purpose manipulator intelligence.
- **Armada** – A creative blend of "arm" with the idea of a coordinated system (though traditionally meaning a fleet).

Final Recommendation: MobileManipulationCore. This name is concise and descriptive – it indicates a central core for mobile manipulation intelligence without tying it to a specific robot (no "Ranger" in the name). It will be clear to stakeholders that this repository contains the higher-level AI/control stack for a mobile manipulator. The term “Core” implies foundational control logic, and “MobileManipulation” conveys the combination of mobility and manipulation. This name is unique enough to avoid confusion with existing projects and general enough to be reused for other robots beyond the Ranger platform.

3. Complete Folder Structure

Below is the proposed directory tree for the new repository, organized as a ROS 2 workspace with multiple packages and support files:

```
MobileManipulationCore/              # Root of the repository (ROS 2 workspace)
├── README.md
├── LICENSE
├── .gitignore
├── .github/
│   └── workflows/
│       └── ci.yml                   # CI pipeline definition (build, lint, test)
├── src/                             # Colcon workspace source packages
│   ├── manipulation_perception/     # Package: perception and sensor processing
│   │   ├── package.xml
│   │   ├── CMakeLists.txt
│   │   └── src/... (e.g. camera or object detection nodes)
│   ├── manipulation_policy/         # Package: policy model client/inference
│   │   ├── package.xml
│   │   ├── setup.py                 # Python package (uses HF LeRobot/OpenVLA)
│   │   └── manipulation_policy/...
│   ├── manipulation_adapter/        # Package: adapter that combines arm+base commands
│   │   ├── package.xml
│   │   ├── CMakeLists.txt           # Likely C++ for real-time considerations
│   │   └── src/... (e.g. adapter_node.cpp)
│   ├── manipulation_msgs/           # Package: custom ROS2 interface definitions
│   │   ├── package.xml
│   │   ├── CMakeLists.txt
│   │   └── msg/ action/ srv/ ...    # e.g. definitions for Observation, Action, etc.
│   ├── manipulation_bringup/        # Package: launch files to start the stack
│   │   ├── package.xml
│   │   ├── CMakeLists.txt
│   │   └── launch/
│   │       ├── core_launch.py       # Launch all core nodes (for real robot)
│   │       └── sim_launch.py        # Launch for simulation/testing
│   └── (optional other packages as needed, e.g. manipulation_sim or utils)
├── scripts/
│   ├── install_piper.sh            # Script to install/update Piper ROS & SDK
│   ├── update_piper.sh             # (Optional) Script to pull specific Piper versions
│   └── setup_policy_server.sh      # (Optional) Script to deploy remote policy server
├── config/
│   ├── piper_versions.yaml         # Pin file: Piper ROS repo tag/commit, SDK version
│   ├── robot_params.yaml           # Robot-specific params (e.g. topic names, frame
IDs)
│   └── policy_params.yaml          # Configuration for policy model (endpoints, etc.)
├── docs/
│   ├── design.md                  # Detailed architecture design document
│   ├── usage.md                   # How to run, configure, and extend the stack
│   └── api_reference.md           # Documentation of ROS interfaces (topics/services)
├── sim/
```

```
│   ├── urdf/                       # (If needed) URDF/Xacro for robot or objects, if not
using external
│   ├── worlds/                     # Gazebo world files for testing scenarios
│   └── launch/
│       └── gazebo_launch.py        # Launch file to spawn robot in Gazebo for testing
├── deployment/
│   ├── docker/
│   │   ├── Dockerfile.jetson       # Dockerfile for running all components on Jetson
│   │   ├── Dockerfile.policy       # Dockerfile for remote policy server (GPU-enabled)
│   │   └── docker-compose.yml      # Compose file to coordinate Jetson + server
deployment
│   └── k8s/                        # (Optional/future) Kubernetes manifests if scaling
└── tests/
    ├── unit/                       # Unit tests for modules (e.g. adapter logic)
    └── integration/                # End-to-end test scripts (possibly using simulation)
```

Notes: We do **not include Piper source code** in this repository to avoid duplication, since the base repo (ranger-garden-assistant) or upstream AgileX repos provide it. Instead, under `scripts/` we supply an **`install_piper.sh`** that can fetch the required Piper ROS driver and SDK. For example, this script might clone the AgileX `piper_ros` repo (Humble branch) and install `piper_sdk` via pip at a known-good version. A `piper_versions.yaml` file pins the exact git commit or version (acting as a single source of truth for Piper dependencies) so we can achieve reproducible setups. This approach follows the instruction to "send command/script to install or update Piper" rather than copying sources.

We organize the core code into multiple ROS 2 packages for clarity and colcon build isolation. For instance, `manipulation_msgs` will hold custom message/action definitions (like `EEFAction`, `Observation` messages) shared across modules. The `manipulation_policy` package is a Python-centric package (using `ament_python`) that interfaces with the ML model (OpenVLA/LeRobot) – it might also include any ML model downloading or preprocessing logic. The adapter is likely performance-critical, so we envisage `manipulation_adapter` as a C++ package (using `rclcpp`) that can operate at high frequency to command the hardware safely. We also include a `manipulation_bringup` package to contain ROS 2 launch files, making it easy to bring up the whole system with one launch command (and separate launch configurations for simulation vs real robot).

The repository also includes support directories: `docs` for documentation (important for a production-grade repo), `deployment` for containerization and deployment scripts, and `tests` for continuous integration. The `ci.yml` GitHub Actions workflow will automate building and testing every pull request (more in section 7). We also provide a `sim/` folder or package to support simulation using Gazebo (or Isaac Sim if needed) – this is critical for safe testing of manipulation logic in a virtual environment before running on the real hardware.

This structure is designed to be **colcon-friendly** (the `src/` directory houses all packages). A developer can simply clone the repo, run `rosdep install`, and then `colcon build` to build all packages. The multiple packages approach also allows selective build/test of components and fosters clear separation of concerns (e.g. policy vs adapter). In summary, the folder layout supports development, testing, documentation, and deployment, aligning with best practices for a production-grade ROS 2 codebase.

4. Architecture Mapping

Overall Data Flow: *Sensors and state from the base → Perception & Observation processing → Policy inference (OpenVLA/LeRobot) → High-level action (EEF target) → Adapter mapping → Low-level commands* *to base and arm controllers*. This pipeline can be visualized as a unidirectional flow with feedback via ROS topics/TF:

- **Sensors & State (External Base):** The base stack provides incoming data: camera images, depth, LiDAR (if any), joint states, and TF transforms (robot pose, arm link positions, etc.). The new repo's perception module will subscribe to these. We treat the base and its sensors as the "eyes and proprioception" of the system.
- **Manipulation Perception:** In our `manipulation_perception` package, we handle any processing needed to turn raw sensor data into **observations** for the policy. This could include image pre-processing, aggregating multi-sensor info, or feeding images through an object detector or state estimator if required. For instance, if using a language-conditioned policy, we might also take a high-level task command (text) as input. The output of this stage is a structured **observation message** containing all necessary info for decision-making (e.g. latest image frame, detected object locations, current robot joint angles, etc.).
- **Policy (High-Level Intelligence):** The policy node (leveraging Hugging Face LeRobot/OpenVLA as an *upstream source of truth* model) consumes the observation and produces an **action output**. This policy is essentially a large pretrained model (like OpenVLA 7B[3]) capable of mapping vision and state to robot actions. In real-time operation, this likely means the policy receives an image (and maybe text instruction) and autoregressively outputs a token sequence that decodes to a continuous action (as OpenVLA does[4]). We will wrap this model inference in a ROS 2 node (`manipulation_policy` package) that takes in the observation message and outputs a preliminary action message. *Example:* The policy might output a target end-effector pose (or delta) in 3D space or high-level instruction like "grasp at (x,y,z)". This output is high-level and may not yet be feasible or safe without further processing.
- **Adapter (Bridging Policy to Control):** The adapter module is the critical bridge that **maps the policy's output to the full mobile-manipulator action space**. It takes the policy's suggested end-effector movement or arm action and expands or adjusts it to concrete commands for both the base and arm. This involves kinematic reasoning with TF and safety checks:
- If the end-effector target from the policy is immediately reachable by the arm in the current robot configuration, the adapter will calculate the required arm joint motion (inverse kinematics or joint deltas) and command the arm accordingly.
- If the target is **outside the arm's workspace** or would require an uncomfortable pose, the adapter can decide to move the mobile base to assist. For example, if the policy wants to grab an object that is 2m forward but the arm reach is 1m, the adapter can command the base to drive closer while coordinating the arm motion. In this sense, the adapter expands the action from just the arm to **base + arm + gripper** commands.
- The adapter also enforces **TF-aware constraints**: it uses the robot's transforms (base_link, arm links, map frame, etc.) to interpret the policy output correctly. If the policy's coordinate frame differs (e.g. policy might output an end-effector pose relative to the camera frame or base frame), the adapter will transform that into the appropriate reference frame (usually the base or map frame) using the TF tree. This ensures consistent interpretation of targets.
- **Safety** is layered into the adapter: we will include checks for joint limits, self-collisions, singularities, and perhaps environmental obstacles (if we have a map or use MoveIt). The adapter acts as a **filter and planner** – it should never blindly execute policy commands if they violate safety constraints. For instance, if the policy erroneously suggests an arm motion that would hyper-extend a joint or collide with the robot's base, the adapter will adjust or reject that command.

- The output of the adapter is a set of low-level commands: e.g. a geometry_msgs/Twist for base velocity or a NavigateToPose action for the base, a trajectory or joint command for the arm, and an open/close command for the gripper.

Observation Schema: We define a clear schema for the policy observation. This could be encapsulated in a custom ROS 2 message (e.g. `manipulation_msgs/Observation`) that includes: - One or more camera images (likely an RGB image, and possibly a depth image or point cloud). We might not embed the actual image data in a custom message (since sensor_msgs/Image already exists), but the concept is the policy gets the latest image frames. - Robot proprioceptive state: joint angles of the arm, gripper state, etc., and possibly the base pose or odometry. These could come from `/joint_states` and TF, but the observation message might include a snapshot of joint states for convenience. - Task context: if there is a natural language instruction or goal description, that would be included (e.g. a string for the current task or goal). - Environment context: we may include binary signals like whether the base is currently localized or any costmap info, though likely the policy doesn't need raw costmap. But "nav state" could include whether the base is currently moving or stopped, or distance to known goal if that matters. - TF frames: rather than raw frames, the observation might include the current transform of key frames (e.g. base_link in map frame, or camera frame in base_link) if needed for the model. However, most VLA models operate on images and text, not explicit coordinate frames, so the adapter might use TF more than the policy does. The observation could be as simple as an image + textual prompt (for VLA models) plus some structured numeric data for joint angles.

In practice, we might not bundle all of this into one ROS message for performance reasons (images are large). Instead, the policy node can subscribe to the image topic, the joint states topic, etc., and internally assemble its "observation." The **observation schema** thus serves as a conceptual contract: *the policy needs X, Y, Z data at each inference step*. We will document exactly what data is fed to the policy. For example: "The policy takes as input a 224x224 RGB image from the wrist camera, the 7 DoF arm joint positions, and the last user command in text." This ensures any future change in sensor setup or policy input is deliberate.

Action Schema: We design the policy's action output in a way that the adapter can consume easily. The action may include: - **EEF pose or velocity:** e.g. a desired end-effector pose (position + orientation) relative to a certain frame, or a velocity command for the end-effector. Alternatively, the policy might output joint angle deltas for each joint (some policies operate in joint space). We need to accommodate either. Likely we define a message that can hold an end-effector target pose (geometry_msgs/Pose) *and/or* a list of joint angle increments. - **Gripper command:** the desired gripper state, e.g. open or close, or a continuous value (like width or effort). If OpenVLA's output tokens include a "open_gripper" action, we'll translate that to a gripper command here. - **Base movement:** optionally, the policy might suggest base motion. Some generalist policies (e.g. say "move forward and then pick up object") could imply the base should reposition. If OpenVLA was trained on mobile manipulator data, it might output base motion tokens. If so, our action schema should carry either a desired base velocity (twist command) or a target pose for the base. We include this as optional fields. If the policy doesn't provide it, our adapter can decide base motion autonomously. - **Frame identifiers or context:** We include metadata like what frame the end-effector pose is expressed in (camera frame vs base frame) if not fixed. However, we might standardize that the policy always outputs EEF targets in the robot's base_link frame (after training it in simulation accordingly). In any case, the adapter will confirm/transform frames via TF.

We will likely implement this schema as a ROS 2 **action** or message. One idea is to define a custom **Action** (ROS 2 Action type) called `MobileManipulationAction`, which has a Goal containing an Observation (so the policy can be invoked as an action server taking observation as goal and returning result with action commands). However, for continuous control, it might be simpler to use a topic stream. For now, we assume the policy node continuously publishes an `Action` message (or a equivalent custom message) at some frequency (e.g. 5–10 Hz) representing the latest intended action for the robot.

Adapter API: The adapter will subscribe to the policy's action output (topic or action result) and then perform the mapping logic. We can think of the adapter as providing a function or service: `execute_manipulation(policy_action) -> real_robot_motion`. It doesn't expose a formal service in ROS, but internally its "API" is: input = policy action, output = commands to hardware. The adapter will use several internal components: - **Kinematics solver:** to convert end-effector poses to joint angles (if not using something like MoveIt's planning pipeline, we might integrate MoveIt for inverse kinematics and trajectory planning). OpenVLA might output a sequence of waypoints (since it's autoregressive, maybe each token corresponds to a short motion). The adapter could use a time parameter (like assume each action is to be executed over the next 1/Nth of a second) or simply treat each output as an instantaneous target and feed it to a low-level controller (like a joint velocity controller). - **Motion constraints:** e.g. maximum joint speeds, acceleration limits, base speed limits. The adapter must ensure commands stay within these. It can incorporate a **rate limiter** or smooth planner for both arm and base motions. For example, if the policy suddenly requests a large jump in end-effector position, the adapter could break it into a trajectory that respects velocity limits. - **Safety checks:** as discussed, using environment info. If we integrate with MoveIt or have a known 3D model of the robot, the adapter can do collision checking for the planned motion. Initially, we might do simpler checks (like not exceeding known joint limits and ensure base doesn't move if arm is extended in a potentially unstable way). Over time, this can be refined. - **Frame management:** The adapter will heavily rely on TF. For example, if the policy gave an EEF target relative to the camera frame (perhaps the policy "sees" an object in the image and outputs a vector to it), the adapter will convert that point into the base_link frame to decide how to move the arm. We assume all necessary TF frames (map→odom→base_link→arm_link_N, camera frames, etc.) are available from the base's URDF and state publisher. The adapter must use the correct timestamped transforms to avoid errors in fast motion.

Finally, the adapter sends out commands to the actual controllers: e.g. a ROS 2 action to the arm's FollowJointTrajectory controller (or Piper's SDK commands), and velocity commands or a navigation goal to the base. This closes the loop: the base and arm execute the commands, changing the robot's state, which will be sensed and fed into the next observation.

To summarize, the architecture ensures a **modular flow**: - The **policy module** is kept as close to the upstream model as possible (we treat OpenVLA/LeRobot as a black box that outputs ideal actions in an unconstrained space). - The **adapter module** introduces robotics-specific intelligence (kinematics, dynamics, safety) to map those ideal actions into feasible motor commands. This separation means we can upgrade the policy model or even swap it (e.g. use a different ML policy) without re-writing low-level control logic, and vice versa. - The interface between modules is well-defined by the observation/action schemas, which also aids in testing (we can record an observation and see what action the policy suggests, and test how the adapter handles it).

Integration of RoboNeuron: We note that future integration with a cognitive framework like RoboNeuron is possible. RoboNeuron proposes a modular decoupling of sensing, reasoning (LLMs/VLAs), and control via a Model Context Protocol[5]. Our design aligns well with this concept: the policy node could be seen as a "reasoning" module (VLA model) and the adapter as the "control" module. We should design the adapter's interface such that an orchestrator (like an LLM in RoboNeuron) could call high-level functions. For example, RoboNeuron's MCP might allow an LLM to request "move arm to X" or "navigate to Y" – those would correspond to sending goals or enabling/disabling our policy as needed. By keeping the sensing and actuation separated and using ROS unified interfaces[6], we can later expose certain services (like a service to execute a pick-place task) that RoboNeuron could invoke. In essence, our architecture can serve as the lower layers (perception and control) that a higher-level cognitive agent (LLM) orchestrates. Ensuring our ROS messages are well-defined and semantically meaningful will make integration with such frameworks easier down the road.

5. ROS2 Integration Contract

To interface **MobileManipulationCore** with the existing Ranger base (navigation + Piper arm drivers), we define a clear set of ROS 2 topics, services, and actions. These form the **API boundary** between the two repos. Below is the list of key interfaces, along with their message types and intended usage:

Incoming data (from ranger-garden-assistant to our stack): - **`/camera/color/image_raw`** (`sensor_msgs/msg/Image`): Main RGB camera feed from the robot. Our perception node will subscribe to this to obtain visual input for the policy. - **`/camera/depth/image_raw`** (`sensor_msgs/msg/Image` or `sensor_msgs/msg/PointCloud2`): Depth information aligned with the RGB image. If available (e.g. from an RGB-D camera), this can be used for 3D understanding (optional for policy). - **`/joint_states`** (`sensor_msgs/msg/JointState`): Robot joint states, including at least all arm joints (and possibly base wheel joints if configured). Piper's driver likely publishes the arm's joint angles on this topic (possibly with a specific name for the arm joints). The policy/adapter can use this for proprioceptive input and to seed kinematics calculations. - **`/tf` and `/tf_static`** (transforms via TF2): The transformation tree of the robot. We expect frames such as `map`, `odom`, `base_link`, each arm link (`piper_link_1` … `piper_link_6` for a 6-DoF arm, for example), the `gripper_link`, and camera frames (e.g. `camera_link` or `camera_color_optical_frame`). Our system will use TF to translate between frames (no new topics, but it's part of the interface contract that the base provides a correct URDF and TF stream). - **`/odom`** (`nav_msgs/msg/Odometry`): Robot base odometry. If needed, the adapter can subscribe to odometry to know base linear/angular velocities or pose drift. Alternatively, the base_pose can be taken from TF (odom → base_link transform). This is mainly for awareness; the policy might not use odom directly, but the adapter might use it for planning base motions. - **`/nav2d_amcl_pose`** or **`/pose`** (geometry_msgs/PoseStamped): If the base uses localization (AMCL or SLAM), it might publish the robot's pose in map frame. This could be useful if the policy or adapter needs to know global position (for example, to decide that an object is out of reach and we need to navigate to a new spot). - **`<additional sensors>`:** Any other sensor topics provided by base – e.g. `/scan` for a laser scanner (sensor_msgs/LaserScan) if needed for collision avoidance. Our stack could subscribe if we implement obstacle-aware manipulation (not in initial scope but noted for future). - **Task command input** (if applicable): Possibly a topic or service where high-level commands come in (e.g. a user says "pick up the red ball"). This could be a text command topic or action, for example **`/task_command`** (`std_msgs/String` or a custom Goal message). This would originate from a user interface or higher-level orchestrator, not from the base. We'll assume for now that the task context is provided somehow to the policy (maybe just via a parameter or a hypothetical voice command node). It's mentioned for completeness.

Outgoing commands (from our stack to the base hardware controllers): - **`/cmd_vel`** (`geometry_msgs/msg/Twist`): Velocity command for the mobile base. The adapter will publish here to drive the base for small repositioning or continuous motion. We will use this for low-level base movements (like adjusting a few centimeters). The base's navigation stack (Nav2) or motor driver listens to `/cmd_vel`. - **`/navigate_to_pose`** (ROS 2 Action, `nav2_msgs/action/NavigateToPose`): If larger base movements are needed (e.g. moving across the garden to a work location), the adapter can call upon Nav2's NavigateToPose action rather than manual cmd_vel control. We'll integrate by creating a Nav2 action client in the adapter that sends a goal with a Pose (target location) when needed. The namespace for this is typically global (or under `/nav2/` if the base launch has one). We will confirm how `ranger-garden-assistant` exposes Nav2 – assuming it provides the standard Nav2 action server on `/navigate_to_pose`. - **Arm control interface:** This depends on Piper's ROS interface: - If Piper's ROS2 driver provides a FollowJointTrajectory action server (common for arms, likely on something like **`/piper_arm_controller/follow_joint_trajectory`** with type `control_msgs/action/FollowJointTrajectory`), our adapter will send trajectory goals to it. We should verify Piper's ROS2 interface; if not available, we may directly publish joint targets. - Piper SDK might also allow direct commands. The ROS1 example showed a node `piper_ctrl_single_node.py` and remapped `joint_ctrl_single` to `/joint_states`[7], which implies that setting joint targets might be done by publishing on a topic that node subscribes to. Possibly something like **`/piper_arm_controller/command`** with a `std_msgs/Float64MultiArray` of joint positions or velocities. We will need to consult Piper ROS2 docs, but for design we assume a **joint trajectory interface** exists. We plan to use standard ROS control approaches (FollowJointTrajectory) if available because it's well-supported (e.g. MoveIt can plug into it). - **Gripper control:** If the Piper arm has a gripper (it sounds like it does, as `girpper_exist` param in launch was true[8]), controlling it might be via a dedicated topic or part of the arm controller. Sometimes the gripper is treated as an extra joint. In Piper's ROS1, joint7 might correspond to the gripper opening (with range 0 to 0.08 m)[8]. We saw mention that in RViz the joint7 was used for gripper. So likely in ROS2, the gripper is just the last joint in the trajectory. Alternatively, there could be a custom topic like **`/gripper_command`** (`std_msgs/Float64` for width). We will clarify when integrating, but we will expose a command either by publishing to a topic or calling a service if one exists (e.g. some robot arms have open/close services). - **`/enable_arm`** (std_srvs/SetBool or custom service): Piper might require an enable signal (the ROS1 instructions talk about CAN enable). If the base repo doesn't handle it automatically, our launch could call a service to enable power to arm. Not strictly a command, but part of integration (ensuring arm is operational when needed). - **Feedback topics**: Our system may publish its own internal info for monitoring: - **`/manipulation_status`** (`std_msgs/String` or a custom status msg): Could report what the manipulation subsystem is doing (e.g. "approaching object", "waiting for policy output"). This is mostly for debugging and user info. - **`/tf` (additional frames)**: If our adapter creates any virtual frames (for example, a frame for a grasp point or a frame attached to a detected object), it might broadcast those on TF for transparency. However, primary TF comes from base, we likely won't add frames unless needed for visualization.

Namespace Conventions: We will likely run our stack in its own ROS namespace, e.g. `manipulator/` or `mm_core/`, to avoid collisions and clearly delineate components. For instance, our policy node could publish `manipulator/policy_action`, and adapter could subscribe to that. However, when interfacing with the base, we will use the base's expected namespace: - If the base is running in the root namespace (e.g. `/cmd_vel` global), we publish there. - If the base repo used a namespace like `/ranger/` for everything, we would follow that (e.g. publish to `/ranger/cmd_vel`). We'll confirm how the base is structured; many Nav2 setups keep topics global. For clarity, we might not namespace the base interface topics, but we will namespace our internal topics to avoid clutter. E.g., `MobileManipulationCore` nodes could have a namespace `mm_core` and within it, the policy publishes `~/policy_action` that the adapter in the same namespace consumes. But externally, the adapter uses absolute topics for base.

TF Frames Expectations: We expect the following frames to be present and static/dynamic transforms published: - `map` – global frame for Nav2 (if using localization). - `odom` – the odometry frame (drifts relative to map if not reset, etc.). - `base_link` – robot base reference frame (attached to the robot chassis). - `piper_link_<N>` – frames for each arm link and `piper_gripper` or `piper_ee` for the end-effector. These should be provided by the Piper driver (possibly via a joint state publisher or integrated in URDF). - `camera_link` and/or optical frames – if a camera is mounted, its pose relative to base_link should be in TF (either static if fixed, or dynamic if on a pan-tilt). - Possibly `odom->base_footprint` or similar conventions if any (depending on how Ranger base is represented). - Our adapter will rely on `base_link -> gripper_link` transform (via current joint states) to do inverse kinematics checks, etc. We assume the Piper's URDF is loaded (maybe the base URDF already includes it, or we might have to include Piper URDF in our launch if base doesn't). - For consistency, all position commands we compute for the arm will be in the **arm's base frame** (which is coincident with `base_link` if the arm is mounted rigidly on the robot). We need to verify if the Piper URDF mount is aligned such that `piper_link_0` is at the base. We should also ensure the coordinate conventions (e.g. x-axis forward, etc.) match between our internal calculations and the controller.

By adhering to these topics and frames, any other mobile manipulator with a similar setup could be integrated by just remapping topics and updating config. For example, another robot might have `/arm_controller/follow_joint_trajectory` instead of Piper's topic – our architecture can accommodate that via configuration.

In summary, the ROS 2 API surface between the repos is defined by: - **Sensor topics** we subscribe to (image, depth, joint_states, etc.). - **Control topics/actions** we publish or call (cmd_vel, NavigateToPose, joint control). - **TF frames** used for coordinate transformations. - We will document these in `docs/api_reference.md` and ensure the base team agrees on these interfaces. This contract means our system can command the base safely *through ROS messages only*, with no direct function calls – a clean separation.

6. Deployment Profiles

We plan for two deployment configurations:

A. Jetson-Only Deployment (Single-Machine): In this profile, the entire MobileManipulationCore stack runs on the Jetson AGX Orin (Ubuntu 22.04, ROS 2 Humble) on the robot. This includes perception nodes, the policy model inference, and the adapter/control nodes. The base's navigation stack is also on the same Jetson (as is typical for a single onboard computer setup). Everything runs in one ROS 2 domain for simplicity.

- *Process layout:* We will have multiple ROS 2 nodes, which can be split across processes as needed. For example, the heavy policy node (which might use a lot of Python and GPU) can run in its own process, while real-time critical nodes like the adapter could be separate and potentially run with higher scheduler priority. We might use ROS 2 composition for efficiency if appropriate, but likely separate processes for isolation (especially since the policy will use GPU and Python).
- *Performance considerations:* The Jetson AGX Orin 64GB is powerful, but a 7B parameter model like OpenVLA is borderline for real-time on it. We will attempt to optimize for on-device use. Techniques include converting the model to an ONNX/TensorRT engine with FP16 or INT8 optimization, which can significantly speed up inference on Jetson GPUs (reports suggest converting OpenVLA to TensorRT on Orin can nearly double throughput). If the model is still too slow to reach ~20–24 Hz control frequency, the Jetson-only profile may operate at a reduced loop rate (e.g. 5–10 Hz) and/or use a smaller model variant. We could explore NanoVLA or "small VLA" models[9][10] that achieve faster inference by sacrificing some generality. This profile is simpler (no network dependency), and is suitable for scenarios where network latency is a concern or a remote server is unavailable. However, the trade-off is potentially lower policy performance due to compute limits.
- *Launch & runtime:* We will provide a launch file (e.g. `core_launch.py` in `manipulation_bringup`) to start all components on the Jetson. This will also include steps like ensuring Piper drivers are running (the base might have launched them already; if not, our launch could include an `<include>` of Piper's launch or a system command to run `install_piper.sh` beforehand if

needed). In Jetson-only mode, no special networking configuration is needed beyond what ROS 2 does by default on loopback.

B. Split Deployment – Jetson + Remote Server (Two-Machine): This profile offloads the heavy policy inference to a remote GPU server (e.g. a desktop with an RTX card or a cloud instance), while keeping all real-time ROS 2 nodes on the Jetson. The motivation is to leverage more powerful GPUs for the VLA model (for speed or to run an even larger model) and mitigate the Jetson's load[11]. The Jetson will handle sensor publishing, running the perception and adapter nodes, and low-level control, whereas the remote server will handle running the policy model (OpenVLA or similar).

- *Architecture:* We adopt a **"LeRobot-style" remote inference pattern**[12]. The Jetson will act as a client that sends the latest observation (image and state) to the server and receives back an action. There are a couple of ways to implement this:
- **ROS 2 over network:** Run a ROS 2 node on the server that hosts the policy (perhaps the same `manipulation_policy` node, just launched on the server). The Jetson's perception node could publish images over the ROS 2 network and the server's policy node subscribes, then publishes action results that the Jetson's adapter subscribes to. This essentially extends the ROS graph across two machines. If we choose this, we'll configure DDS to operate in a multi-machine environment. ROS 2 DDS by default can work over LAN; we'd ensure the server and Jetson are on the same DDS domain (same ROS_DOMAIN_ID) and have network connectivity. We might need to tweak DDS settings (e.g. use unicast or configure initial peers) because mDNS discovery might not cross subnets. On a local network, setting the environment variable `ROS_DOMAIN_ID` the same on both might suffice. We would likely set up a **VPN or secure network** if the connection is over the internet, since DDS is not firewall-friendly by default and typically requires all ports open or a VPN tunnel.
- **Custom client-server (gRPC/REST):** Alternatively, we implement a lightweight RPC mechanism. For example, a small ROS 2 client node on Jetson could take an image and send an HTTP/gRPC request to a server endpoint that runs the model (perhaps using NVIDIA Triton Inference Server or a simple Flask app). The server would respond with the action data, which the client translates into a ROS message for the adapter. This method can be more firewall-friendly (e.g. a single TCP port) and even allow using cloud services. However, it introduces more custom code outside ROS. Given our system is ROS-centric, we lean toward keeping it in ROS if possible, but we remain open to using an RPC if ROS 2 networking proves too latent or complex to secure.
- *Latency mitigation:* Offloading introduces network latency and potential jitter. To mitigate this:
- We will compress and reduce data sent. For instance, the Jetson can downsample images or send JPEG-compressed images to the server to cut bandwidth. We might not send every video frame; if running at 10 Hz, that could be okay. We will measure the round-trip latency. If it's, say, 50ms, that's manageable for ~10–15 Hz control. If higher, we might further lower image resolution or use grayscale.
- The adapter can implement **latency compensation** strategies: e.g. if there's a predictable delay, the adapter could extrapolate the robot's motion during that delay (perhaps negligible if robot is mostly static when manipulating). We can also use the last known command to "coast" for one cycle if a new command hasn't arrived (preventing sudden halts).
- We will include a **timeout and fallback**: if the remote policy call fails or is too slow, the adapter will either stop the robot or fall back to a simpler built-in policy. For example, a fallback mode might be a scripted behavior to safely retract the arm or maintain a hold position until commands resume. This prevents unsafe behavior if connectivity is lost.

- We can also explore **predictive planning**: having the policy predict a sequence of actions rather than one step, though that depends on model capability. OpenVLA being an autoregressive policy might output incremental actions rather than a full trajectory.
- *Communication security:* In a production deployment, it's critical to secure the robot-server link. If using ROS 2 DDS directly, we will enable **DDS Security (SROS2)** to encrypt communications and authenticate nodes[13][14]. This ensures that camera feeds or motion commands aren't intercepted or tampered with over the network. SROS2 uses certificates and permissions to allow only authorized nodes (our Jetson and server nodes) to talk, and encrypts the data. We would generate a shared keystore for the two machines. If using a custom gRPC, we'd use TLS encryption and API keys for auth. Additionally, a VPN (e.g. WireGuard) between the robot network and the server network could add another layer of security so that even the DDS traffic is tunneled.
- *Docker deployment:* We will provide Dockerfiles for each side. The `Dockerfile.jetson` will contain all ROS 2 packages and required dependencies, optimized for arm64 and Jetson's CUDA (likely based on NVIDIA's ROS/JetPack base image). The `Dockerfile.policy` for the server will have the model and runtime (could be a lighter ROS install or just the necessary inference code with something like FastAPI/gRPC server). Using Docker ensures consistency between dev and prod environments and makes it easier to deploy updates. We'll also supply a `docker-compose.yml` that illustrates how to launch the Jetson container (with `runtime=nvidia` to access the Jetson's GPU if needed) and the server container together. In the Jetson-only scenario, one could still use the same container on Jetson and not launch the server container.
- *Networking (DDS specifics):* If using ROS 2 across hosts, we may configure **Fast DDS** or **CycloneDDS** to use a known multicast or unicast setup. E.g., Fast DDS allows an XML profile where we can specify the server's IP as a peer for discovery, bypassing the need for multicast discovery. We will document steps such as setting `FASTRTPS_DEFAULT_PROFILES_FILE` or analogous for Cyclone to ensure the two nodes discover each other. We will also consider Quality of Service (QoS) settings: for example, the image topic might be set to `best effort` reliability instead of reliable, to avoid latency spikes if a frame is dropped. We can tolerate dropping some images rather than queueing them. Control commands should be reliable but small in size.
- *Resource allocation:* We'll ensure the Jetson has enough CPU for the adapter and Nav2, and dedicate the GPU mostly to perception or maybe a smaller model. The remote server will handle the heavy model. In essence, this is a form of **edge-cloud offloading**, which is recommended when on-board resources are insufficient[11][15].

In both profiles, we will implement **DDS Quality of Service** tuning for real-time: e.g. using appropriate DDS history settings (perhaps keep last 1 message for images to avoid lag), and priority for control topics. The Jetson + server setup will be tested in a LAN setting first (to simulate an on-premises GPU server or a nearby edge box). If needed for truly remote (cloud), we'll incorporate that feedback (likely focusing on robust network and security).

Security basics: Aside from encrypting comms, we will also follow OS hardening on the Jetson: e.g. enabling Ubuntu firewall (ufw) to restrict open ports except those needed for ROS 2 communication with the known server. If using Docker, we might run containers as non-root where possible and use read-only file systems, etc., to limit damage from any compromise. We'll manage secrets (like certificate files for SROS2) carefully, likely not committing them to the repo but generating them per deployment.

Maintenance: We plan to use containerization or VM isolation for the remote server for easier maintenance (the Dockerfile can include model download from HuggingFace, etc., so the server can be re-deployed in one command). On the Jetson side, some may run natively (for lower latency); we still provide a Docker as an option, but some users might just install directly. We will supply instructions for both.

By providing these two profiles, we ensure flexibility: users can start with everything on the robot (simpler, no network dependence) and later scale up to a distributed setup for better performance. The code is written such that the mode is configurable – e.g., the policy node could be switched between "local mode" (load model locally) and "remote client mode" (send requests to server) via a parameter or launch argument. This design follows best practices for handling heavy AI models in robotics by offloading when needed[16], while keeping latency-sensitive control on the edge.

7. Engineering Quality Gates

To achieve a production-quality repository, we will enforce several engineering practices and **CI/CD pipelines**:

- **Continuous Integration (CI):** We set up GitHub Actions (see `.github/workflows/ci.yml`) to run on every push and pull request. The CI pipeline will:
- **Build Verification:** Use `colcon build` to compile all packages in an isolated environment (with `--merge-install` or similar). This ensures that the repository is always in a buildable state. We'll target ROS 2 Humble in CI (with perhaps a Docker container of ROS Humble for consistency).
- **Linters and Style Checks:** Run ROS 2 linters like `ament_lint_auto`. This includes `ament_cpplint` and `ament_copyright` for C++ code style, `ament_flake8` or `flake8` for Python style, `xmllint` for XML files (launch and package manifests), and `black`/`yapf` for Python formatting if desired. Consistent style is enforced to improve readability and catch common issues early. The CI will fail if linting issues are found.
- **Static Analysis:** We can integrate `ament_clang_tidy` for C++ static analysis (to catch potential bugs or bad practices in C++ adapter code) and maybe `pyflakes`/`pylint` for Python static checks in the policy code. This acts as a "quality gate" to maintain code health.
- **Unit Tests:** We will write unit tests for critical components (especially the adapter logic and any utility functions). For C++, using GTest via `ament_add_gtest`; for Python, using `pytest`. For example, we might have a test for the inverse kinematics function in the adapter (feeding it known transforms and seeing if it computes expected joint angles), or a test that the observation message builder correctly normalizes image data. The CI will run `colcon test` and fail if any test fails. Code coverage can be measured as well to ensure we have adequate test coverage.
- **Integration/Sanity Tests:** A minimal integration test might be run in CI without requiring a full simulation (to keep it lightweight). For instance, we could launch a dummy node that publishes a fake image and joint state, run the policy node in a special "dry-run mode" (maybe a trivial model that outputs a fixed action), and verify the adapter produces a expected output. This ensures that launch files and topic connections are basically working. We might use ROS 2 launch testing framework to automate this.
- **Simulation Test (nightly or on-demand):** If feasible, we might have a headless Gazebo test where we spawn a simple world with the robot and issue a command, verifying it reaches a goal. However, running Gazebo in CI for every commit can be slow and flaky. Instead, we could do this in a scheduled pipeline (e.g. nightly builds or a manual trigger before releases). In any case, before any tagged release, we would run a full sim test as a quality gate.

- **Continuous Delivery/Deployment (CD):** While not initially required, we can set up workflows to build Docker images (for Jetson and server) and possibly even push them to a registry when changes are merged to main. This way, the latest code is readily deployable. We'll also version the repository with tags, and could integrate a release workflow that attaches a `.repos` file or documentation for each release.

- **Dependency Management:** We will manage dependencies carefully to ensure reproducibility:

- We use **rosdep** to handle all ROS package dependencies and system libraries. Each package's `package.xml` will list dependencies (e.g. `rclcpp`, `sensor_msgs`, `cv_bridge`, etc.). Using rosdep means a developer can run `rosdep install` to get all apt packages (like OpenCV, PCL, etc.) on their system[17]. This avoids manual error-prone setup.
- For Python-specific dependencies (especially ML ones like `transformers`, `torch`, `huggingface_hub`, etc.), we choose to use a **Python virtual environment with pinned requirements**. We will include either a `requirements.txt` or a Poetry `pyproject.toml` – after consideration, we lean towards a **requirements.txt + venv** for simplicity. The reason is that ROS 2 integrates with system Python, and using Poetry or Conda could conflict with ROS's environment. A virtualenv allows isolation of Python libs without interfering with system packages. We will pin exact versions of critical libraries (e.g. "transformers==4.x.y", "torch==2.x.y for aarch64 with CUDA"). The `install_piper.sh` already uses pip for `piper_sdk`. We'll do similarly for ML libs. This approach is straightforward: developers create a venv (or we do it in Docker) and pip install the requirements.
    - *Justification vs alternatives:* We considered **Conda**, which can provide precompiled scientific Python libs and manage versions. Conda could be useful on Jetson to get specific versions of PyTorch with CUDA support. However, Conda environments can complicate ROS usage (since ROS expects system-installed packages). Using Conda inside ROS Docker or alongside apt might lead to conflicts (multiple pythons). Given a small team and to keep things standard, we prefer sticking to rosdep + pip.
    - We also considered **Poetry** for dependency locking. Poetry is great for pure Python projects because it can lock exact versions and manage virtualenvs automatically. In our case, we have a hybrid (C++ and Python). We could still use Poetry just for the `manipulation_policy` package to ensure all ML deps are locked. This is an option and we might adopt it if pip becomes unwieldy. Initially, we will keep it simpler with pip/requirements and possibly use pip's hashing (`requirements.txt` with hash pins or using `pip-compile` from pip-tools for a lock file).
- We will document the exact versions of critical dependencies in `config/piper_versions.yaml` and possibly a `requirements.txt`. For example, which commit of `piper_ros` was last tested, which version of OpenVLA model (we might tag the model weights with a version or commit from HuggingFace). This ensures that years later, one can reproduce the working combination by following the docs, even if upstream repos have changed.
- **System dependencies:** We rely on ROS 2 Humble and Ubuntu 22.04 – we'll note that JetPack (for Jetson) includes CUDA, etc., so we won't list those, but if any specific system tuning is needed (like enabling CAN interfaces or installing specific kernel modules for the Ranger's motor), that will be in docs or scripts.

- **Code Reviews and Branching:** As an internal practice, we'll enforce code reviews for all merges to main. The repository will likely use a branching strategy (feature branches merging into main via PR). We might also maintain a dev branch for integration testing, but with 1–3 devs a simpler approach is fine. Automated checks via CI must pass before merge. We can also set up protection so that the main branch requires CI passing and at least one review approval.

- **Issue Tracking & Project Board:** We'll maintain a GitHub project board or use issues to track the backlog (see section 8). Each issue should have clear acceptance criteria (e.g. "CI passes all tests", "Robot is able to pick up object in sim 80% of time" for higher-level tasks).

- **Logging and Monitoring:** In runtime, we'll use ROS 2 logging (rclcpp/rclpy logs) appropriately (debug for detailed internal steps, info for high-level actions, warn/error for issues). This helps during testing and also if something goes wrong in production, logs can be checked (perhaps aggregated via rosbag or external tools). We might integrate a tracing system (ROS 2 has tracing capabilities) for performance tuning if needed.

- **Hardware-in-the-loop testing:** Not exactly a "gate", but as part of our definition of done for features, we plan to test on actual hardware (Ranger + Piper) for key scenarios. This will flush out real-world issues (latency, sensor noise, etc.) which our simulation might not capture fully. Those findings will feed back into software improvements (e.g. adding filtering in perception if real sensor is noisy).

In summary, the quality gates ensure that at any point, the code is well-formed, tested, and reproducible. By using CI to automate enforcement of these standards, we reduce the chance of regressions. Dependency management through rosdep and venv/pip strikes a balance between ROS ecosystem norms and the needs of ML packages. This disciplined engineering approach is crucial given the complexity and safety requirements of a mobile manipulator system.

8. Starter Backlog

To kick off development and ensure an end-to-end minimally viable system, we've outlined an initial backlog of issues (in rough order). Each issue represents a task or user story to implement, with the aim that after completing them, we have a working prototype of the manipulation intelligence stack:

1. **Repository Initialization and CI Setup** – *Create the base repository structure*. Define the ROS 2 workspace (`src/` with a dummy package). Set up GitHub Actions CI with a basic colcon build and lint job to establish the pipeline. **Acceptance:** CI passes on a hello-world package build and lint (using ament_lint).[1]

2. **Define ROS Interface Messages** – *Create the `manipulation_msgs` package*. Define custom messages/actions: e.g. `Observation.msg` (if needed to aggregate sensor data for policy), `ManipulationAction.action` (with result containing EEF target, etc.), or simpler `PolicyOutput.msg`. Also include any service definitions (if we plan a service for high-level commands). **Acceptance:** `.msg` and `.action` files are in place, and `colcon build` generates the interfaces with no issues.

3. **Perception Module Skeleton** – *Implement a basic `manipulation_perception` node*. It should subscribe to camera image (use a dummy image source if actual camera not available yet) and to `/joint_states`. For now, it can simply republish or log these to verify subscription works. If any image preprocessing needed (e.g., converting ROS Image to OpenCV format), include that. **Acceptance:** When running on the robot or simulation, the node successfully receives images and joint states (verify by console log or rqt topic echo).

4. **Integrate Piper Arm Driver** – *Ensure the Piper arm can be controlled via ROS2*. This might involve running `scripts/install_piper.sh` and writing a launch to bring up Piper's driver node (from the agilex `piper_ros` package). **Subtasks:** (a) Verify that `piper_ros` (Humble) is installed and the Piper arm responds to commands; (b) Test a simple arm movement: e.g. call a service or publish

a joint target to move one joint, observe arm motion. **Acceptance:** We have a documented procedure to enable the Piper arm and can command it through ROS (joint angle or trajectory). A small demo: arm moves to a predefined pose via our system (even if just calling the driver directly). This proves our environment and dependency on Piper is correct.

5. **Adapter Basic Implementation** – *Create the `manipulation_adapter` node that can command base and arm.* Initially, implement simple pass-through or manual control: for example, add a service `/adapter/test_move` that when called, commands the arm to a fixed pose and/or drives the base forward 0.5m. This is to establish that the adapter can talk to `/cmd_vel` and the arm controller topic. **Acceptance:** Calling the test service results in the robot moving (in simulation or real): e.g. base wheels spin or arm joint moves. This validates end-to-end connectivity from our stack to hardware.

6. **OpenVLA Policy Node Integration (Stub)** – *Set up the `manipulation_policy` node with a stubbed model.* Since running the real model is complex, first implement the node to simulate policy output. For example, on receiving an observation, have it output a fixed action or a simple rule (like "move end-effector 10cm forward"). This allows us to test the loop without the actual model. Integrate Hugging Face libraries in the environment so later we can load the real model. **Acceptance:** Policy node subscribes to image/joint state (or is triggered by a timer) and publishes a `PolicyOutput` message regularly. The adapter receives this (even if it's nonsense data for now). The plumbing between perception → policy → adapter is confirmed.

7. **Simulation Environment Setup** – *Prepare a Gazebo simulation for the mobile manipulator.* This includes obtaining the URDF of the Ranger + Piper robot. Possibly use the URDF from `ranger-garden-assistant` (export it or reference it). Create a simple world (flat ground or a table with an object) for manipulation testing. Include a Gazebo plugin or controller so that `/cmd_vel` moves a simulated base and the arm can be controlled (this might require a MoveIt or gazebo_ros2_control setup for the arm). **Acceptance:** We can launch `sim_launch.py` and see the robot in Gazebo, and manually command the arm or base via ROS 2 (e.g., publish to `/cmd_vel` and the sim robot moves, send a joint trajectory and arm moves in sim). This provides a safe playground for further testing issues.

8. **Basic Pick-and-Place Scenario (Demo in Sim)** – *Implement a simple hard-coded manipulation behavior through the system.* For example, script a sequence: "drive forward to object, lower arm, close gripper". This could be done by sending fake observations to policy or bypassing policy for a moment. The goal is to demonstrate that the adapter can coordinate base and arm: e.g., place an object at known coordinates, have the robot move within reach and pick it up. We might use MoveIt to plan the arm path for this known scenario to validate kinematics. **Acceptance:** In simulation, the robot successfully performs a simple pick (even if using hard-coded waypoints). This builds confidence in the coordinate transforms and controller operation.

9. **Policy Model Integration (Online)** – *Integrate the actual OpenVLA/LeRobot model into `manipulation_policy`.* This involves downloading the model (possibly a 7B weights from HuggingFace, which could be ~> 10GB, ensure we handle that), setting up the model pipeline in code, and performing a test inference on the Jetson. We might start with a smaller checkpoint (like a 1B model or quantized version) to ensure it runs. Optimize if needed (use FP16). **Acceptance:** The policy node can load the model (within memory limits) and run inference on a sample image+command, outputting a reasonable action. This might be tested off-robot on a dev machine first due to resource demands, but final acceptance is it runs on Jetson albeit perhaps slowly. We'll likely iterate on performance after this.

10. **Remote Inference Mode** – *Implement the split deployment capabilities.* Develop a mechanism (perhaps a ROS 2 service or separate node) for sending observations to a remote server and getting actions. For now, could simulate the server on the same PC for testing. Define the message or RPC call format (possibly reuse the Observation message for request and PolicyOutput for response). **Acceptance:** We run a "server" node (which may just echo back a dummy action) and the Jetson "client" successfully communicates and gets a response. This sets the stage for plugging in the real model server later.

11. **Latency & Fallback Handling** – *Enhance the adapter for reliability.* Add timeout handling: if no new policy output is received within X seconds, log a warning and maybe halt arm movement (to avoid stale commands). Also implement a basic fallback routine: e.g., if remote inference is down, perhaps switch to a "safe mode" where the robot only navigates or the arm goes to home position. **Acceptance:** Through testing (e.g., kill the policy node while running), the adapter recognizes the loss and stops issuing commands (the robot should stop moving rather than continue on last command indefinitely).

12. **End-to-End Field Test on Hardware** – *Take the integrated system and trial on the real robot (Ranger + Piper).* This involves running the base stack (Nav2, etc.) and our manipulation stack together. Test a simple task like "pick up object in front of robot" using the actual camera feed and a dummy policy (or if possible the real model with a simple prompt). Record any issues like coordinate misalignment or delays. **Acceptance:** The robot can at least attempt a manipulation: e.g., the arm moves towards an object. It's okay if success isn't perfect at this stage – the aim is to identify gaps (calibration, etc.) under real conditions. Issues found here will spawn new backlog items (e.g. "improve calibration of camera to base_link" or "tune gripper strength").

13. **Documentation and Usage Guides** – *Write comprehensive documentation.* Fill in `docs/usage.md` with instructions for setup, including how to run `install_piper.sh`, how to launch the stack, how to deploy remote server. Also document the ROS API (topics/services) in `api_reference.md`. Create a basic tutorial (maybe in README) for a new developer to run the system in simulation and on real hardware. **Acceptance:** All team members review the docs for clarity, and a newcomer can follow them to get the system running.

14. **Extend Policy Integration (HuggingFace & RoboNeuron)** – *Investigate advanced integration.* (Optional stretch) Test connecting the system with a language model or orchestrator. For example, use RoboNeuron's approach: feed a high-level command to an LLM that then triggers our policy. Or try a different VLA policy from LeRobot's model zoo (if available) to see if swapping is easy. **Acceptance:** A report or demo showing that our system could interface with an LLM (even if just a simulation where an LLM chooses when to call our action). This is more research-oriented and can be deferred, but listing it to keep it on the radar.

15. **Project Release 0.1** – *Cleanup and prepare a tagged release.* Close out remaining minor issues (such as fixing any memory leaks, finalizing parameter tuning for speeds, etc.). Ensure all tests pass, and tag a v0.1.0 release on GitHub. **Acceptance:** We have a stable commit that we consider a baseline MVP. At this point, the backlog can be updated with new goals (like improving success rates, adding more tasks, UI integration, etc.).

These issues are organized to build the system incrementally, **de-risking high unknowns early** (like Piper integration and model integration). We identify highest-risk unknowns as the real-time performance of the large model on Jetson and the coordination of base and arm. By issue 9 and 12, we directly confront those: if OpenVLA is too slow, we'll know and can pivot (e.g. use a smaller model or rely more on remote). If base/arm coordination has problems, we'll catch them in simulation and field tests and can adjust (maybe incorporate MoveIt's planning sooner, or add more sensors for better perception). Each completed issue brings the project closer to a functioning intelligent mobile manipulator that generalizes beyond the Ranger scenario.

Throughout the backlog, we maximize generality: for instance, using standard ROS interfaces (Navigation2 action, FollowJointTrajectory) means other robots with different arms or bases could reuse our software by adapting configuration. By the end of this backlog, we aim to have a repository that is **structured, documented, tested, and demonstrates core functionality** of a mobile manipulator "brain," ready for further refinement and expansion.

---

[1] Packages | Articulated Robotics

https://articulatedrobotics.xyz/tutorials/ready-for-ros/packages/

[2] Source control, packages, and metapackages in 2021? - ROS Answers archive

https://answers.ros.org/question/369595/

[3] [4] OpenVLA: An Open-Source Vision-Language-Action Model

https://openvla.github.io/

[5] [6] [2512.10394] RoboNeuron: A Modular Framework Linking Foundation Models and ROS for Embodied AI

https://arxiv.org/abs/2512.10394

[7] [8] GitHub - agilexrobotics/piper_ros: piper ros workspace

https://github.com/agilexrobotics/piper_ros

[9] [10] NanoVLA: Routing Decoupled Vision-Language Understanding for Nano-sized Generalist Robotic Policies | OpenReview

https://openreview.net/forum?id=yeHBrNVZoV

[11] [12] [15] [16] Multi-Model AI Resource Allocation for Humanoid Robots: A Survey on Jetson Orin Nano Super - DEV Community

https://dev.to/ankk98/multi-model-ai-resource-allocation-for-humanoid-robots-a-survey-on-jetson-orin-nano-super-310i

[13] [14] Securing ROS2 Nodes with SROS2: Encryption and Permissions for Robot Communications - DEV Community

https://dev.to/sebos/securing-ros2-nodes-with-sros2-encryption-and-permissions-for-robot-communications-m55

[17] Crucial ROS2 Dependencies Solved: The Definitive Beginner's Guide

https://robotisim.com/ros2-dependencies-beginners-guide/