ChatGPT

Project Lens – Compute & Primary Storage Roadmap

Mission Statement: Design and implement the onboard computing and storage for a mobile autonomous archival robot (~0.75–1.0 m size). This includes providing ample compute horsepower and high-write-endurance storage for real-time autonomy tasks and raw sensor data logging. The roadmap below breaks this mission into a series of mini-projects that a motivated beginner can tackle over the course of a year. Each mini-project has a clear goal, recommended standard components/tools, rationale, and expected outcomes. The projects are listed in a logical build order (later projects may rely on earlier ones).

Mini-Project 1: Select and Set Up the Main Computing Board (SBC/SoM)

- **Goal:** Choose a suitable Single Board Computer (SBC) or System-on-Module (SoM) as the robot's "brain", then install and configure its operating system (with ROS 2) for development. This will be the primary computer handling sensor processing, decision making, and high-level control.
- **Recommended Components:** A well-supported SBC with CPU/GPU acceleration for robotics. For example, an **NVIDIA Jetson** dev kit (such as the Jetson Orin Nano or Xavier NX) is ideal for heavy AI and vision tasks, offering a built-in GPU for real-time computer vision and deep learning at the edge [1]. Alternatively, a **Raspberry Pi 4** (or Compute Module 4) can be used for simpler projects or as a budget-friendly option, benefiting from a huge community and many tutorials [2] (though lacking a powerful GPU for advanced AI). Along with the board, you'll need a compatible OS image (e.g. Ubuntu 22.04 with ROS 2 Humble for Jetson or RPi OS/Ubuntu for Raspberry Pi), a power supply, and peripherals (SD card if needed, HDMI, keyboard/mouse for initial setup or a serial console).
- **Rationale:** The main board will run ROS 2 nodes, perform sensor fusion, and possibly run CPU/GPU-intensive algorithms (like SLAM or neural networks). We prioritize boards with strong community support and long-term availability. NVIDIA Jetson is widely used in robotics because it's designed for autonomous machines and provides lots of compute for AI and vision [1]. Raspberry Pi is extremely popular for beginners due to its low cost, versatility, and extensive documentation [3] [2], and can handle moderate workloads (potentially augmented with a Google Coral or Intel Neural Stick for AI acceleration if needed). No strict budget means you can choose the board that best fits the performance needs.
- **Steps:**
- **Research & Selection:** Compare a couple of candidate boards (Jetson vs. Raspberry Pi, etc.) in terms of CPU speed, GPU capability, RAM, and IO. Consider what your autonomy tasks require – for instance, running neural network inference for vision will greatly benefit from a Jetson's CUDA cores, whereas basic navigation with lighter processing might run on a Pi. Community support and available examples are important for a beginner (Jetson has official tutorials for ROS, and Pi has countless community projects).
- **Initial Setup:** Purchase the chosen board and any required accessories (for Jetson: a dev kit usually includes needed parts; for a Pi: you'll need a quality 5V power supply, a microSD card or SSD, etc.). Follow the official getting-started guide for your board to flash the OS. For Jetson, flash the NVIDIA

JetPack image (Ubuntu-based) onto the device (JetPack includes drivers and SDKs for the GPU). For Raspberry Pi, flash the latest Raspberry Pi OS or Ubuntu Server 22.04 image to a microSD card (or eMMC/SSD if using CM4 or Pi 4 with USB boot).

- **Bring-up and Basic Testing:** Boot the board, go through initial configuration (expand filesystem, set up usernames, enable SSH for convenience, etc.). Next, install **ROS 2** (e.g., ROS 2 Humble for Ubuntu 22.04) following standard instructions. Test a simple ROS 2 talker/listener demo or run a basic example (if Jetson, you could try a sample using the GPU like running a TensorRT optimized model, or if Pi, try accessing the camera and printing images). Verify that the board can access the internet, all key interfaces (USB, camera, etc.) work, and that you can program it (try a "Hello World" in Python or C++).
- **Expected Outcome:** By the end of this mini-project, you will have a fully functional main computer for the robot, with a proper OS and ROS 2 setup. You will have learned how to flash OS images, configure an SBC, and run basic robotics software on it. This foundational step sets the stage for all subsequent projects, as it provides the development and execution platform for the robot's software.

Mini-Project 2: Ensure Adequate Cooling and Thermal Performance

- **Goal:** Design and add a cooling solution for your chosen SBC/SoM to guarantee it can sustain heavy workloads without overheating or throttling. This includes selecting appropriate heat sinks/fans and testing the system under load to measure temperatures. The goal is to establish a **thermal design margin** so the CPU/GPU can run at near 100% utilization reliably.
- **Components:** A compatible **heatsink and/or fan** for the SBC (many dev kits come with basic cooling; if not, purchase an official cooling kit or a third-party solution that fits your board). For example, Jetson dev kits often include a small heatsink+fan, but you might upgrade to a larger aftermarket heatsink or a case with better airflow. Tools include temperature monitoring software (e.g., `tegrastats` for Jetson, or `vcgencmd measure_temp` for Raspberry Pi, or generic tools like `watch -n1 sensors`). You may also use a stress-testing utility (like `stress-ng`, `sysbench` or running a heavy ROS node or AI model) to generate load.
- **Rationale:** High-performance SBCs can run hot when executing intensive tasks (e.g., vision processing or neural network inference). Without proper cooling, the CPU/GPU will reduce its speed to prevent damage (thermal throttling), leading to degraded performance or even system instability. A well-designed thermal solution keeps temperatures in check, **preventing throttling and maintaining consistent high performance** [4] for AI, robotics, and edge computing applications. This project teaches good hardware practice and helps your robot run 24/7 reliably.
- **Steps:**
- **Baseline Temperature Test:** First, measure how the board behaves out-of-the-box. Record the idle temperature and then run a workload (for example, compile something, run a sample AI inference, or use a stress test tool) for several minutes. Monitor temperatures and note if the system clock speeds reduce (some SBCs log a message or you can see frequency drop). This gives a baseline and shows if the default cooling is sufficient.
- **Install Improved Cooling:** If the board doesn't already have a heatsink/fan, install one now. For a Jetson, you might mount a heatsink and plug in a fan to the board's fan header. For Raspberry Pi, you can use stick-on heatsinks and a small 5V fan, or even better, a case with a fan. Ensure good contact (use thermal paste or pads as needed). Double-check the fan power (5V or 12V as appropriate) and that it spins up.
- **Thermal Stress Test:** Re-run the heavy workload (or even a combination, like running GPU inference and CPU tasks simultaneously). Monitor the temperature over time (you want the system to reach

steady-state). Ideally, it should stay below the throttling threshold (e.g., many boards throttle ~80°C or above). If it still hits high temps, consider more aggressive cooling: e.g., a bigger heat sink, higher airflow fan, or even active methods like a small blower or liquid cooling (usually not necessary for these SBCs). Also consider the operating environment of the robot – if it will be in an enclosure, account for that (you might design vents or add an additional chassis fan).

- **Thermal Design Margin:** Aim to have some headroom – for instance, if under worst-case load you reach 70°C, that's good margin below an 85°C throttle point. This ensures on a hot day or with dust buildup, you still won't overheat. Document the solution (maybe take note of current draws of fans, etc., for power budgeting) and any modifications to the case/enclosure for airflow.
- **Expected Outcome:** You will have upgraded the robot's main computer with proper cooling such that it can run continuously at high performance. You'll learn how to monitor system thermals and understand the relationship between load and temperature. The result is a stable compute platform ready for the heavy lifting of autonomy algorithms without fear of overheating.

Mini-Project 3: Enable Real-Time Linux for Deterministic Performance

- **Goal:** Configure the main board's operating system for **real-time operation**. This involves installing or patching a **Real-Time Preempt** kernel (PREEMPT_RT) or at least a low-latency kernel, and tuning the system for scheduling predictability. The end goal is to reduce latency and jitter for time-critical processes (e.g., motor control loop, sensor reading tasks), making the system behave more like a real-time system. You'll also verify the improvements with simple tests.
- **Recommended Tools:** If using Ubuntu (as on Jetson or Pi), you can install a *low-latency kernel* (Ubuntu provides one via `apt`) or build a custom kernel with the PREEMPT_RT patch. For example, Ubuntu 22.04 has an official linux-lowlatency package that, while not fully real-time, often suffices [5]. For true real-time, download the kernel source and the corresponding RT patch from the Linux Foundation, then compile and install it. Tools like `cyclictest` (from rt-tests) can measure latency. Additionally, familiarize with Linux scheduler settings (e.g., using `chrt` to set thread priorities, isolating CPUs for real-time tasks, disabling frequency scaling or power-saving modes on those cores, etc.). ROS 2's real-time working group resources can also guide you on executor settings and real-time programming considerations.
- **Rationale:** A standard Linux kernel is not strictly real-time – under load, you might experience unpredictable delays (in the order of tens of milliseconds), which can be problematic for high-frequency control loops or precise timing. By using a real-time kernel, **trajectory control becomes smoother and more reliable**, as noted by robotics developers who saw non-smooth motion when not using real-time scheduling [6]. Real-time kernel (PREEMPT_RT) allows nearly all parts of the kernel to be preempted, drastically reducing worst-case scheduling latency. This mini-project is crucial if the robot needs hard/firm real-time guarantees on the main CPU (complementing the use of a co-processor in the next project).
- **Steps:**
- **Real-Time Kernel Installation:** Choose the approach that fits your comfort level:
    - *Easier:* Install a low-latency kernel via your package manager (on x86 Ubuntu this is straightforward; on Jetson, NVIDIA provides an example RT patched kernel for some JetPack versions [7], otherwise you compile it). For Raspberry Pi, you might find community-provided RT kernel images (there are ready-made SD card images with ROS 2 and PREEMPT_RT for Pi [8]).

    - *Advanced:* Build the PREEMPT_RT kernel yourself. This requires downloading the kernel source matching your OS version, applying the RT patch, configuring (you can start from the current kernel config and enable `PREEMPT_RT`), and compiling. This can take some time (and ~30GB disk as a ROS tutorial notes). NVIDIA's forums and JetsonHacks blogs have step-by-step guides for compiling an RT kernel on Jetson [9] [10].
- **Configure System for Real-Time:** Once the RT or low-latency kernel is installed and booted into, do additional tuning. For example, isolate a CPU core (or two) for real-time tasks using kernel boot parameters (e.g., `isolcpus`), so that general OS threads don't run on those. Adjust IRQ affinities if necessary so important device interrupts go to isolated cores. Use `ulimit` and `/etc/security/limits.conf` to allow real-time scheduling for your user or specific processes (e.g., give ROS executables permission to use `FIFO` scheduler with priority). Essentially, set things up so your critical ROS nodes can run at high priority without interference.
- **Testing & Verification:** Run `cyclictest` with a high priority and measure the max latency over a few minutes to see the improvement (a PREEMPT_RT kernel on decent hardware can achieve tens of microseconds latency in ideal conditions). You can also write a simple program or use the ROS 2 pendulum demo to see differences with and without RT. Observe if sensor readings or actuator commands now come at more stable intervals. If you have specific requirements (e.g., control loop needs <1ms jitter), check that you meet them.
- **Real-Time in ROS 2:** Note that simply having a real-time OS doesn't automatically make your ROS 2 stack real-time – you must also write real-time safe code (no blocking calls, avoid dynamic memory allocation in loops, etc.). This is a deeper topic, but at least now your OS won't be the limiting factor. Document the kernel version and settings you ended up with for future reference.
- **Expected Outcome:** Your robot's main OS is now configured for improved real-time performance. You will have learned how to patch or configure a Linux kernel, and gained insight into low-level OS behavior. The robot will be more responsive and reliable in timing-critical tasks – for instance, control loops running on the main CPU will have less jitter, improving motion stability. This sets a strong foundation for integrating the co-processor and other software in subsequent steps.

Mini-Project 4: Integrate a Real-Time Co-Processor (Microcontroller or FPGA)

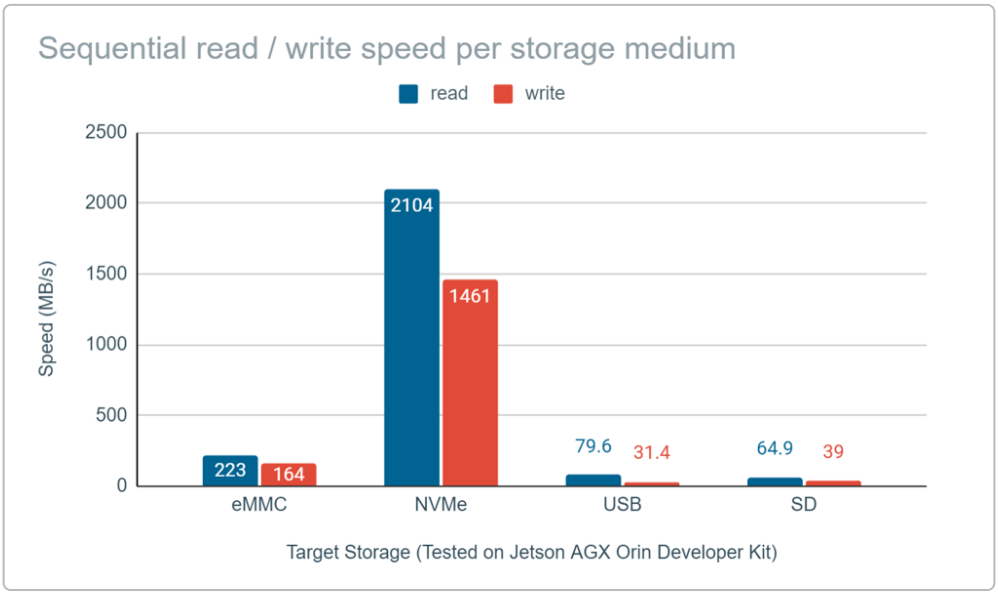
- **Goal:** Add a dedicated **microcontroller** (or an FPGA, if inclined) to offload hard real-time tasks from the main computer and to serve as a safety watchdog. Concretely, you will select a microcontroller unit (MCU), program it to handle low-level tasks such as motor control loop timing or sensor polling with precise intervals, and implement a watchdog mechanism where the MCU monitors the main SBC's health. This project will result in a two-tier computing system: the SBC for high-level planning and the MCU for time-critical control.
- **Components:** A suitable microcontroller board with good community support. Examples:
- **Arduino-family MCU:** An Arduino Uno or Mega (easy for beginners, great community). However, for more performance, consider an **Arduino Due** (32-bit ARM) or a **Teensy 4.0/4.1** (Cortex-M7 at 600 MHz, lots of IO – powerful and Arduino IDE compatible). These provide more headroom for complex real-time tasks.
- **Raspberry Pi Pico** (RP2040 microcontroller, dual-core, MicroPython or C++ support) is another cheap and capable option, and it's supported by the ROS 2 *micro-ROS* project.
- **STM32 or ESP32 dev boards:** Many ROS 2 users employ these with micro-ROS. For instance, an **STM32 Nucleo** board or an **Esp32** can run FreeRTOS and integrate with ROS 2 messages.

- (FPGA option: If you have FPGA experience or interest, a small FPGA like an Intel Cyclone IV or Lattice iCE40 could be used to implement ultra-fast control logic. However, this is advanced and the learning curve is steep. Most beginners should start with an MCU, which is easier to program.)

Other needed items include logic level shifters or circuitry if the MCU operates at a different voltage than SBC (e.g., level shifting for UART if 5V vs 3.3V), and wires or a prototyping board to connect the MCU to the main board (via USB, UART, I^2C, or SPI). Also, choose a programming environment: e.g., the Arduino IDE for simplicity, or PlatformIO for more advanced development. If using micro-ROS, you'll need to set up the build environment for the MCU (which can be complex but micro-ROS has fine documentation). - **Rationale:** Even with a real-time OS, a general-purpose SBC cannot always guarantee *hard* real-time performance, especially under heavy CPU load. Microcontrollers, however, excel at predictable timing and direct hardware control – they run simple firmware loops with minimal jitter. **Almost every robotic system uses MCUs** for tasks like reading sensors, controlling motors, and handling interrupts [11] . Reasons include: direct hardware access (analog/digital I/O), handling sub-millisecond timing, and reducing the load on the main CPU. Furthermore, an MCU can act as an independent **watchdog** that resets or alerts if the main system crashes (increasing reliability). This project will teach you embedded programming and how to make the SBC and MCU work in tandem as a robust system. - **Steps:** 1. **Select & Set Up MCU:** Choose one of the suggested microcontroller boards. For learning purposes, something like a **Teensy 4.x** is great (Arduino compatible but much faster than typical Arduinos). Set up the toolchain (install Arduino IDE or vendor IDE, board support packages, etc.). Verify you can upload a basic blink program to the MCU. 2. **Implement a Hard Real-Time Task:** Identify a task to offload. A classic example is a **motor control loop**: e.g., reading an encoder and updating motor PWM at 1 kHz consistently. Write firmware on the MCU to do this. Another example is reading an IMU sensor via SPI and filtering data at a high rate. Because the MCU is separate, it can maintain this loop timing independent of what the SBC is doing. 3. **Communication with SBC:** Decide how the MCU and SBC will communicate. A simple method is via **serial (UART)** – you can send status or sensor readings from the MCU to SBC, and send commands from SBC to MCU. For instance, the MCU could stream wheel encoder counts or IMU data, and listen for velocity commands. Alternatively, if using micro-ROS, set up the micro-ROS agent on the SBC and have the MCU act as a ROS 2 node (this would let you send ROS 2 messages between them, e.g., the MCU directly publishes sensor topics). Start simple: maybe use plain serial or I2C initially to grasp the concept. 4. **Watchdog Mechanism:** Implement a watchdog to supervise the SBC: - **On MCU side:** program a timer that expects a "heartbeat" signal from the SBC at regular intervals (for example, the SBC could toggle a particular GPIO line or send a specific message every second). If the heartbeat stops (meaning the SBC is frozen or crashed), the MCU can take action. Action could be turning on an LED/buzzer to indicate fault, or if hardware allows, toggling a reset line or power-cycling the main board. Many MCUs have internal watchdogs too, but here we're implementing an external one for the SBC. - **On SBC side:** write a simple ROS 2 node or script that periodically signals the MCU (e.g., sends a byte over serial or toggles a pin via GPIO) to assure it "I'm alive". Make sure this runs with high priority (perhaps as a real-time thread) so that it can even preempt other tasks if the system is under load. - Test the watchdog: intentionally stop the heartbeat (kill the process or simulate a hang) and verify the MCU detects it. This will give you confidence that if the main computer locks up, the co-processor can notice and potentially initiate a safe shutdown or reboot sequence. 5. **Optional - Integrate with ROS 2:** If not using micro-ROS, you can still integrate at a higher level. For example, have the SBC read data from the MCU and publish it as ROS topics, or have the SBC subscribe to ROS commands and forward them to the MCU. This exercise will teach you how to connect ROS with lower-level microcontroller code, a common practice (e.g., many ROS-based robots use an MCU running Arduino code to interface with motors, known as the "motor controller board"). - **Expected Outcome:** By completing this project, you will have a functioning **two-processor system**: the main board and a real-time microcontroller working together. You'll have learned how to program microcontrollers for real-time tasks, and how to establish reliable communication between an MCU and a Linux SBC. The MCU

will handle time-sensitive operations (ensuring, for instance, consistent motor PWM updates even if the SBC is busy) and improve safety via the watchdog. In essence, you are replicating a professional robotics setup where the high-level decisions come from a powerful computer and the low-level actuator/sensor management is done by dedicated controllers [11]. This makes your robot more robust and responsive.

Mini-Project 5: Set Up High-Endurance Primary Storage (NVMe SSD)



Sequential read/write speed comparison for different storage media on a Jetson platform. NVMe SSD (PCIe interface) provides order-of-magnitude higher throughput than USB or SD card storage, which is crucial for high data-rate logging. NVMe drives also have sophisticated controllers that **spread writes across many flash cells**, giving them much higher write endurance than typical microSD cards [12]. Using an NVMe as primary storage greatly improves the robot's data capacity, I/O speed, and long-term reliability for logging large sensor streams.

- **Goal:** Install and configure a **NVMe SSD** as the robot's tier-1 storage. This will hold the operating system (if possible) and especially all the data logs from sensors. You will set up appropriate partitions (for OS and data separation), enable wear leveling features (mostly inherent to the SSD), and ensure the file system is robust for constant writes. By the end, the robot will have ample fast storage for both the OS and high-frequency data recording (camera images, Lidar scans, etc.), avoiding the pitfalls of SD card wear-out.
- **Components:** A **PCIe NVMe SSD** in M.2 form factor (commonly 22×80 mm size). Since budget isn't a concern, choose a reputable brand and a model known for durability – for example, a Samsung 970 EVO Plus or 980, or a Western Digital Black SSD. Many of these have high TBW (Terabytes Written) ratings (e.g., hundreds of TB written endurance) and onboard DRAM for speed. Aim for at least 250 GB or 500 GB to store lots of data (depending on your logging needs). Also ensure your SBC has an M.2 **M-key slot** for NVMe (most Jetson dev kits do; Raspberry Pi 4 does not natively, but you can use a USB3 to NVMe adapter or a Pi Compute Module with an NVMe carrier). You'll also use partitioning tools (`gdisk` or `fdisk`) and file system tools (`mkfs.ext4` or others) in this project.
- **Rationale:** Data logging in robotics (recording sensor data, images, etc. for later analysis or for building maps) generates a **lot of writes**. Consumer microSD cards, while convenient, have limited

write endurance and can fail or corrupt after intensive use. In contrast, SSDs (especially NVMe) contain many flash memory cells and advanced wear-leveling algorithms – an SSD **spreads writes over many cells**, so it can handle far more write cycles than an SD [12] . Additionally, NVMe drives offer very high throughput (hundreds to thousands of MB/s) as shown above, whereas SD cards or USB drives are an order of magnitude slower. This means the robot can save high-bandwidth sensor data (like HD video streams) without dropping frames or slowing down. Overall, moving the primary storage to NVMe improves reliability, speed, and capacity.

- **Steps:**
- **Install the NVMe Physically:** Power off the SBC. Insert the NVMe SSD into the M.2 slot (typically labeled M.2 Key M). Secure it with the screw. (If the board doesn't have an M.2 slot, use an adapter as mentioned, but note that USB3 will bottleneck speeds and is less ideal [13] .) Ensure the device is recognized by powering on and checking `lsblk` or `dmesg` output – you should see a `/dev/nvme0n1`.

- **Plan the Partitioning Scheme:** Decide how to use the SSD:

    - You could **boot the OS from NVMe** (some SBCs support this directly, like newer Jetsons can boot from NVMe if flashed accordingly, or Raspberry Pi 4 can boot from USB/NVMe with bootloader config). Booting from NVMe will make the system more responsive.
    - Alternatively, keep the OS on the internal eMMC/SD for now and use the NVMe purely for data. This is simpler to start with: mount the NVMe as `/data` or `/home` where logs and bag files will go.

  For a robust system, it's common to **separate OS and data**. For instance: create two partitions on the NVMe – one for an eventual OS root filesystem, and one for data logs. Or if the board has internal eMMC, use that eMMC as the rootfs and dedicate most of NVMe to a `data` partition. The reasoning is that you might want to reflash or update the OS without wiping large data logs, and conversely, if data logging fills the disk, it won't crash the OS. 3. **Format and Mount:** Use `gdisk` (GPT is recommended for SSD) to create partitions. Perhaps allocate, say, ~64 GB for an OS partition and the rest for data (if you plan to migrate OS to NVMe), or one large partition if just for data. Format the partition(s) with a suitable filesystem. **ext4** is a good default for Linux (journaling provides some protection; it's generally fine for SSDs). If you want to maximize endurance, you might tune the filesystem (for example, mount with `noatime` to reduce writes, and ensure TRIM is enabled so the SSD can optimize free space).

  Mount the new partition to a path (e.g., `/mnt/data` or update `/etc/fstab` for automatic mounting). Test writing to it: for instance, run a quick benchmark like `dd if=/dev/zero of=/mnt/data/test.bin bs=100M count=10` to see write speed, and `dd if=/mnt/data/test.bin of=/dev/null bs=100M` for read speed. You should observe speeds in the hundreds of MB/s (depending on the board's PCIe throughput). 4. **Use for ROS Bag Storage:** Configure your robot's software to use the NVMe for heavy data. For example, if using ROS 2 bagging to record topics, set the output path to the NVMe mount. Similarly, direct any databases or large map files to that disk. This ensures those frequent writes go to the robust NVMe.

  (Optionally, implement log rotation or monitoring – even though NVMe is robust, it's not unlimited. For instance, a 256 GB drive with a TBW of 150 TB could technically endure writing the entire disk 600 times. It's plenty for a long time, but good to monitor usage.) 5. **OS on NVMe (Optional Advanced):** If you prefer to run the OS from the NVMe (for faster boot and a truly immutable base

on internal storage), you can attempt to clone the current OS to a new partition on NVMe and configure the bootloader. For Jetson, tools like `initrd` flashing or Nvidia's documentation [14] can guide this. For Raspberry Pi, you can enable USB boot in the firmware and flash an image to NVMe. This can be done later in Project 6 when implementing the dual system image. - **Expected Outcome:** The robot now has **fast and reliable storage** configured. You'll have learned how to work with NVMe SSDs on embedded systems, including partitioning and Linux mount management. Immediately, you should notice faster file I/O (for instance, loading large maps or saving bag files will be much quicker). More importantly, the high write endurance of the SSD means you can log large amounts of data continuously (e.g., hours of 4K video, Lidar scans) without fear of the storage failing early. This is a crucial upgrade for an "archival companion" robot that must safely store lots of sensor data.

Mini-Project 6: Robust System Image Management – Immutable OS, Secure Boot, and OTA Updates

- **Goal:** Implement a **robust software update and system integrity strategy** for the robot's main computer. This includes setting up an **immutable base operating system image** (which can be updated only in a controlled fashion), enabling **secure boot** (to ensure only trusted firmware/software runs), and using a **dual A/B partition scheme with rollback** so that software updates can be done safely (if an update fails, the robot will automatically revert to the last good version). Essentially, you will treat the robot like a deployable device that needs reliable updates and security, not just a dev kit. This mini-project ties together previous ones by ensuring the software running on your chosen SBC/SoM is maintainable long-term.
- **Recommended Tools/Frameworks:** There are existing solutions that you can leverage:
- **Ubuntu Core 24** or similar immutable Linux distros – Ubuntu Core uses read-only core system snaps and supports automatic rollback on update failure, which is aligned with our goals. (It's great for robotics deployments, offering infrastructure for secure, OTA updates with rollback [15].) However, ROS 2 is not officially packaged for Ubuntu Core yet, so you might need to containerize ROS or use snaps for ROS nodes.
- **Mender.io or SWUpdate/RAUC** – these are OTA update frameworks for embedded Linux that implement A/B partition updates. Mender, for instance, has an open-source edition that manages dual-partition updates and can integrate with your build system [16].
- **Yocto Project** – for ultimate control, you could create a custom Linux image with Yocto, using meta-security layers for secure boot and meta-updater for OTA updates. This is advanced and likely overkill for a beginner year, but worth mentioning as a pro option.
- **U-Boot and Bootloader configuration** – You will need to interact with the bootloader (e.g., U-Boot or L4T bootloader on Jetson) to set up dual boot slots and secure boot keys. Nvidia's documentation on A/B redundancy (they call it "rootfs A/B" which can be enabled in Jetson's OS configuration) may be relevant if using Jetson hardware.
- **Hardware secure elements:** If available (TPM chips or secure enclaves on the SBC), these can store keys for secure boot or attest device identity.

For simplicity, you might start with manual steps (like using two partitions and scripting the switch) before using a full OTA service. - **Rationale:** When your robot transitions from development to deployment (even if it's just your personal project, consider "deployment" as when it's running unattended for long periods), you want it to be **rock-solid and recoverable**. An immutable OS means the system's core software can't be accidentally corrupted or altered during runtime – all changes (like installing new packages or updates) are done in a controlled way (e.g., on an inactive partition or in read-only overlays). **Secure boot** ensures that

the robot only runs firmware signed by you, preventing unauthorized code from running if someone tampers with it. And using an **A/B update scheme** with rollback means you can update the robot remotely or headlessly with minimal risk – if the update fails or the new software crashes at boot, the bootloader will revert to the old working image automatically [15] . All these practices are used in industry (phones, cars, IoT devices) to maximize uptime. This project will familiarize you with these concepts and significantly increase the resilience of your robot's software. - **Steps:** 1. **Partition the Disk for Dual System Slots:** If not already done, decide where the two OS images will live. For example, if you are using the NVMe for OS, you could allocate two equal partitions (say, *systemA* and *systemB*). Or if using eMMC + NVMe, you might use eMMC as systemA and NVMe partition as systemB (though it's more typical to have both on the same device). Ensure the bootloader is capable of selecting between them. Many embedded devices keep a small boot partition with kernel and use device tree or boot config to point to either rootfsA or rootfsB. On Jetson, you might use the initrd flash mechanism to set this up (NVIDIA's bootloader supports A/B but it's disabled by default [17] ; you can enable rootfs redundancy and use `nvbootctrl` to switch slots). If this is too involved, you can simulate the concept by simply keeping a **full backup image** of your system that you can swap in if needed. 2. **Immutable Base Image:** Configure the Linux system to be mostly read-only. For instance, if sticking with Ubuntu, you can mount the root filesystem as read-only and use an overlayfs for /etc or other mutable directories. A simpler approach is to use **Docker containers or snaps** for all your applications (ROS nodes), so the base system remains clean. This way, at runtime, you aren't changing the base OS – all logs and data go to the data partition, applications are either containerized or come from read-only packages. Ubuntu Core inherently does this (the root system is read-only snaps). If you continue with Ubuntu Classic, consider using the **OSTree** model (used by Fedora Silverblue and others) where updates are delivered as atomic snapshots. As an exercise, you might try turning your current setup into a read-only root: for example, remount `/` as read-only and see what breaks, then whitelisting specific paths (you'll likely need /var and /etc writable to some extent unless you redesign things). 3. **Secure Boot Setup:** This step depends on your hardware. On a PC or some SBCs, secure boot involves UEFI keys – on Raspberry Pi, true secure boot is not available (it doesn't have a TPM by default, though you could incorporate one). On NVIDIA Jetson, there is a fuse-based secure boot mechanism (you burn keys into the module's fuses and from then on it will only boot images signed with your key). This is a **permanent** step, so do not enable it until you are absolutely ready. For learning, you can do a "secure boot test" by using U-Boot's sandbox or QEMU to simulate. At minimum, understand the process: you generate an RSA keypair, you sign the bootloader and kernel images with the private key, and you provision the public key in a secure element or fuse. Once in place, the boot ROM will refuse any unsigned or tampered code. If you prefer not to actually blow fuses on your dev kit, just document how you *would* do it. You can still implement secure boot on a conceptual level by checking signatures of software before installing updates, etc. 4. **Implement OTA Update with Rollback:** Set up a mechanism to update the robot's software remotely or without manual re-imaging. For a simple approach, you can do a local A/B switch: e.g., have systemA active, flash a new OS image to systemB partition (perhaps by streaming a pre-built image or using `rsync` to copy files), then instruct the bootloader to boot into B on next reboot. If that new image boots and runs well, you mark it good; if not, the bootloader will fall back to A. Test this by making a tiny modification (like a different LED behavior or version file) in the "new" image to confirm you're running the updated system, then simulate a failure (perhaps create a bad update that doesn't boot correctly) to ensure rollback works. This can be done manually at first. If using a tool like **Mender**, you could integrate their client which automates these steps and even provides a web UI for triggering OTA updates. According to Mender's documentation, **dual partitions** is a widely-adopted strategy for fail-safe updates [16] . 5. **Deployment Mindset:** As a final exercise, put it all together. Imagine this robot is deployed in the field: you should be able to (a) trust that it's running only your approved code (thanks to secure boot), (b) update it to a new software version with minimal downtime (A/B slots), and (c) recover it to a known good state if something goes wrong (rollback).

Document procedures for building a new release of your software and rolling it out. In practice, you might maintain two complete SD/NVMe images as your A and B, or use a build system to generate update packages. This discipline ensures the longevity of your project – you won't easily "brick" the robot with a bad update, and you minimize on-site debugging. - **Expected Outcome:** By completing this project, you will have moved from a "maker/hacker" setup to a **professional-grade embedded system** approach. Your robot's main software will be robust against failures and security threats. Specifically, you'll achieve: - **Reliability:** The robot can automatically rollback to a working software version if an update fails or is buggy [15], meaning it can recover from errors without manual intervention. - **Security:** With secure boot and immutable system design, it's much harder for malware or unauthorized changes to take root in the robot. This is important if the robot might be in sensitive environments or if you simply want peace of mind that "what I program is what's running". - **Maintainability:** You can push improvements or bug fixes to the robot over time in a controlled way. This is the basis for fleet management if you ever had multiple units.

You will have learned about bootloaders, disk partitioning schemes for OTA, and system security features – these are complex topics, but even a basic implementation will greatly enhance your skills. Treat this as a capstone where you document how to build and deploy a complete software image for your robot. In the end, you'll have a mobile autonomous companion that is not only powerful (thanks to the earlier projects) but also **reliable and upgradable** for the long term, ready to fulfill its mission of autonomous archiving.

Conclusion and Next Steps

By following this structured roadmap, you'll progressively build up the **Compute & Primary Storage** capabilities of the Project Lens robot. The journey started with choosing the right computing platform and ends with making that platform robust and secure. Along the way, you gained hands-on experience in diverse areas: Linux setup, hardware cooling, real-time systems, embedded microcontroller programming, storage management, and device security. Each mini-project provides foundational knowledge that will serve you in broader robotics development:

- After **Mini-Project 1–3**, you have a strong main computer running optimally (correct hardware, cooled, and tuned for real-time).
- With **Mini-Project 4**, you've bridged into low-level embedded control, essential for robotics where precise timing matters.
- **Mini-Project 5** ensures your robot can handle data-intensive tasks and store large logs without flinching.
- **Mini-Project 6** prepares your system for longevity and easy iteration, which is invaluable as you update your robot's software or scale up the project.

The recommended order is as listed, but some tasks can be done in parallel or iteratively. For instance, you might integrate the MCU (Project 4) while still refining real-time settings (Project 3). Just be mindful of dependencies (e.g., do the basic board setup before anything else, and set up the NVMe before trying the dual-boot image).

With the compute and storage platform solidified, you can then focus on other aspects of the Project Lens robot (sensing, locomotion, AI software, etc.) with confidence in your robot's "brain" and "memory". Good luck, and enjoy the process of building your autonomous archival companion!

Sources:

1. Stereolabs – *Getting Started with ROS on Jetson Nano* (Jetson Nano is designed for autonomous machines, enabling real-time vision and AI at the edge) [1] .
2. ThinkRobotics – *Raspberry Pi for Robotics* (Raspberry Pi's popularity is due to low cost and a vast community, making it ideal for beginners) [2] .
3. Vecros – *Jetson NX/Orin NX Heatsink & Fan Product Details* (Active cooling prevents thermal throttling, sustaining high performance for AI and robotics applications) [4] .
4. Universal Robots ROS2 Driver Docs – *Real-Time Scheduling* (Recommends a real-time kernel for smooth high-frequency control; non-RT systems may lead to non-smooth trajectories) [6] .
5. micro-ROS documentation – *Why Microcontrollers?* (Microcontrollers are used for direct hardware access and low-latency real-time tasks in almost every robotic product) [11] .
6. Storj community forum – *SD vs SSD endurance discussion* (SSDs have many more flash cells and better wear leveling, so **SD cards are always inferior** to SSDs in write endurance) [12] .
7. NVIDIA Isaac ROS docs – *Jetson Storage Setup* (Advises using an NVMe SSD for large, fast storage on Jetson; USB drives are slower than NVMe PCIe storage) [18] .
8. Ubuntu Robotics – *Robotics fleet management with Ubuntu* (Ubuntu provides OTA update infrastructure with failure rollback to maintain robot uptime and stability) [15] .
9. Mender.io – *OTA Updates with Dual Partitions* (Describes the widely-adopted A/B partition strategy to enable fail-safe updates and easy rollback in embedded Linux devices) [16] .

---

[1] Getting Started with ROS on Jetson Nano | Stereolabs
https://www.stereolabs.com/blog/ros-and-nvidia-jetson-nano

[2] [3] Raspberry Pi for Robotics- Creating Smart Robots on a Budget – ThinkRobotics.com
https://thinkrobotics.com/blogs/learn/raspberry-pi-for-robotics-creating-smart-robots-on-a-budget?srsltid=AfmBOoqtZwHl6a3DNNBrpOeLxBKGg2fRpU1E4iq62SzOqkilNB3b50Tv

[4] Heatsink & Fan for Jetson Xavier NX/Orin NX – vecros store
https://store.vecros.com/products/heatsink-fan-for-jetson-xavier-nx-orin-nx?srsltid=AfmBOop5vz3gTMU271uWWHDSNzNO6ZEnV9yM1UvE20Vt0p_okcPycbTV

[5] [6] Setup for real-time scheduling — Universal Robots ROS 2 Driver Documentation 0.1 documentation
https://docs.universal-robots.com/Universal_Robots_ROS2_Documentation/doc/ur_client_library/doc/real_time.html

[7] RTOS Support on Jetson Orin Nano
https://nvidia-jetson.piveral.com/jetson-orin-nano/rtos-support-on-jetson-orin-nano/

[8] An image for the Raspberry Pi 4 with ROS 2 and Linux RT preinstalled
https://github.com/ros-realtime/ros-realtime-rpi4-image

[9] RT Kernel on Jetson Nano - Simon Ghyselincks
https://chipnbits.github.io/content/projects/RLUnicycle/rtkernel/rtpatch.html

[10] Building a real-time kernel for the NVIDIA Jetson TK1 - JetsonHacks
https://jetsonhacks.com/2015/05/19/building-a-real-time-kernel-for-the-nvidia-jetson-tk1/

[11] micro-ROS | ROS 2 for microcontrollers
https://micro.ros.org/

12 SD vs SSD what the difference and why one cannot fully replace the other? - Storj Community Forum (official)
https://forum.storj.io/t/sd-vs-ssd-what-the-difference-and-why-one-cannot-fully-replace-the-other/10935

13 18 Jetson Setup — isaac_ros_docs documentation
https://nvidia-isaac-ros.github.io/getting_started/hardware_setup/compute/jetson_storage.html

14 How to flash and boot a Jetson from NVMe SSD
https://developer.ridgerun.com/wiki/index.php/How_to_flash_and_boot_a_Jetson_from_NVMe_SSD

15 Robotics | Ubuntu
https://ubuntu.com/robotics

16 Robust OTA updates with A/B Partitions for Linux devices
https://mender.io/blog/robust-ota-updates-with-partitions-for-linux-devices

17 Proper way of updating rootfs in a jetson nano with A/B redundancy enabled - Jetson Orin Nano - NVIDIA Developer Forums
https://forums.developer.nvidia.com/t/proper-way-of-updating-rootfs-in-a-jetson-nano-with-a-b-redundancy-enabled/316086