ChatGPT

Code Quality & Programming Standards for Humanoid Robot Development (ROS 2 Humble)

Introduction

Developing software for humanoid robots requires strict coding standards and best practices to ensure reliability, safety, and maintainability. This **codex** provides a comprehensive rulebook for junior developers working on ROS 2 Humble (Ubuntu 22.04) projects, focusing on code quality and programming standards. It covers guidelines for both **C++17** and **Python 3** (the primary languages in ROS 2 Humble [1] [2]), handling common robotics file formats (JSON, XML, URDF, SDF, Xacro), ROS 2 naming conventions, environment setup, build and launch procedures, secure coding practices, code organization, testing, code reviews, and continuous integration.

The goal is to establish a **foundational internal knowledge base** that junior roboticists can reference daily to write clean, robust, and idiomatic code for humanoid robots. Each section includes examples and authoritative references (especially from ROS 2 Humble and RViz2 documentation) to reinforce the guidelines. Adhering to these standards will lead to code that is easier to understand, extend, and debug – which is critical in complex humanoid robotics projects.

Programming Languages and File Formats

C++ Standards and Best Practices

ROS 2 Humble targets **C++17** and follows a coding style derived from the Google C++ Style Guide with ROS-specific modifications [3]. Key C++ guidelines include:

- **Code Style:** Use consistent formatting and naming. ROS 2 allows either CamelCase or snake_case for functions (Google style recommends CamelCase, but ROS 2 core uses snake_case) – **be consistent within a project** [4]. Class names should use **PascalCase** (CamelCase starting with capital) [5]. For constants, ROS 2 codebases historically use a mix of `ALL_CAPS`, `PascalCase`, or `snake_case` – follow the convention of the package you're working in [6]. In general, prioritize readability and match existing style in related ROS 2 packages [7]. Use `//` **for inline comments** and `///` or `/** ... */` for documentation comments (for Doxygen/Sphinx) [8] [9]. Always put braces around control blocks, even single-line `if/else` bodies, to prevent errors [10].

- **Memory Management:** Favor modern C++ practices for safety. Use RAII and smart pointers (`std::shared_ptr`, `std::unique_ptr`) instead of raw pointers where possible to avoid leaks and dangling pointers. Avoid manual memory management of ROS messages; use the provided message classes and let ROS middleware handle memory. Avoid copying large data unnecessarily – pass by reference or use move semantics if appropriate. When using pointers or resource handles, always ensure they are properly freed or closed (consider using destructors or smart pointers to automate this). These practices align with the **SEI CERT C++ secure coding guidelines** which emphasize proper memory management and pointer safety [11].

- **Error Handling:** Check the return values of all system calls and library functions (e.g., file I/O, memory allocations). **Defensive programming** is encouraged – validate assumptions and handle errors as early as possible [12]. Use exceptions in C++ for error conditions that are truly exceptional or unrecoverable in the current context. ROS 2 allows exceptions, especially for user-facing APIs, but avoid throwing exceptions from destructors and be cautious if you plan to wrap C++ APIs for use in C [13] [14]. For expected error cases, you can use error codes or `std::optional`/`std::expected` patterns; just ensure errors are not ignored. All error or warning messages should be directed to `stderr` (ROS 2 logging will handle this via logging macros) [15].

- **Concurrency:** When writing multi-threaded code (common in robotics for sensor processing, control loops, etc.), use thread-safe practices. Protect shared data with mutexes or use lock-free structures where appropriate. Leverage ROS 2's Executor and callback group mechanisms to manage callback concurrency instead of manual thread spawning when possible. Always avoid data races and deadlocks by design (e.g., prefer scoped locks, avoid holding locks during lengthy operations or while calling into user code).

- **Libraries and Dependencies:** Use the STL and ROS 2 provided libraries instead of reinventing functionality. Avoid using Boost unless absolutely necessary (ROS 2 prefers avoiding Boost for new code) [16]. For parsing and handling data formats (JSON, XML, etc.), prefer well-tested libraries (see *File Formats* below). Ensure any third-party dependencies are added to your package manifest and CMake appropriately, and try to minimize dependencies to reduce build complexity.

- **Compiler and Warnings:** Stick to C++17 standard as used by ROS 2 Humble [1]. Enable compiler warnings (and treat warnings as errors if possible) to catch issues early. The ROS 2 build system with `ament_cmake` can integrate linters like `ament_cpplint` and `ament_clang_tidy` – use these to enforce style and catch potential bugs. (More on linters in a later section.)

Python Standards and Best Practices

ROS 2 uses **Python 3** (with ROS 2 Humble, typically Python 3.10 on Ubuntu 22.04). For Python development:

- **Code Style:** Follow **PEP 8** style guidelines for Python code formatting [17]. Use `snake_case` for function and variable names, and `PascalCase` (CamelCase) for class names (e.g., `LegController`, `sensor_reader`) [17]. Constants can be `UPPER_CASE`. Keep line lengths reasonable (PEP 8 suggests 79 characters, but ROS 2 permits up to 100 characters in C++ [3], and similar leniency is fine in Python if it improves readability). Indentation should be 4 spaces (never tabs). Use docstrings for modules, classes, and functions to document their purpose and usage.

- **Idiomatic Python:** Write clear, Pythonic code. Prefer list comprehensions and generators for clarity and efficiency when appropriate, but not at the expense of readability for complex logic. Leverage Python's standard library modules (e.g., `json`, `logging`, `subprocess`) rather than writing custom code for those purposes. In ROS 2, use rclpy (ROS Client Library for Python) constructs like `Node` and `Logger` for node logic and logging (avoid using print statements for logging; use `self.get_logger().info()`, etc.). Manage resources with context managers (the `with` statement) to ensure proper cleanup (e.g., file I/O).

- **Performance Considerations:** Python is used for flexibility, but for performance-critical loops (like high-frequency control), prefer C++ or ensure the Python code is optimized (e.g., use NumPy for heavy math). Avoid unnecessary global variables; in ROS nodes, use class members or local variables. If interfacing with C++ libraries or hardware drivers, consider using Python bindings or C++ nodes for those parts to maintain performance. Always be mindful of the Global Interpreter Lock (GIL) in Python when using threads – for CPU-bound tasks, spawning multiple OS threads may not speed up execution; consider multiprocessing or moving to C++ in those cases.

- **Memory and Error Handling:** Python is garbage-collected, but circular references or unclosed resources (files, network sockets) can still cause issues. Always close files or use `with open(...) as f:` patterns. Catch exceptions that you expect (for example, a `ValueError` from bad input) and handle them, but don't blanket-catch all exceptions; let unexpected exceptions propagate (or crash) so they can be noticed and fixed. Make use of Python's warnings module to flag deprecations or usage issues without stopping execution when appropriate. Use type hints (PEP 484) for function signatures to improve code clarity and facilitate static analysis, but remember they are not enforced at runtime – they are mainly for developers and linters.

Handling JSON, XML, URDF, SDF, and Xacro

Humanoid robot software often interacts with various data file formats: configuration files (JSON, YAML), robot models (URDF/Xacro, SDF), and other XML-based formats. Adhere to the following practices:

- **JSON and YAML:** Use standard libraries to handle serialization and parsing. In Python, use the built-in `json` module for JSON (and `yaml` via `PyYAML` if needed for YAML). In C++, consider using established libraries like *nlohmann/json* for JSON or *libyaml* for YAML (or the ROS 2 `rcl_yaml_param_parser` for parameter files). **Never use ad-hoc string manipulation to parse JSON/XML**, as this is error-prone and insecure (e.g., risks like code injection). Validate JSON/YAML schema if possible – ensure the data has expected fields and types, and handle missing or extra fields gracefully (for instance, provide default values or warnings). When writing JSON or YAML configuration files (such as for robot calibration data, gait parameters, etc.), organize them logically and include comments (YAML supports comments, JSON does not – for JSON use documentation to explain fields). Keep these files in the `config/` directory of your ROS package for clarity [18].

- **URDF (Unified Robot Description Format):** URDF is an XML format for robot models. Keep URDF files **well-organized and modular**. Leverage **Xacro (XML macros)** to avoid duplication and to reuse snippets (Xacro is essentially a macro language on top of XML/URDF) [19]. For example, define a macro for a repeated structure like an arm link or finger and reuse it with different parameters for left/right sides. Split large robot descriptions into multiple Xacro files (for instance, one for the leg, one for the torso, one for sensors, etc.) and include them in a main file – this improves maintainability [19]. Each link and joint name in the URDF should follow a consistent naming convention. For humanoids, it's common to include side and joint names, e.g., `left_hip_joint`, `right_knee_joint`, `left_ankle_link`, etc., sometimes with suffix `_link` or `_joint` for clarity [20]. Adhere to ROS conventions or REP guidelines for humanoid joint naming when available (e.g., REP-120 for humanoid coordinate frames defines standard names like `base_link`, `l_wrist`, `r_ankle`, etc. [21] [22]). This consistency makes it easier for others to understand the robot model and integrate with common tools.

- **SDF (Simulation Description Format):** SDF is often used by Gazebo for simulation. If your project uses Gazebo (Ignition Gazebo or classic Gazebo), you may need SDF. You can often generate an SDF from URDF, but for complex simulation-specific elements (like physics parameters, friction, plugins), you might maintain a separate SDF or Xacro that outputs SDF. Keep simulation-specific content (Gazebo plugins, transmission tags, etc.) separated from the core URDF describing the robot's kinematics/visuals. One approach is to maintain a primary URDF/Xacro for the robot, and a separate Gazebo-specific Xacro that includes the core Xacro and adds simulation elements (this can be conditionally included via Xacro properties). This way, the real robot description is not tangled with simulation details. **Keep model files under version control** and review changes carefully, as errors in inertial values or joint orientations can be hard to debug. Use tools like `check_urdf` or `ign sdf -k` (for SDF) to validate the files.

- **XML and Other Formats:** For generic XML processing (e.g., reading configuration or launch XML files if any), use robust parsers (Python's `xml.etree.ElementTree` or C++ TinyXML2, which is used by ROS for URDF parsing). Always handle parse errors (malformed XML) and validate expected structure. For example, if loading an XML that contains waypoints or behavior scripts, ensure required tags exist and report meaningful errors if not. When writing XML (or generating URDF programmatically), ensure proper escaping of special characters and test that the output can be parsed by the intended consumer (e.g., robot_state_publisher or Gazebo).

- **File Organization:** Organize files by type in your ROS package. As mentioned, use a `config/` **folder** for configuration files (YAML/JSON/etc.), a `urdf/` **or** `models/` **folder** for URDF/Xacro and meshes, and possibly a `launch/` **folder** for any standalone launch files related to model visualization or testing [23]. For example, in a `my_robot_description` package, you might have `urdf/humanoid.urdf.xacro`, `urdf/head.xacro`, `meshes/head.dae`, and a `launch/display.launch.py` that loads the URDF into ROS and launches RViz [24]. Keeping these structured makes it easy to find and update the robot model and associated resources.

Naming Conventions and Casing

Consistent naming is crucial for readability and to avoid confusion in a large codebase or robot system. Below are naming conventions to follow for code symbols and ROS resources:

- **Packages:** ROS packages names are all **lowercase**, starting with a letter, and words separated by underscores [25]. No capital letters or hyphens should be used (e.g., `walking_controller`, not `WalkingController` or `walking-controller`). Package names should be descriptive of their content or purpose [26]. For instance, a package for inverse kinematics might be `humanoid_ik`, and one for sensor processing might be `lidar_processing`. Avoid overly generic names like `utils` – be specific (e.g., `robot_utils` if it's truly a grab-bag of small utilities) [27]. If your project is specific to a particular robot, using a prefix is common (e.g., `atlas_control`, `atlas_description` for Atlas robot) [28] [29]. However, do not prefix with `ros` (redundant) [30]. Check ROS package index to ensure your package name isn't already used [31].

- **ROS Nodes:** Node names (the name a node uses at runtime) should also be lowercase with underscores if needed. By convention, node names often match the primary purpose of the node (e.g., `state_estimator`, `footstep_planner`). Node names can be hierarchical with namespaces (e.g., `perception/camera_driver` could be a node in the `perception` namespace), but avoid deep or overly broad namespaces. In ROS 2, node name rules are similar to topic names: alphanumeric and underscores are allowed, as well as forward slashes for namespaces; they must not start with a number or end with a slash [32]. Use **meaningful names** that reflect the function, as this helps when using tools like `ros2 node list` and `ros2 topic list` to understand the running system.

- **Topics and Services:** Topic and service names in ROS 2 should be **lowercase with underscores** (no CamelCase) [33]. This follows ROS 1 tradition for readability (e.g., `/left_arm/torque_command`) and avoids confusion in DDS, which is case-sensitive. They may contain only alphanumeric characters and underscores, and forward slashes for namespaces [32]. Do not use `__` (double underscore) in names, as it has special meaning in ROS names (ROS 1 used `__` for private names; ROS 2 still reserves double underscores in some contexts) [34]. Also avoid trailing slashes or any whitespace. A good convention is to **name topics by the data they carry**, e.g., `"/camera/left/image_raw"` for a raw image topic, `"/robot/pose"` for a robot pose. Use plural nouns for streams of data (topics) and verb phrases for services/actions (e.g., `"/plan_footsteps"` could be an action or service, whereas `"/footstep_plan"` might be a topic publishing a continuous plan). If a topic is only relevant within a node (i.e., not intended for public interface), consider using a private namespace (in ROS 2 you can use `~` in code which expands to the node's namespace) to prevent name collisions.

- **Parameters:** ROS 2 parameters are namespaced within nodes. Parameter names should be lower_case (and typically use underscores). If you have structured parameters, ROS 2 supports hierarchy by dots in names (for example `camera.calibration.focal_length`) [35] [36]. Use this namespacing to logically group parameters. For example, a humanoid's control node might have `gains.ankle_p` and `gains.ankle_d` under an overall namespace for gains. Avoid very long parameter names and keep them intuitive. Also, **document each parameter** (via YAML comments or in code) so others know what it affects. For constant configuration values that rarely change, parameters are preferred over hardcoding, as they can be tuned without recompiling. Parameter files (YAML format) should be placed in the `config/` directory of the package and can be loaded at runtime (see *Launch Files* below).

- **C++ Variables and Functions:** In C++ ROS code, follow a consistent casing style. As mentioned, ROS 2 core uses snake_case for functions and variables [4]. For example, a C++ class method might be `compute_kinematics()` and a variable `target_position`. Stick to **snake_case for non-class names** to match the ROS client library style (rclcpp itself uses snake_case for methods like `create_publisher`). If you choose to use CamelCase for functions (which Google style suggests), do so consistently and be aware that most ROS examples use snake_case. **Class member variables** can optionally use a prefix to distinguish them (some styles use `m_` or `_` prefix, e.g., `_is_enabled` for a private member). If using such prefixes, mention it in your style guide for the project – the key is consistency. Template parameters and macros are typically `CamelCase` or `ALL_CAPS` respectively by convention.

- **Python Variables and Functions:** Python should strictly follow **snake_case** for variables and function names (per PEP 8). For example, `def calculate_trajectory():` and `left_arm_joint_positions = [...]`. Instance attributes should be snake_case (`self.current_state`). Constants (module-level) can be `UPPER_CASE` (e.g., a constant `MAX_SPEED = 1.5`). Class names in Python use CamelCase (e.g., `HumanoidController`). One common ROS pattern: if a Python file is intended to be an executable node, it may use a compound name with underscores (e.g., `teleop_keyboard.py` containing class `TeleopKeyboard`). Filenames for scripts should also be lowercase (they can use underscores to separate words).

- **Message, Service, Action Names:** When defining custom message types, the message files (in the `msg/` directory) should use **CamelCase** with each word capitalized (e.g., `WalkingCommand.msg`, `JointAngles.msg`). The ROS interface naming conventions often use CamelCase for the message type names (but the fields within are lowercase with underscores). For message fields, use lower_case with underscores (e.g., `float64 stride_length` inside `WalkingCommand.msg`). Service (.srv) and action (.action) files follow similar patterns. Keep names descriptive (a service could be `ResetOdometry.srv`). If a message is specific to your robot, it might be prefixed by the robot or package name (though since they live in your package's namespace, this is optional).

By following these naming conventions, new team members can easily understand what an identifier represents, and tools can more easily process or display names (for example, ros2 topic/param CLI tools list names in consistent formats). Remember that **consistency beats individual preference** – it's more important to stick to one scheme across the project than whether you choose camelCase vs snake_case in a given situation [37] [5]. When in doubt, mirror the naming style of well-known ROS 2 packages or examples for similar elements.

Environment and Workspace Setup (Virtual Envs & Colcon)

A proper development environment ensures reproducibility and isolates project-specific dependencies. In ROS 2, the typical workspace and environment setup includes using **colcon** for building and possibly Python virtual environments for Python dependencies. Here are the best practices:

- **ROS 2 Workspaces:** Develop in an isolated ROS 2 workspace (typically a directory with a `src/` subdirectory). For example, create `~/humanoid_ws/src/` and put all your packages in `src`. Use `colcon build` to compile the workspace. Keep your ROS 2 installation (`/opt/ros/humble`) separate from your workspace – do not mix your own packages into the ROS system install. This separation makes it easy to clean, rebuild, or deploy your code. For source control, typically you'll include the package source directories and maybe a top-level `COLCON_IGNORE` or `README`, but not the build/install logs.

- **Colcon Build Tool: Colcon** is the standard build tool for ROS 2. Always source the ROS 2 setup script and your workspace's install script when building or running. For example, a typical build might be:

```
$ source /opt/ros/humble/setup.bash
$ colcon build --symlink-install
```

The `--symlink-install` is useful during development (it symlinks Python packages rather than copying them, so edits reflect without rebuild). Organize your `CMakeLists.txt` or `setup.py` in packages properly so that colcon can build them in the correct order (use `ament_target_dependencies` for C++ to link against other packages, declare `<exec_depend>` and `<build_depend>` in package.xml, etc.). If you have multiple independent parts (e.g., simulation vs real robot drivers), you can use colcon mixins or profiles, but that's advanced. For juniors, the main point is: use colcon as the single entry point to build, test, and even lint (colcon can run tests and linters via verbs). Do not manually invoke `cmake` or `make` on ROS 2 packages – let colcon handle it, as it sets up environment variables and parallelizes builds.

- **Python Virtual Environments:** When your ROS 2 Python packages require additional pip packages (e.g., `numpy`, `opencv-python`, or custom libraries), it's best to use a **Python virtual environment** to avoid cluttering the system Python or conflicting versions. Create a virtual env inside your workspace (but exclude it from colcon builds) [38]. For example:

```
$ cd ~/humanoid_ws
$ python3 -m venv venv      # create virtual environment
$ source venv/bin/activate
$ touch venv/COLCON_IGNORE  # ensure colcon ignores the venv folder [39]
$ pip install numpy opencv-python
```

Activating the venv will isolate the Python interpreter to use those dependencies. You should activate the venv *in addition to* sourcing ROS when working with Python nodes. A common practice is to extend your workspace's local setup script to also activate the venv for convenience. Alternatively, you can containerize the development environment with Docker, but that adds complexity for beginners. The key is to avoid installing lots of pip packages globally; use `venv` or conda environments so that requirements can be maintained per project. Document the needed pip packages (e.g., provide a `requirements.txt`). Note that when running `colcon build`, the environment at build time (including activated venv) will be baked into scripts. The ROS 2 docs provide guidance on using virtual environments with colcon [40] [41]. When releasing packages (bloom releases), pip deps should be in `package.xml` as `<exec_depend>` and in rosdep if available – a virtualenv is mostly for development workflow.

- **System Isolation:** If working on multiple ROS 2 projects or versions, consider using tools like `vcstool` to manage workspace source, and separate workspaces for different projects or ROS distributions (e.g., one workspace for Humble, one for Rolling). Always double-check that you've sourced the correct ROS distribution's setup before building or running (this is a common hiccup for beginners). ROS 2's `ros2 doctor` tool (ros2doctor) can help diagnose environment issues.

- **Environment Variables:** Certain environment variables are important in ROS 2 – for example, `ROS_DOMAIN_ID` if you need to isolate DDS communication, or `RMW_IMPLEMENTATION` if choosing a specific middleware. By default, you might not need to change these. But if you have multiple robots on the same network, ensure they have different domain IDs to avoid cross-talk. You can set such environment variables in your launch files or in a bash script when starting your system. Avoid setting global environment variables in your code; instead, document them and set them externally or via launch configurations.

In summary, maintain a clean, isolated workspace, use virtual environments for Python, and leverage colcon for all build/test tasks. This setup will make sure your development environment is reproducible and consistent across the team.

Building, Launching, and Running Nodes (with Bash & Launch Files)

Developers often need to automate common actions like building packages, launching multiple nodes, or visualizing the robot model. ROS 2 provides the **ros2 CLI** and **launch files** for these tasks, but sometimes custom bash scripts are useful for project-specific workflows. Here's how to handle these tasks in ROS 2 Humble:

- **Build Scripts:** While you can build with `colcon build` manually, it's helpful to have a script (e.g., `build.bash`) at the root of your workspace that sources the environment and runs colcon with your preferred options. For instance, a `build.bash` might contain:

```
#!/bin/bash
source /opt/ros/humble/setup.bash
source ~/humanoid_ws/install/setup.bash   # if re-building and you want overlay
colcon build --symlink-install --event-handlers console_cohesion+ --parallel-
workers 4
```

This ensures the ROS environment is loaded and uses some colcon options (like nicer console output grouping and limiting parallel jobs). Always include error handling in scripts: e.g., `set -e` to stop on the first error, so you don't miss a failed build. If your project has multiple build profiles (debug vs release), you can incorporate colcon mixins or pass `--cmake-args -DCMAKE_BUILD_TYPE=Release`. A script makes it easier for everyone to build in a consistent way (and CI can also invoke it).

- **Launch Files (ROS 2 launch system):** ROS 2 uses Python-based launch files (`.launch.py`) which are very powerful. Use launch files to start up multiple nodes, set configurations, and even bring up RViz or Gazebo. For example, you might have a `bringup.launch.py` that launches all the core nodes of your humanoid robot (controllers, sensors, state estimation, etc.) with proper namespaces and parameters. Launch files allow you to declare launch arguments for flexibility (e.g., `use_sim_time` or `robot_description_file`). Write launch files in the `launch/` directory of a package (commonly a dedicated *bringup* package, e.g., `my_robot_bringup`) [42] [43]. Keeping all runtime launch files in one package is recommended to avoid scattering launch logic [44] [42]. This *bringup* package can depend on all others and serve as the entry point to run the robot's software. Each launch file should be well-documented: clearly state which nodes it starts and which parameters or arguments it accepts.

Example: A `visualize.launch.py` in `my_robot_description` package might: declare an argument for which URDF/Xacro to use, run the `xacro` command to generate URDF, set the `robot_description` parameter, start `robot_state_publisher` with that URDF, and launch RViz pointing to a given RViz config file. This is a common pattern to visualize a URDF in RViz2. In ROS 2 Humble, you might do something like:

```python
# Inside visualize.launch.py
from launch import LaunchDescription
from launch_ros.actions import Node
from ament_index_python.packages import get_package_share_path

def generate_launch_description():
    pkg_path = get_package_share_path('my_robot_description')
    urdf_file = str(pkg_path / 'urdf/humanoid.urdf.xacro')
    rviz_config = str(pkg_path / 'rviz/model.rviz')
    # Node to run xacro and publish robot state
    robot_state_pub = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        name='robot_state_publisher',
        output='screen',
        parameters=[{'robot_description': Command(['xacro ', urdf_file])}]
    )
    rviz = Node(
        package='rviz2', executable='rviz2', name='rviz2',
        arguments=['-d', rviz_config]
    )
    return LaunchDescription([robot_state_pub, rviz])
```

This example uses `xacro` via a `Command` substitution to generate the URDF on the fly. After launching such a file (`ros2 launch my_robot_description visualize.launch.py`), you should see RViz open with your robot model. The ROS 2 tutorial demonstrates how `display.launch.py` is used to visualize a URDF, by spawning the necessary TF frames and RobotModel display in RViz [45] [46]. Launch files support conditions, environment variables, and many other features – use them to simplify complex startup logic rather than writing long bash scripts that run multiple `ros2 run` commands.

- **Bash Scripting for Running Nodes:** In some cases, simple bash scripts are still useful (for instance, to quickly test something or to integrate with system startup). If you write a bash script to launch nodes manually, remember to `source /opt/ros/humble/setup.bash` and your workspace's `setup.bash` at the top of the script, so that the ROS 2 CLI and libraries are in PATH. Use `ros2 run <package> <executable>` to start single nodes from bash. For launching multiple nodes without a launch file, you could open multiple terminals or background processes, but it's better to use a launch file or a process manager (like `tmux` or `ros2 launch`). For example, a quick test script might look like:

```bash
#!/bin/bash
source /opt/ros/humble/setup.bash
source ~/humanoid_ws/install/setup.bash
set -e  # exit on first error
ros2 run humanoid_control balance_controller &
```

```
ros2 run humanoid_sensors sensor_bridge &
wait
```

This would run two nodes in parallel and wait for them to finish. But note, using `&` like this means you won't easily capture if one fails. A more robust approach is to use a launch file or a supervisor that restarts nodes if they crash. For development or debugging, launching from bash is fine; for regular operation, prefer launch files (or for very simple cases, a single node can be launched with `ros2 run` directly with no script).

- **Launching RViz2 and Other Tools:** RViz2 is a crucial tool for visualization. You can launch RViz2 with a config file via CLI or launch file. For instance: `rviz2 -d src/my_robot_description/rviz/model.rviz`. Typically, you'd include RViz in a launch file when needed. Similarly, Gazebo (if used) can be launched via `ros2 launch gazebo_ros gazebo.launch.py` or a custom launch. Provide run scripts or launch files for RViz and simulation so that junior developers can easily visualize data. One could have a script `run_rviz.sh` that simply sources ROS and runs the rviz command with the config.

- **Version-Specific Notes (ROS 2 Humble):** ROS 2 Humble's launch system is quite stable. Make sure to use the correct syntax (it evolved from ROS 2 Foxy). For example, `DeclareLaunchArgument` and how substitutions like `Command()` (to call xacro) work might differ from older versions. The above examples are Humble-compatible. Another Humble-specific: RViz2 in Humble uses Ogre 3D for rendering; there's no major user-facing change, but just ensure configs are compatible (if using older .rviz files, they usually are fine). When writing bash scripts, note that ROS 2 Humble might install executables to `/opt/ros/humble/bin`, which is added by the setup.bash – so don't assume ros2 is on PATH without sourcing.

In summary, use **ROS 2 launch files** for complex startup logic (setting parameters, remappings, multiple nodes, including RViz). Use **bash scripts** for simpler or development tasks, always sourcing the environment. Keep these scripts in version control (perhaps in a `scripts/` directory or within the `README` documentation) so everyone uses the same commands. This consistency ensures that launching the humanoid robot's software is straightforward and repeatable.

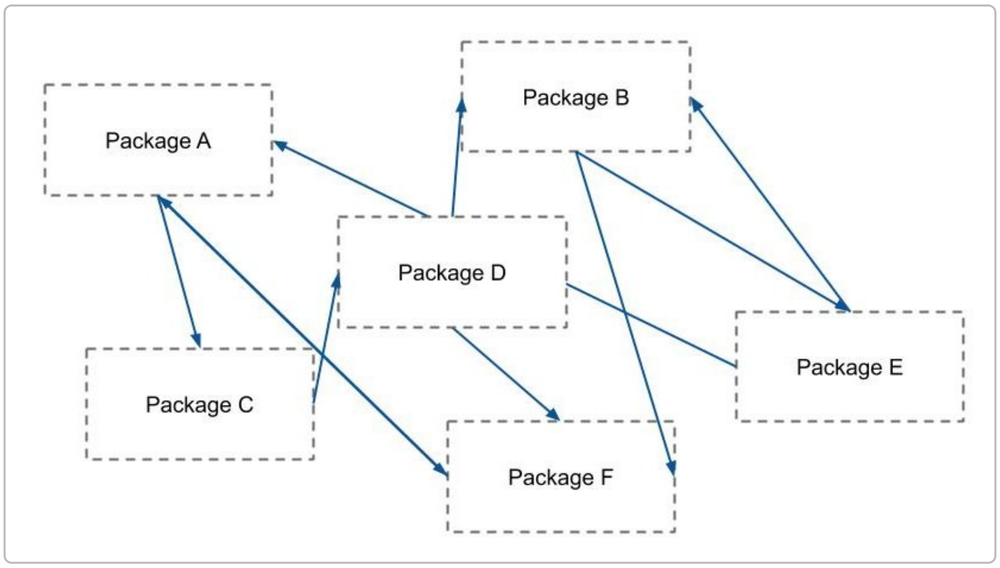

Illustration: An unclear and messy package/node dependency graph. Avoid complex, tightly-coupled startup scripts or interdependent packages that create such tangled relationships. Instead, structure launch files and packages with clear responsibilities (as in a clean graph with one-way dependencies) [47] [48] .

Secure Coding Practices

Security is increasingly important even in robotics. Humanoid robots often interact with networks, accept external inputs, or even interface with web services (for updates or telemetry). Writing secure code helps prevent accidents (safety) and malicious exploits. Here are key secure coding practices, with references to standards like **SEI CERT** and **OWASP**:

- **Validate All Inputs:** Any data coming from outside your code (sensor readings, user commands, configuration files, network messages) should be treated as potentially invalid or malicious. Check ranges, formats, and types. For example, if a JSON config is supposed to contain a list of joint angles, verify each value is within plausible limits before using it. In C/C++, use functions like `std::stoi` with try/catch instead of `atoi` (which doesn't report errors well), and always check the success of strtoX functions. In Python, use try/except around conversions or file parsing. Never blindly trust file contents or network data to, say, execute system commands. This aligns with OWASP principles of **input validation** and guarding against injection attacks (even though robotics systems aren't typical web apps, similar principles apply – e.g., a malformed config that breaks your system is akin to an injection) [49] .

- **Avoid Dangerous Functions and Patterns:** In C, avoid unsafe functions like `gets()` (which is removed in C11) or `strcpy` / `sprintf` without size checks – use `strncpy` , `snprintf` , or safer abstractions (like `std::string` in C++). In C++, avoid `new[]` / `delete[]` for dynamic arrays; use `std::vector` which handles bounds and memory automatically. Enable warnings or tools to catch common pitfalls (buffer overflows, off-by-one errors). The CERT C++ guidelines highlight issues like uninitialized variables, risky type casts, and unchecked array indices [11] . Use static analysis tools (many integrate CERT rules) to catch these. Also, **initialize all variables** (CERT MSC00 rule) and zeroout structures if passing to lower-level APIs. For pointers, after deleting (if you must delete manually), set them to nullptr to avoid dangling usage.

- **Memory Safety and Error Handling (CERT C++):** As noted, memory leaks or misuse can not only crash the program but also lead to undefined behavior which could be a security issue. Use tools like Valgrind to test for leaks or invalid accesses. Follow the rule of 3/5 (or use smart pointers) to manage copy and destruction. Catch exceptions at an appropriate level; unhandled exceptions will terminate the node – which might be fine for certain failures (better to restart than to continue in a bad state), but consider using ROS 2's lifecycle states or error recovery logic for critical components. Do not ignore returned error codes from functions; handle them or at least log them. For instance, if a hardware driver returns an error for a motor command, don't just silently continue – log it and take appropriate action (maybe stop the robot or retry). These practices echo **SEI CERT's secure coding rules** aimed at reliability and security [50].

- **Use Principle of Least Privilege:** If your robot's software runs on a computer that interacts with a network, ensure the process doesn't have more privileges than needed. This might be more of a deployment concern (run as a non-root user, use AppArmor or Seccomp profiles if appropriate). Within the code, principle of least privilege means don't grant write access or high authority to parts of the system that don't need it. For example, a node responsible for logging should not be issuing motor commands. Compartmentalize capabilities (this is often naturally done by separate ROS nodes/topics with specific scopes).

- **Protect Sensitive Information:** If your code deals with credentials (e.g., Wi-Fi passwords, cloud API keys for analytics), never hard-code them in the source. Use configuration files or environment variables, and secure those appropriately. If using ROS 2 Security (SROS2) with enclaves and certificates, treat keys and certificates carefully (don't commit them to public repos, follow ROS 2 security tutorials to generate and deploy them securely). Use ROS 2 security features if your robot operates in untrusted networks – SROS2 enables encryption and authentication of ROS topics/services.

- **Follow OWASP Secure Coding Practices:** OWASP's guide is more web-focused, but many principles are universal [51] [52]. For instance, **logging and monitoring** – ensure your system logs important events (and to a secure location) because if something goes wrong, you need to trace it. Use ROS logging appropriately (info, warn, error, debug levels) and avoid logging sensitive data. Another example is **asserting invariants**: use assertions in C++ (in debug builds) or simple `if` checks in runtime to catch impossible states. It's better to fail early than to continue in a corrupted state that could be dangerous for a robot. For web interfaces or GUIs that control the robot, use secure coding as if it's any web app (validate requests, avoid eval-ing user input, etc.). If the robot runs a REST server or connects to one, be aware of injection or buffer issues at those boundaries.

- **Concurrency and Security:** Robotics software is often concurrent (multi-thread or multi-process). Race conditions can become security issues (for example, a race could allow an operation without proper check if two threads interleave incorrectly). Always guard shared resources with locks or use lock-free data structures carefully. Deadlocks can cause safety issues (robot stops updating sensors, etc.), so design thread interactions minimalistically. Use condition variables or ROS 2's callback groups to manage concurrency deterministically.

- **Regular Updates and Patches:** Keep dependencies up to date. Use `rosdep` to manage system dependencies, and regularly update your ROS 2 installation to pick up security fixes in DDS or ROS libraries. If a CVE (Common Vulnerabilities and Exposures) is announced for a library you use (e.g., OpenCV), update to a patched version promptly. For custom code, if any security issue is found (e.g., by static analysis or pen-testing), fix it across the codebase (don't just patch one occurrence if the pattern might exist elsewhere).

- **Education and Tools:** Encourage team members to learn basics of secure coding. Utilize tools: static analyzers like **clang-tidy**, **cppcheck**, or **sonarqube** can catch issues. Dynamic analysis tools (AddressSanitizer, MSan, etc.) can detect memory misuse. For Python, linters like **pylint** or **bandit** (a security linter) can find potential issues (Bandit, for example, will warn if you use `yaml.load` unsafely or use `eval()` on untrusted input). Integrate these in CI so that insecure code doesn't slip in unnoticed.

In essence, **secure coding for robotics = safe coding**: by preventing buffer overruns, null pointer dereferences, race conditions, and validating inputs, you not only secure the system against attacks but also against bugs that could cause unsafe robot behavior. Adopting standards like SEI CERT C/C++ [53] [54] and referencing OWASP guidelines [55] provides a framework to achieve this. Always consider the worst-case scenarios and code defensively to handle them gracefully.

Code Structure and Modularization Best Practices

Organizing code and packages in a modular way is vital for managing a humanoid robot's software complexity. A well-structured codebase allows multiple developers to work in parallel and fosters reuse. Here are the best practices for code organization in ROS 2 humanoid projects:

- **One Package, One Responsibility:** Each ROS package should be an independent unit with a clear purpose [48]. Do not cram unrelated functionalities into a single package. For example, create separate packages for **control**, **perception**, **navigation**, **teleoperation**, **description (URDF)**, etc., rather than one giant package that does everything. This modular approach improves maintainability and readability [56] [57]. If your package is doing too many things, consider splitting it. A good heuristic: if you cannot describe what a package does in one phrase, it probably contains multiple responsibilities.

- **Minimize Inter-package Coupling:** Design package dependencies to be acyclic and minimal [58]. Circular dependencies (Package A depends on B and B on A) are a nightmare – avoid them by refactoring common parts into a third package or rethinking responsibilities. Use messages or services/actions to decouple runtime interaction rather than making packages depend on each other's libraries whenever possible. For instance, your *perception* package might publish processed sensor data, and your *planning* package subscribes to it – there's no need for *planning* to depend on *perception* at build time if they only communicate via ROS messages. This way, you could even replace the perception package with another implementation as long as it sends the same message types. The ROS paradigm encourages such loose coupling via interfaces. In practice, of course some packages will depend on others (e.g., a *bringup* package depends on everything to launch it, or a *teleop* node might depend on *control_msgs* defined in a *msgs* package). Just keep the dependency graph as sparse as possible and **layered** (like a stack of layers: drivers at bottom, control, planning, then higher-level behavior).

- **Core vs Specialized Packages:** Identify core packages that nearly every robot has: for example, a `my_robot_msgs` package for custom message definitions, a `my_robot_description` for URDF/Xacro, `my_robot_bringup` for launch files, `my_robot_control` for control algorithms, `my_robot_driver` for hardware interface [59] [43]. These are commonly seen patterns [60]. Build these first, then add specialized packages as needed (e.g., `my_robot_vision` for vision processing, which depends on some vision library, etc.). Keeping messages in their own package is especially helpful – it allows other packages to use those interfaces without creating circular deps (all can just depend on the msgs package) [61] [62]. The description package containing URDF and meshes should be separate so that both simulation and real robot bringup can use it without depending on each other.

- **Folder Structure within Packages:** Follow the standard ROS package structure for consistency [63] [64]. This means: put C++ source in `src/`, headers in `include/<package_name>/`, Python code in a `<package_name>/` directory (with `__init__.py`), launch files in `launch/`, config files (YAML, etc.) in `config/`, and test code/data in `test/`. ROS 2's developer guide explicitly lists this layout [63]. For example, a `humanoid_control` package might look like:

```
humanoid_control/
├── package.xml
├── CMakeLists.txt
├── include/humanoid_control/  (C++ headers)
├── src/                       (C++ source files)
├── humanoid_control/          (Python module, if any Python code)
├── launch/                    (launch files)
├── config/                    (yaml config files for controllers)
├── test/                      (unit test files, maybe using GTest or pytest)
└── README.md                  (optional documentation)
```

This consistency helps new developers find things quickly (e.g., they know where to look for configs or launch files) [18]. Moreover, some tooling expects this (ament build will install things from certain dirs by default if you call the right CMake macros, e.g., `install(DIRECTORY launch DESTINATION share/<pkg>)`).

- **Code Organization within a Package:** Structure your code into logical classes or modules. For instance, in a control package, you might have classes like `BalanceController`, `ArmController`, etc., and perhaps a main node class that uses them. Avoid extremely large source files – if a single file exceeds a few hundred lines, consider splitting it if it naturally breaks into separate components. Each class or module should have a clear responsibility (e.g., one class handles sensor fusion, another handles publishing TF frames). Use namespaces in C++ (e.g., `humanoid_control::controllers::Balance`) to avoid name collisions and logically group code. In Python, use modules and packages (the directory structure) to organize – e.g., a `humanoid_control` Python module could have submodules for different parts. This modularization inside the package makes unit testing easier too (you can test classes in isolation).

- **Reusability and Libraries:** If some code is reusable across packages (say a math utility or a common sensor processing routine), you have options: (1) Make it a small library within a package that others can link to (using CMake `ament_export_libraries`). (2) Factor it into a separate common package (e.g., `humanoid_utils`). The second approach increases modularity but too many tiny packages can be overhead. A good middle ground is to use a `common` or `utils` namespace or directory within a package for things only used by that package. For cross-package utilities, a separate package is cleaner. For example, if both vision and control need a certain kinematics function, you could have a `humanoid_kinematics` package that both depend on. Strive to **avoid code duplication** – it's better to refactor into a shared module than to copy-paste code into two places, as duplicates tend to diverge or double the maintenance work.

- **Launch and Configuration Separation:** We touched on having a dedicated *bringup* package. This package can serve as the integration point of all others, containing high-level launch files that pull together nodes from various packages. This way, individual packages (like control or vision) can be launched on their own for testing, but the bringup handles full robot startup. It's also a convenient place to put overarching config, like a top-level YAML that includes configs from sub-systems or sets global parameters (e.g., a global `use_sim_time` flag). Keeping launch files in one place means changes to startup (like adding a new node) are done in one package rather than scattered. The *bringup* package may have a structure like: `launch/robot_bringup.launch.py`, `config/controllers.yaml`, etc., combining pieces.

- **Example Modular Breakdown:** As an example, imagine a humanoid robot project layout:

- `humanoid_description` – contains URDF/Xacro, meshes, RViz config.
- `humanoid_msgs` – custom messages (e.g., WalkingCommand, JointStates if not using std_msgs).
- `humanoid_control` – balance and posture controllers (depends on msgs, description for model if needed).
- `humanoid_navigation` – SLAM or localization, path planning.
- `humanoid_perception` – vision or lidar processing nodes.
- `humanoid_bringup` – launch files that bring everything together, plus maybe default parameters.
- `humanoid_teleop` – (optional) for teleoperation interfaces.
- `humanoid_simulation` – (optional) if you have simulation-specific nodes or configurations (could also be integrated into bringup with a flag).

Each of these can be developed and tested somewhat independently. New features or hardware can be added by adding or swapping out a package (e.g., a new `humanoid_manipulation` package for arm control). This modular approach also aligns with ROS's philosophy of composing systems from nodes and packages.

- **Documentation and Readability:** Maintain a high-level documentation (could be in a markdown or on a wiki) that outlines the package architecture and node graph of the system. This isn't code structure per se, but it complements it by giving newcomers the map of how things are organized. Within code, use comments and descriptive names to make structure apparent (e.g., if you break a class into helper functions, consider grouping them in the file and commenting "// Kinematics helpers" etc.).

By following these structural best practices, you achieve a **scalable architecture**: new team members can work on separate packages without stepping on each other's toes, you can replace modules (e.g., try a new perception algorithm) with minimal impact on others, and debugging is easier because you know which part of the code to look at for a given issue. A clean, modular code structure is depicted by a directed acyclic graph of packages (and nodes) with clear data flows, instead of a tangled web. Aim for the "clear and clean" organization [47] [65] and avoid the "unclear and messy" big-ball-of-mud architecture [66].

Testing Strategies (Unit, Integration, E2E)

Testing is critical to maintain code quality and ensure that changes don't break existing functionality. In robotics, testing can be challenging due to hardware and timing, but ROS 2 provides a framework for both offline and simulation tests. We recommend adopting a **test pyramid** approach: many fast unit tests, fewer integration tests, and a few end-to-end tests (perhaps in simulation) [67] [68]. Here's how to implement various test levels:

- **Unit Tests:** These test individual components in isolation. For C++ code, use **Google Test (GTest)**, which is integrated with ROS 2's build system (you can add GTests in CMake with `ament_add_gtest()` and link against your library). For Python, use **pytest** or the built-in `unittest` module (ROS 2's `ament_pytest` integrates pytest nicely; just create tests in the `test/` folder and name them with `test_*.py`). Unit tests should mock or stub out any ROS dependencies – for instance, test a control algorithm class by feeding it fake sensor data, rather than requiring a live ROS topic. This keeps tests fast and deterministic. Aim to cover edge cases and typical cases. Every package should have some unit tests [69] [70]. For example, a `kinematics` function should be tested with known inputs and expected outputs (you might compute expected results offline). If a bug is found, write a unit test exposing it and then fix it – this prevents regressions.

Organize unit tests alongside code logically (maybe mirror the source directory structure under `test/`). Use ASSERTs/EXPECTs (in GTest) to verify conditions. For Python, use assertions or pytest's features. ROS 2's documentation has examples: *Writing Basic Tests with GTest* [71] and *with Python* [72]. To run tests, you can use `colcon test` which will invoke them (and `colcon test-result` to summarize results). Integrate tests into your CI pipeline so that merges require tests to pass.

- **Integration Tests:** These test interaction between multiple components or with some ROS framework elements. ROS 2 provides **launch_testing** to launch multiple nodes in a test and then run assertions, even allowing you to simulate temporal sequences (e.g., send a message and expect a response). For example, you can write a launch test that starts a talker and listener node and verifies that messages flow between them. In a humanoid context, an integration test might bring up a controller node and a dummy hardware interface node that responds to commands, then check that the controller sends appropriate commands when given certain sensor inputs. Another integration test might launch the entire perception pipeline (with recorded rosbag data) and verify it produces a detection topic within X seconds. ROS 2 launch_testing uses Python's unittest under the hood and is quite powerful [72]. Use integration tests to cover things like *do nodes connect properly? Does parameter loading from YAML actually set the right values in the node? Does a sequence of actions produce the expected outcome on a topic or service?* Because these tests involve multiple processes or threads, they are slower – you won't have as many integration tests as unit tests. But they add confidence that the pieces work together. Keep integration tests deterministic: control randomness (set seeds if using random inputs) and give time allowances for things to happen. ROS 2's launch_testing allows you to set timeouts for waiting on conditions.

Integration tests can also be structured to run with or without actual hardware. For hardware, you might use a simulator or a stub. For example, test your arm planner with a stub execution node that pretends to move the arm instantly to any commanded position and reports success – then check that the planner transitions to "goal reached" state. The ROS 2 Developer Guide suggests keeping integration tests in the same package as the code under test (to avoid cross-package dependencies just for tests) [73] .

- **System/End-to-End Tests:** These are broad tests covering the entire system as a black box. In robotics, this often means simulation tests. For instance, use Gazebo or a physics simulator to run the whole stack and verify high-level behavior (e.g., the robot can walk a step or maintain balance for 10 seconds without falling). These tests are the highest level and typically the fewest because they are time-consuming and sometimes flaky (due to sim timing or nondeterminism). Nonetheless, having a few smoke tests like "spawn the robot in Gazebo and see if all crucial nodes come up and no critical errors are logged" is very useful. You can automate such a test with launch_testing by launching a Gazebo world and all your bringup launch, then checking for certain conditions (for example, check that `/tf` is publishing transforms, or that a topic `/battery_state` has published at least once). System tests might also involve actual hardware in a staging environment (if available), but running on hardware in CI is often not feasible. Instead, you might run hardware tests manually before releases.

Keep system tests separate (possibly in a dedicated package or a test suite that is not run on every commit, but maybe nightly or pre-release) to avoid slowing down regular development. ROS 2's build farm, for instance, may not run Gazebo for each PR, but you can configure a limited set of integration tests to always run, and heavy simulation tests to run less frequently. The Developer Guide notes that system tests (end-to-end across packages) are often in their own packages to avoid introducing circular dependencies and to isolate heavy test dependencies [74] .

- **Test Coverage and Quality:** Aim for high coverage in critical algorithms (controllers, planners, etc.). ROS 2 core packages aim for >95% line coverage [75] , which is a good inspiration, though your project might not reach that initially. Use coverage tools: for C++, you can compile with `-coverage` or use lcov/genhtml to get coverage reports; for Python, `pytest --cov`. This helps identify untested code. However, don't chase 100% blindly – focus on *important logic*. For example, test edge cases like singularities in kinematics, loss of sensor input, etc. Also test error-handling paths (if a function is supposed to throw on bad input, test that it does).

- **Continuous Testing:** Integrate tests with a Continuous Integration (CI) system (like GitHub Actions, GitLab CI, Jenkins). There are ROS 2 Docker images or GitHub Action setups that can build and test your ROS 2 package on every push. Use these to catch issues early. Also, run local tests frequently during development (colcon can watch for file changes, or you can run individual tests directly). Write tests along with new features (Test-Driven Development is great if you can, but at least add tests when fixing bugs to lock in the correct behavior). A culture of "if it's not tested, it might be broken" can help maintain code quality as the codebase grows.

- **Code Coverage in CI:** If feasible, add a CI job that calculates test coverage and fails if coverage drops below a threshold. This ensures new code comes with tests. Even without gating, having the coverage report can motivate contributions to increase tests. The ROS 2 community emphasizes testing, as seen with the quality levels for packages – to reach quality level 1 (highest), a package needs extensive tests and coverage [76].

In summary, adopt a layered testing strategy: **Unit tests** for individual functions/classes (fast and numerous), **Integration tests** for ROS node interactions and subsystems, and occasional **End-to-End tests** for full system validation. This strategy, often depicted as a test pyramid, ensures confidence at all levels of the system with efficient use of testing resources [77] [70]. Writing tests may seem time-consuming, but it pays off by catching regressions early and facilitating safe refactoring – which in robotics can prevent physical mishaps and downtime.

Code Review and Quality Assurance

Code reviews are a cornerstone of maintaining high code quality. Every code change (especially by junior developers) should ideally be reviewed by someone else on the team. A good review process catches bugs, enforces standards, and spreads knowledge. Here are protocols and tools to use:

- **Pull Request (PR) Workflow:** Use a version control system (git) and have a clear branching strategy (e.g., feature branches that get merged into main via PRs). Each pull request should focus on *one logical change or feature* [78]. Avoid PRs that mix unrelated changes (for example, don't refactor the entire codebase in the same PR as adding a new feature – split them). This makes reviews manageable. In the PR description, explain the purpose of the change, and mention any new dependencies or migrations. Link to issues if applicable. Follow the project's template if there is one (some projects use templates that prompt for "What does this PR do", "How to test it", etc.).

- **Review for Readability and Maintainability:** Reviewers should read the code not just for correctness, but for clarity. Could another developer understand this code six months from now? Are variable and function names self-explanatory? If the code is doing something non-obvious, is there a comment explaining the rationale? Ensure that the code adheres to the coding standards outlined in this codex (naming, style, etc.). If not, request changes to bring it into compliance (or fix it directly if minor). Consistency is key for maintainability [79] [37]. For example, if most of the code uses `snake_case` and a new contribution uses `CamelCase` for variables, point that out in review.

- **Automated Tools in Review:** Set up automated checks that run on PRs. These include:

- **Linters/Formatters:** Run `ament_lint` (which includes cpplint, pep257, etc.) automatically. If a PR fails linting, have the CI mark it, or even better use a bot or git hook to auto-format (e.g., using `ament_uncrustify` for C++ and `black` or `yapf` for Python). This reduces nitpicks in reviews about spaces or braces because the tool enforces it. ROS 2's recommended linters (via `ament_lint_common`) should be enabled and all packages use them [80]. This covers code style, common errors, etc. For instance, `ament_cpplint` will ensure no forbidden functions, `ament_flake8` covers Python PEP8 violations, and so on. Have your CMakeLists include `ament_lint_auto` to run all linters by default [80].
- **Static Analysis:** Use tools like clang-tidy or cppcheck in the CI. They can catch performance or security issues (e.g., use-after-free, misuse of API) that reviewers might miss. There are ROS 2 support packages for static analysis (e.g., `ament_clang_tidy`). Make sure to configure these tools

according to your code standards (for example, set the clang-tidy config to Google style with ROS customizations).

- **Continuous Integration (CI):** The CI pipeline should build the code, run tests, and run linters for each PR. This automated "quality gate" prevents broken code from being merged. It also frees reviewers to focus on design and logic, rather than "please run clang-format". Use the ROS Build Farm or GitHub Actions for this. For example, you might use the official ROS 2 docker image in a CI job to colcon build and test your packages on Ubuntu 22.04, ensuring your new code doesn't break the build or tests. Merging is only allowed when CI passes (and ideally when at least one human review approves).

- **Review Criteria:** When reviewing, consider:

- **Correctness:** Does the code do what it's supposed to? Check logic, calculations, state machine transitions, etc. Sometimes it helps to pull the branch and run it (or run the new unit tests).
- **Adherence to Requirements:** If there's an issue or design document, does the code solve that? If the code introduces new public APIs (like new ROS topics/services or config parameters), are they named and handled appropriately?
- **Complexity:** Is the code as simple as possible? Could it be broken into smaller functions or be made more generic for reuse? Conversely, is it over-engineered for the current need? Aim for clear, straightforward implementation. Complex algorithms may need more comments or even a reference to a paper.
- **Tests:** Are there tests covering the new changes? If not, request adding tests. At minimum, ensure the main functionality has some level of test. If the PR is a bug fix, a regression test should be included.

- **Documentation:** If the change affects user-facing behavior, ensure documentation (package README or ROS interface documentation) is updated. For example, if a new parameter is added, the package's documentation should mention it. Even internal code benefits from documentation: e.g., non-obvious functions should have a brief Doxygen comment.

- **Code Review Etiquette:** For junior developers, code reviews are also a learning opportunity. Feedback should be constructive and explanatory. Rather than just saying "use snake_case for this," explain that "our convention is snake_case for variable names, to be consistent with ROS and PEP8 [17]." Encourage questions. If a piece of code is particularly well done, mention that too – positive feedback helps reinforce good practices. Keep the tone collaborative: it's about making the code better, not a personal critique. Conversely, as a reviewee, try not to take criticism personally; everyone's goal is a high-quality codebase.

- **Review Checklist:** It can be useful to have a checklist for reviewers:

- Code compiles and passes tests (CI is green).
- Code follows style guidelines (automated check or visual scan).
- All new ROS interfaces (topics, services, params) are documented and use proper naming.
- No obvious memory leaks or thread issues (e.g., each `new` has a `delete` or better, uses smart pointers).
- Functions have clear responsibilities; lengths are reasonable.
- Unit tests cover new logic (and pass).

- Any deprecated practices replaced (e.g., no ROS 1 remnants like `ros::spinOnce()` in ROS 2 code, etc.).
- Performance implications considered (e.g., no heavy computation in high-frequency callbacks without need).
- Security/safety considered (e.g., asserts or error handling for potentially bad inputs).
- Sufficient comments for hard-to-understand parts.

- Changelogs or version bump prepared if this is a significant change (for releases).

- **Pair Programming/Over-the-Shoulder Reviews:** Sometimes a synchronous review (either in person or via screen share) can be valuable, especially for complex code. The developer can walk the reviewer through the changes. This isn't always necessary for each PR but can accelerate understanding for tricky parts or help mentor a junior through a new design.

- **Maintaining Quality Long-Term:** Over time, do periodic **refactoring** and **debt pay-down**. If reviews identify code that is functional but could be cleaner, file a tech debt task or schedule refactoring sprints. Keep an eye on lint output even for things not enforced – e.g., warnings about complexity. Use tools like **ros2 doctor**, **clang-tidy** static checks, etc., as part of continuous quality. If the team adopts new conventions (say you decide to use `auto` more in C++ for readability), ensure reviewers propagate that in future PRs.

In essence, the code review process is about **ensuring readability, enforcing standards, and catching issues early**. It complements testing by providing a human perspective – for instance, a test might not catch that a piece of code is overly complex or using an outdated approach, but a reviewer can. With a strong review culture, even junior devs can contribute significant code while maintaining the integrity and quality of the codebase.

Linting, Formatting, and CI/CD Automation

Consistent style and continuous quality checks are achieved through linting/formatting tools and CI/CD (Continuous Integration/Continuous Deployment) practices. This section describes how to use these tools to uphold coding standards automatically:

- **Linters and Formatters:** ROS 2 comes with a suite of linters (via the `ament_lint` packages) to enforce code style and find common issues. Set up your packages to use **ament_lint_auto** and **ament_lint_common**, which include checks for C++ (cpplint, uncrustify), Python (pycodestyle, flake8, PEP257 for docstrings), CMake lint, XML lint, etc. [80] [81]. For C++:
- Use `ament_uncrustify` with the ROS 2 code style configuration to automatically format C++ code. Uncrustify will handle spacing, indentations, line breaks, etc., according to the Google style (with ROS modifications). You can run it via `ament_uncrustify` target or have an IDE integrate it.
- Use `ament_cpplint` for additional style guide enforcement (naming, etc.). Note that ROS 2's cpplint is configured to allow some deviations (like snake_case functions) [4].
- Optionally, use `ament_clang_format` or `ament_clang_tidy` for even more rigorous checks (clang-tidy can catch bugs or suggest modernizations).

- The ROS Quality Guide suggests that all packages *must* use these standard linters [80]. Enable them by adding in CMakeLists:

```
find_package(ament_lint_auto REQUIRED)
ament_lint_auto_find_test_dependencies()
```

This will automatically include all linters configured in your environment (which `ament_lint_common` brings in). You can also selectively enable/disable certain lints if needed, but strive to keep as many as possible.

For Python: - `ament_flake8`: runs flake8 which checks PEP8 compliance and some logical errors. Configure the flake8 in `setup.cfg` if needed (e.g., line length). - `ament_pep257`: checks that Python docstrings exist and follow conventions. - **Black** (the Python formatter) is not (as of Humble) in ament by default, but you can use it manually as a pre-commit hook or editor format-on-save. Black autoformats Python code to a consistent style (PEP8-ish). If you use black, mention it in CONTRIBUTING docs and enforce it by CI (some projects fail CI if `black --check` finds differences).

- **Continuous Integration (CI):** Set up a CI pipeline that triggers on pushes/PRs. Commonly, this pipeline will:
- **Build** the code (colcon build) on a fresh environment (to ensure all dependencies are declared).
- **Run Linters:** Execute `colcon test --packages-select <mypkg> --ctest-args -L lint` or simply rely on `ament_lint_auto` which runs linters as tests. This ensures style guidelines are followed. If any lint test fails, CI should mark the build failed.
- **Run Unit/Integration Tests:** `colcon test` will run all tests. Ensure the CI exports the test results (`colcon test-result` or collecting JUnit reports) so you can see what failed. CI should fail if any tests fail.
- (Optional) **Compute Coverage:** You could have a job to run tests with coverage instrumentation and archive the report (to track coverage over time).
- (Optional) **Static Analysis:** A separate job to run tools like cppcheck or clang-tidy over the codebase (not failing the build but reporting issues maybe).

Use ROS-specific base images for CI (like `ros:humble-ros-core-focal` docker) to avoid the overhead of installing ROS every time. For example, a GitHub Actions workflow might use `ubuntu-22.04` VM, then `apt install ros-humble-desktop`, etc., but it's faster to use a container with ROS pre-installed.

- **Continuous Deployment (CD):** For internal projects, you might not need automatic deployment, but if you do (for example, automatically build a debian and deploy to a robot, or create a docker image), consider a CD step. ROS 2 has the concept of the build farm for releasing into ROS distributions. For an internal project, you could use the same bloom release approach to generate .deb packages if you want. Alternatively, use Docker to containerize the whole workspace and deploy the container to robots. Ensure whatever deployment pipeline you use is also automated and reproducible.

- **Pre-commit Hooks:** To help developers lint before pushing, consider using git hooks or tools like **pre-commit**. For instance, you can configure a pre-commit config to run `ament_uncrustify`, `flake8`, etc., on changed files. This catches issues early and reduces back-and-forth in CI.

- **IDE Integration:** Encourage setting up IDE or editor to use the same formatting/linting. VSCode with the ROS 2 extension can auto-configure include paths and has some linting integration. For C++

formatting, provide the `.clang-format` or uncrustify config in the repo so developers' editors format correctly. For Python, most editors can integrate flake8 or black.

- **Documentation and CI:** If you generate documentation (e.g., Doxygen for C++ API or Sphinx for a user manual), you can automate that with CI as well. Perhaps have a job that builds docs and deploys to a GitHub Pages or internal site whenever main branch updates.

- **ROS 2 Build Farm (Optional):** If your project is meant to be publicly released or just to use ROS's infrastructure, you might consider bloom releasing it into your own repository or the ROS index. That's beyond a junior level, but be aware that maintaining a package to REP-2004 Quality Level 1 involves CI for multiple platforms, nightly builds, etc., similar to what ROS 2 does. For an internal project, adopt what makes sense (likely focusing on your target platform, e.g., Ubuntu 22.04).

- **Issue Trackers and Quality Metrics:** Use an issue tracker to note code quality tasks (like "increase test coverage for X" or "refactor Y module"). This ensures code quality isn't only addressed during crises. Some teams use quality metrics (lint score, cyclomatic complexity) – these can be tracked via tools integrated in CI. For example, SonarQube can be used to track code smells and tech debt. At minimum, keep the code free of compiler warnings and lint warnings; treat warnings as errors in CI to enforce that.

- **Deployment Considerations:** When deploying to actual robots, ensure you have a process. Possibly use CI to build binaries or Docker images and then a manual or automated step to update the robot. Consider package versioning so you know what code is running on the robot. This is more operational, but it ties into code quality: a consistent deployment process avoids the "it works on dev, but robot has old code" issues.

By leveraging automated linting and CI, you create a **fast feedback loop** for developers. Instead of style and basic errors being discussed in reviews, they're caught by tools, allowing reviews to focus on deeper issues. Automation also ensures that as the team and codebase scale, quality remains high – every change is vetted through the same strict process. Remember the adage: *"If it's not tested, it doesn't work."* Similarly, if CI isn't green, the code isn't ready. Treat the CI build status as sacrosanct; don't merge broken code, and strive to keep the main branch always in a deployable state.

ROS 2- and RViz2-Specific Best Practices

Humanoid robot development brings some special considerations in the ROS 2 context. This section highlights ROS 2 and RViz2 best practices that supplement the general ones above:

- **ROS 2 Node Architecture:** Design your system with ROS 2's features in mind. Use **multiple nodes** rather than one giant node that does everything. For example, separate a walking gait generator node from a balance controller node; they can communicate via topics or services. This not only follows separation of concerns, but allows you to distribute load and potentially restart components independently. Consider using **Lifecycle Nodes** (`rclcpp_lifecycle::LifecycleNode`) for hardware drivers or other nodes that need controlled startup/shutdown behavior (e.g., sensors that need to be initialized and torn down in a controlled sequence) [82]. Lifecycle nodes have states (unconfigured, inactive, active, etc.) which can be managed via ROS services – this is great for bringup systems where you want to initialize all hardware, then activate all controllers, etc., in steps. Use lifecycle management especially if you foresee needing to reset or reconfigure parts of the system on the fly. For computational nodes that don't need this, a regular Node is fine.

- **Parameters and Configuration:** Use ROS 2 **parameters** instead of hard-coded constants for anything you might want to tune without recompiling (PID gains, topic names to subscribe to, frame IDs, etc.). Leverage parameter files (YAML) to manage groups of parameters and load them via launch files [43] [83]. This allows you to have different configurations (e.g., one for simulation, one for real robot) by using different YAML files, without changing code. Utilize parameter events or callbacks if you want to make parameters dynamic (e.g., allow teleop speed to change at runtime by setting a param). Document the parameters in the node's header or class docstring so it's clear what each one does. Also, limit parameter use to configuration – for high-frequency data (like a control setpoint updated at 1kHz), don't use parameters; that's what topics are for. Parameters are more for static or slow-changing config.

- **Launch File Composition:** For complex robots, consider breaking launch files into smaller ones and including them. For instance, have a `sensors.launch.py` that launches all sensors, a `control.launch.py` for controllers, etc., and then the main `bringup.launch.py` includes those (using `IncludeLaunchDescription`) [84] [42]. This modularizes your launch, making it easier to manage and reuse parts. Also use launch arguments to allow toggling options (like `use_sim_time`, or which world file to load in simulation, etc.). The ROS 2 Launch system allows conditions on launch actions, which is useful, e.g., *if* a `gui` arg is true, then include RViz node. Use those instead of maintaining separate nearly-identical launch files.

- **Package Layout and Reuse:** As covered earlier, follow the standard package layout [63] [64]. This includes keeping **URDF/Xacro in a description package** and not duplicating it elsewhere. Then both the real robot bringup and simulations can use the same robot description. Use the **robot_state_publisher** to publish TF transforms from your URDF – do not publish static TFs manually that could be sourced from the URDF; keep one source of truth (the URDF) for robot kinematics. If you have moving joints, use **joint_state_publisher** (or your controllers publishing JointState messages) in conjunction. The best practice is to have an aggregator of JointState (ros2_control often does this) and feed robot_state_publisher. Ensure that TF frame names in URDF comply with ROS conventions (base_link, etc., as per REP-120/105 for consistency) [21] [85].

- **URDF and Xacro Organization:** Keep the URDF **modular and organized**. For a complex humanoid, break it into multiple Xacro files for leg, arm, torso, sensor head, etc., and assemble them in a top-level file [19]. Use Xacro properties for things like mass, dimensions, so they can be changed in one place. E.g., define `leg_length` property and use it in multiple joints if needed. Also make use of Xacro's include capability to include common piece parts (like a generic `camera_macro.xacro` if you have multiple same cameras). Make sure to run `xacro --check` occasionally to catch Xacro syntax errors. Use `ros2 run xacro xacro --inorder model.xacro > model.urdf` in a build process to generate an URDF for release or debugging (you can even add a CMake target for this, see ROS 2 Xacro tutorial [86]). Keep visual and collision geometry sensible (collision simpler shapes where possible for performance). If the humanoid is very complex, consider using `<gazebo>` tags (SDF extensions in URDF) for physics fine-tuning but keep them optional.

- **RViz2 Configs:** For RViz2, create config files (*.rviz) tailored for your robot – e.g., showing the robot model, important TF frames, sensor feeds (camera images, lidar point clouds) and some custom markers. Save these in a `rviz/` or `config/` directory in the appropriate package. That way, any developer can quickly load RViz with a known-good visualization setup by running `rviz2 -d <your_config>.rviz`. It's helpful to have at least one RViz config for general robot monitoring (maybe in the bringup or description package). Also, if you develop custom RViz plugins (panels or displays), follow the RViz2 plugin development best practices and keep them in a separate package (with proper export in pluginlib). RViz2 is great for debugging – you might set up a config that has plots of certain topics (use the Plot or message display), or a TF tree view, etc. Encourage team members to use and refine these RViz configs as the robot's capabilities grow.

- **Simulators and Tools:** Leverage simulation tools like Gazebo wisely. Keep simulation artifacts (world files, robot gazebo plugins) either in a separate simulation-specific package or clearly marked (like a `gazebo/` folder in description for gazebo-specific Xacro includes). This separation prevents sim-specific hacks from leaking into real robot code. Use RQt and other ROS tools for introspection: e.g., rqt_graph to verify your topic connections (this can be part of a manual test checklist). For performance analysis, use `ros2 topic hz` or `ros2 topic bw` to ensure you're not overloading the system. If you have real-time requirements, consider the ROS 2 real-time guide (switch to realtime kernel, use `rclcpp::CallbackGroupType::MutuallyExclusive` to control executor behavior, etc.). For logging, follow best practices: don't spam at info level (use debug for very frequent logs), and ensure logs have context (e.g., include the joint name in a warning about joint limits rather than a generic "limit reached").

- **ROS2 Middleware Tuning:** If your humanoid robot has high-bandwidth or low-latency needs (like high-frequency IMU at 400Hz, images at 30Hz, etc.), be aware of QoS (Quality of Service) settings in ROS 2. Best practice is to explicitly set QoS profiles for your publishers and subscriptions rather than relying on defaults, especially for sensors (use sensor QoS profile for sensor data, which is best-effort, keep-last with depth). For teleop or critical commands, perhaps use reliable QoS. Also, in multi-machine setups, consider the `ROS_DOMAIN_ID` separation and FastDDS discovery server if needed (ROS 2 docs have guides for lossy networks, etc.). The specifics might be beyond junior level, but at least know that QoS matters – e.g., if you see dropped messages, increase the queue depth or switch to best-effort where appropriate.

- **Checking System Resource Usage:** Regularly check CPU and memory usage of each node (use `top` or rqt_top). A best practice is to ensure no single node is pegged at 100% CPU in nominal operation – if so, consider optimizing or splitting work. Also ensure your launch files set `prefix` with `nice` or `taskset` if you need to isolate CPU or priority for important processes (for example, a control loop might be run with a higher process priority if needed to maintain timing).

- **Leverage Community Patterns:** ROS has many design patterns (like *component nodes* that can be dynamically loaded, etc.). For Humble, composing multiple nodes into one process (to eliminate interprocess overhead) is possible using the composition API. If you have many small nodes that communicate at high frequency, consider composition to reduce latency (with caution to not complicate debugging). Also keep an eye on ROS Discourse or ROS Answers for best practices specific to humanoids (there are often discussions about how to manage large URDFs, or how to implement walking algorithms within ROS framework, etc.).

By following these ROS 2-specific best practices, you ensure that your humanoid robot software is idiomatic to ROS 2 and takes full advantage of its capabilities. This will make your system more robust and easier to integrate with the ROS ecosystem (e.g., using rosbag, RViz, rqt tools seamlessly). Moreover, adhering to conventions (like standard frame names, proper package layout, using parameters) means other ROS developers can understand and use your system more readily [87] [23].

Conclusion

Maintaining high-quality code in humanoid robot development is challenging but immensely rewarding. By adhering to the guidelines in this codex – from consistent naming and style, proper use of tools and environment, secure coding, modular architecture, rigorous testing, thorough reviews, to leveraging ROS 2's features – the development team can ensure that the software is reliable, maintainable, and scalable.

Humanoid robots are complex systems; a disciplined approach to code quality reduces bugs that could lead to erratic behavior or difficult downtimes. It also speeds up onboarding of new developers, as they find a coherent and well-documented codebase. Remember that coding standards are not static: continue to refine this codex as ROS 2 and best practices evolve (ROS 2 will introduce new features and deprecate old ones, and the team may decide on new conventions as the project grows). Encourage a culture where team members not only follow the rules but understand *why* they exist – this leads to more thoughtful coding.

In daily practice, use this codex as a reference: before committing code, run through the relevant sections (did I name things clearly? Did I add tests? Is there a potential security hole here? etc.). Over time these will become second nature. When in doubt, consult ROS 2's extensive documentation and the community for answers – many authoritative references have been cited here (like ROS Dev Guides, REPs, and tutorials) [63] [80]. Following those ensures alignment with the broader ROS community standards.

By building on these foundations, the team will create software for humanoid robots that stands the test of time – easily adaptable to new hardware, robust against failures, and ready for new features. High-quality code is the backbone of turning ambitious humanoid robotics ideas into reality in a dependable way. Let's keep the bar high and continuously improve our craft as we develop these complex robotic systems.

1 2 3 4 5 6 7 8 9 10 13 14 16 17 37 71 72 79 81 82 Code style and language versions — ROS 2 Documentation: Humble documentation
https://docs.ros.org/en/humble/The-ROS2-Project/Contributing/Code-Style-Language-Versions.html

11 50 53 54 What Is CERT C++? Definitions and Examples - Parasoft
https://www.parasoft.com/blog/theres-no-good-reason-to-ignore-cert-c/

12 15 18 25 26 27 28 31 63 64 67 68 69 70 73 74 75 76 77 78 80 ROS 2 developer guide — ROS 2 Documentation: Rolling documentation
https://docs.ros.org/en/rolling/The-ROS2-Project/Contributing/Developer-Guide.html

19 20 87 Describing robots with URDF | Articulated Robotics
https://articulatedrobotics.xyz/tutorials/ready-for-ros/urdf/

21 22 85 REP 120 -- Coordinate Frames for Humanoid Robots (ROS.org)
https://www.ros.org/reps/rep-0120.html

23 24 29 42 43 44 47 48 56 57 58 59 60 61 62 65 66 83 84 Package Organization For a ROS Stack [Best Practices] - The Robotics Back-End
https://roboticsbackend.com/package-organization-for-a-ros-stack-best-practices/

30 ROS Package Naming | Robotics Enhancement Proposals
https://reps.openrobotics.org/rep-0144/

32 34 Topic and Service name mapping to DDS
https://design.ros2.org/articles/topic_and_service_names.html

33 Naming convention - Autoware Documentation
https://tier4.github.io/pilot-auto-ros2-iv-archive/tree/main/design/software_architecture/NamingConvention/

35 36 Understanding parameters — ROS 2 Documentation: Rolling documentation
https://docs.ros.org/en/rolling/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Parameters/Understanding-ROS2-Parameters.html

38 39 40 41 86 Using Python Packages with ROS 2 — ROS 2 Documentation: Humble documentation
https://docs.ros.org/en/humble/How-To-Guides/Using-Python-Packages.html

45 46 Building a visual robot model from scratch — ROS 2 Documentation: Humble documentation
https://docs.ros.org/en/humble/Tutorials/Intermediate/URDF/Building-a-Visual-Robot-Model-with-URDF-from-Scratch.html

49 51 52 55 What Are Secure Coding Standards? CERT, OWASP, and Compliance Explained | Kiuwan
https://www.kiuwan.com/blog/secure-coding-guidelines/