Deep Dive into ROS 2 Waypoint Coordination with MoveIt 2

1. Introduction

In the realm of robotics, the ability to coordinate movements through a series of predefined locations, known as waypoints, is fundamental for achieving autonomous navigation and executing complex tasks. This report presents a detailed analysis of a Python code segment designed to manage robot waypoints within the Robot Operating System 2 (ROS 2) framework, leveraging the capabilities of MoveIt 2 for motion planning. MoveIt 2 stands as a robust open-source platform offering advanced functionalities for robot manipulation, including kinematics, motion planning, collision checking, and execution. The code under examination orchestrates a sequence of actions: it first determines the robot's current end-effector pose, proceeds to calculate a new target pose by applying an offset, records these poses in a Comma Separated Values (CSV) file, and subsequently instructs the robot to move to the calculated offset pose using MoveIt 2. This process involves reading the robot's state, transforming it according to a specified parameter, and then commanding the robot to achieve this new state, illustrating a common paradigm in robotic control systems. The use of a CSV file as an intermediary in this process suggests a deliberate design choice that could facilitate logging, debugging, or interaction with external systems or manual adjustments of the waypoint data.

2. Overview of rclpy in ROS 2

The foundation of this code lies within the rclpy library, which serves as the primary Python interface for developing applications within the ROS 2 ecosystem [1]. Standing for ROS Client Library for Python, rclpy provides a comprehensive set of tools and functionalities that enable Python developers to interact with the core concepts of ROS 2 [1]. These fundamental concepts include the creation and management of nodes, which are the basic executable units in a ROS 2 system [1]. rclpy also facilitates communication between these nodes through various mechanisms such as topics for asynchronous messaging, services for request-response interactions, and actions for managing long-running tasks with feedback [2]. Furthermore, it provides access to essential ROS 2 features like parameter management for runtime configuration, a standardized logging system for outputting information and debugging messages, and utilities for handling time [2].
rclpy distinguishes itself by providing an idiomatic Python experience, utilizing native Python data types and programming patterns [2]. Underneath this Python interface, rclpy is built upon the rcl C API, which forms the core of ROS 2 and ensures consistency in behavior and feature parity across different client libraries, such as rclcpp for C++ [2]. This underlying C API contributes to the performance and stability of Python-based ROS 2 applications. Moreover,

rclpy manages the execution model through the use of threading, allowing for concurrent operations within a single node, which is particularly relevant for handling asynchronous tasks like action execution and timer events [2]. An important aspect of ROS 2 is the ability for nodes written in different client libraries to communicate seamlessly. This interoperability is achieved because all ROS 2 client libraries employ code generators that can process standard ROS interface definition files (.msg for messages and .srv for services), enabling them to understand and exchange data regardless of the programming language used for implementation [2]. The choice of Python and rclpy for this waypoint coordinator likely reflects a desire for rapid development and the ability to leverage Python's extensive ecosystem of libraries for data processing and other related functionalities. The architecture of rclpy, building on the rcl C API, strikes a balance between ease of use for developers and the performance requirements often encountered in robotics applications.

3. CSV File Handling with the csv Module

The code utilizes Python's built-in csv module to manage data stored in Comma Separated Values files [6]. This module provides functionalities for both reading and writing data in this common tabular format. Within the run_cycle function, the code performs two primary operations involving the csv module: writing the current and offset robot poses to a CSV file and subsequently reading the offset pose back from the same file.

For writing, the code first opens the CSV file specified by the self.csv_path parameter in write mode ('w') using a with statement, which ensures that the file is properly closed afterwards, even if errors occur. It then creates a csv.writer object, which provides methods for writing data to the file. The first row written is a header row containing the labels 'x', 'y', 'z', 'ox', 'oy', 'oz', 'ow'. These labels correspond to the x, y, and z coordinates of the position and the x, y, z, and w components of the quaternion representing the orientation of the robot's end-effector. Following the header, two data rows are written. The first data row contains the current pose of the robot, obtained using the TF system. The second data row contains the calculated offset pose. The position components are accessed from the current_pose and offset_pose objects, and similarly, the orientation components (as a quaternion) are accessed. The code includes a try...except block around the file writing operations to handle potential exceptions, such as issues with file permissions or the specified path. If an error occurs during writing, an error message is logged.

For reading, the code again opens the same CSV file, this time in read mode ('r'), also using a with statement. It creates a csv.reader object, which allows iteration over the rows of the CSV file. The code then reads all the rows into a list called lines. It checks if this list contains at least three rows, assuming the first row is the header, the second is the current pose, and the third is the offset pose. If there are sufficient rows, the code extracts the third row (at index 2) and parses the string values in this row into floating-point numbers using the float() function. These numerical values are then assigned to the corresponding attributes (position and orientation) of a new geometry_msgs.msg.Pose object named parsed_pose. Similar to the writing part, the reading operation is also enclosed in a try...except block to catch potential errors, such as the file not being found or the data in the CSV not being in the expected

numerical format. If a parsing error occurs, an error message is logged.
The immediate writing and subsequent reading of the CSV file within the same execution cycle of the run_cycle function might appear to introduce an unnecessary step. However, this design choice could serve several purposes. It might be a simplified approach for logging or debugging the calculated offset poses, allowing for easy inspection of the data. In a more complex scenario, the CSV file could act as an interface with other processes or systems that might need to access or modify the waypoint data. Additionally, it could represent a form of data persistence, although in this specific implementation, the file is overwritten in each cycle. The header row in the CSV file clearly defines the structure and meaning of the data, which is crucial for both human readability and for any program that might need to parse this file. The use of standard column names for position (x, y, z) and quaternion orientation (ox, oy, oz, ow) aligns with common conventions in robotics for representing 3D poses.

4. ROS 2 Node Implementation (WaypointCoordinator Class)

The core of the provided code is the WaypointCoordinator class, which defines a custom ROS 2 node. In ROS 2, a node is a fundamental building block that performs specific computations and communicates with other nodes in the system [1]. To create a ROS 2 node in Python using rclpy, a class must inherit from the rclpy.node.Node class [1].
The WaypointCoordinator class begins with its constructor, __init__(self). The first line inside the constructor is super().__init__('waypoint_coordinator'), which calls the constructor of the parent Node class, initializing the node with the unique name 'waypoint_coordinator' within the ROS 2 network [1]. Following this, the node declares two ROS 2 parameters using self.declare_parameter('csv_path', '') and self.declare_parameter('offset', 0.00). Parameters are configuration variables that can be set at runtime, either through the command line or via a launch file, without requiring modifications to the code itself [2]. Here, 'csv_path' is intended to store the file path for the CSV file, with an initial default value of an empty string, and 'offset' is intended to store a numerical offset value, with a default of 0.00. The current values of these parameters are then retrieved and stored as attributes of the WaypointCoordinator instance using self.csv_path = self.get_parameter('csv_path').value and self.offset = self.get_parameter('offset').value.
Next, the node initializes the necessary components for working with TF (Transformations). self.tf_buffer = tf2_ros.Buffer() creates a buffer to store coordinate transformations broadcast by other nodes, and self.tf_listener = tf2_ros.TransformListener(self.tf_buffer, self) creates a listener that subscribes to the relevant TF topics and populates the buffer with the transformations [3]. The self argument indicates that the current node instance will be used as the context for TF operations.
The node then sets up an action client to interact with a MoveIt 2 action server. self._action_client = ActionClient(self, MoveGroup, '/move_action') creates an action client that will communicate with an action server of type MoveGroup (defined in the moveit_msgs.action package) that is expected to be running on the ROS 2 topic named /move_action [1]. The self

argument again refers to the current node. The variables self._send_goal_future and self._get_result_future are initialized to None; these will be used to store future objects associated with asynchronous action calls.
To ensure sequential execution of its core logic and to introduce a periodic behavior, the node uses a timer. self.is_executing = False initializes a flag to track whether the main cycle is currently running. self.create_timer(3.0, self.run_cycle) creates a timer that will trigger the self.run_cycle method every 3.0 seconds [1]. This sets the frequency at which the node will attempt to get the current pose, calculate the offset, and send a motion goal.
Finally, the node sets up a publisher for the /planning_scene topic using self.scene_publisher = self.create_publisher(PlanningScene, '/planning_scene', 10). This publisher will be used to send PlanningScene messages, which can include information about the robot's environment, such as collision objects. The 10 represents the queue size for outgoing messages. The constructor concludes with a call to self.add_ground_plane(), a method defined within the class to add a virtual ground plane as a collision object to the planning scene.
The use of parameters allows for easy configuration of the node's behavior without needing to alter the underlying code. For example, the user can specify the path to the CSV file and the desired offset value when launching the node. The 3-second timer dictates the rate at which the waypoint coordination process is executed. This interval might need to be adjusted depending on the specific application requirements and the desired responsiveness.

5. Coordinate Transformations using tf2_ros

The tf2_ros library in ROS 2 is essential for managing and accessing coordinate transformations between different reference frames within the robotic system [3]. It provides a way for nodes to understand the spatial relationships between various parts of the robot and its environment.
The tf2_ros.Buffer() created in the __init__ method acts as an in-memory storage for these transformations. Nodes throughout the ROS 2 system broadcast transformations, typically on the /tf and /tf_static topics, and the tf_buffer receives and stores these transformations, indexed by the source and target frames and the time at which the transformation is valid.
The tf2_ros.TransformListener(self.tf_buffer, self) then subscribes to these transformation broadcasts and populates the tf_buffer. The self argument passed here provides the necessary context for the listener to interact with the ROS 2 communication infrastructure.
Within the run_cycle function, the code first checks if the transformation between the 'base_link' frame and the 'wrist_3_link' frame is available in the buffer at the current time using self.tf_buffer.can_transform('base_link', 'wrist_3_link', rclpy.time.Time()). The 'base_link' frame is a common convention representing the main frame of reference for the robot, while 'wrist_3_link' often represents the frame attached to the robot's end-effector. This check is crucial to ensure that the requested transformation has been published by another node in the system before attempting to retrieve it. If the transformation is not available, the node logs a warning and skips the rest of the cycle.
If the transformation is available, the code retrieves it using self.tf_buffer.lookup_transform('base_link', 'wrist_3_link', rclpy.time.Time()). This function

returns a geometry_msgs.msg.TransformStamped object, which contains the transformation data, including the translation (position) and rotation (orientation) of the 'wrist_3_link' frame relative to the 'base_link' frame at the latest available time.
The position is then extracted from the trans.transform.translation component, with its x, y, and z attributes, and the orientation (as a quaternion) is extracted from the trans.transform.rotation component, with its x, y, z, and w attributes. These values are then used to populate a geometry_msgs.msg.Pose object representing the robot's current end-effector pose.
The reliance on the TF system indicates that this code expects another ROS 2 node to be actively broadcasting the transformation between the 'base_link' and 'wrist_3_link' frames. This is a standard practice in robotics, where sensors or robot state publishers provide information about the robot's pose and the poses of its various links. The use of rclpy.time.Time() in the TF lookup ensures that the node is working with the most up-to-date transformation information available in the buffer. This is important for accurately determining the robot's current state before calculating the offset and planning the subsequent motion.

6. Understanding ROS 2 Actions and the MoveGroup Action

ROS 2 actions provide a structured mechanism for managing long-running tasks that require feedback and can be cancelled, contrasting with the simpler request-response model of services [2]. An action involves an action client that sends a goal to an action server. During the execution of the goal, the action server can periodically send feedback to the client, and upon completion (either success, failure, or cancellation), the server sends a result back to the client.
In this code, the WaypointCoordinator node acts as an action client for the moveit_msgs.action.MoveGroup action, which is a standard action definition provided by the MoveIt 2 framework. The MoveGroup action allows a client to request a motion plan and its execution from a MoveIt 2 action server. This server, typically part of a MoveIt 2 setup configured for a specific robot, handles the complex tasks of motion planning, collision avoidance, and trajectory generation to achieve the desired goal pose for the robot's end-effector.
The WaypointCoordinator creates an action client for the MoveGroup action targeting the /move_action topic. When the run_cycle function determines that a motion goal needs to be sent (after calculating the offset pose and performing safety checks), it uses this action client to send a goal to the MoveIt 2 action server.
A key characteristic of ROS 2 actions is their asynchronous nature. When the WaypointCoordinator sends a motion goal using the send_goal_async method, it does not block and wait for the entire motion to complete. Instead, it receives a "future" object that represents the eventual outcome of the goal. The node can then continue with other tasks (although in this example, it primarily waits for the result through callbacks). The action server, meanwhile, works independently to plan and execute the robot's movement. Once the motion is complete, the server sends a result back to the client through the future object. The client

can then use callback functions to handle the response indicating whether the goal was accepted and the final result of the motion execution (success or failure). This asynchronous communication is particularly well-suited for tasks like motion planning and execution, which can take a significant amount of time.

7. Motion Planning Request Details

The WaypointCoordinator node communicates its desired robot motion to the MoveIt 2 action server by constructing and sending a moveit_msgs.action.MoveGroup.Goal message. This goal message contains a crucial field named request, which is of the type moveit_msgs.msg.MotionPlanRequest. This nested message is where all the specifics of the desired motion plan are defined.

Within the send_goal function, the code populates various fields of this MotionPlanRequest. The request.group_name is set to "ur_manipulator", indicating that the motion plan should be generated for the kinematic group representing the robot arm, likely on a Universal Robots (UR) robot. The request.num_planning_attempts is set to 20, specifying that the motion planner should try up to 20 times to find a valid motion plan that satisfies all the given constraints. The request.allowed_planning_time is set to 20.0 seconds, limiting the amount of time the planner can spend searching for a solution. If a plan is not found within this time, the planning process will time out. The request.planner_id is set to "RRTConnect", which selects a specific motion planning algorithm from the available options in MoveIt 2. RRTConnect is a popular sampling-based planner known for its efficiency in finding paths in complex, high-dimensional spaces. The request.max_velocity_scaling_factor is set to 0.4, which will scale down the maximum allowed velocity of the robot's joints to 40% of their configured limits during the execution of the planned trajectory. Similarly, request.max_acceleration_scaling_factor is set to 0.2, limiting the maximum acceleration to 20% of the configured limits. These scaling factors are important for safety and can help ensure smoother robot motions.

Table 2 summarizes these key parameters used to configure the motion planning request:

Table 2: MotionPlanRequest Parameters

Parameter	Value in Code	Description
group_name	"ur_manipulator"	The kinematic group to plan for (likely the robot arm).
num_planning_attempts	20	The number of times the motion planner will try to find a valid plan.
allowed_planning_time	20.0	The maximum time (in seconds) the planner can spend searching for a solution.

planner_id	"RRTConnect"	The identifier of the motion planning algorithm used.
max_velocity_scaling_factor	0.4	A factor (0.0 to 1.0) that scales the robot's maximum allowed velocity.
max_acceleration_scaling_factor	0.2	A factor (0.0 to 1.0) that scales the robot's maximum allowed acceleration.

The configuration of these parameters reflects a balance between the likelihood of finding a valid motion plan, the time it takes to find one, and the safety and smoothness of the resulting robot movement. These values would typically be tuned based on the specific requirements of the robot, the task, and the environment in which it operates.

8. Defining Constraints for Motion Planning

To specify the desired goal state for the robot's end-effector, the WaypointCoordinator utilizes constraints, which are added to the MotionPlanRequest. These constraints define the conditions that the planned motion must satisfy at its final point.

The code defines both a PositionConstraint and an OrientationConstraint. The PositionConstraint is created with its header.frame_id set to "base_link" and its link_name set to "wrist_3_link", indicating that the constraint applies to the position of the robot's end-effector relative to the base frame. The weight is set to 1.0, signifying that this constraint is important and should be fully satisfied if possible. The constraint itself is defined using a shape_msgs.msg.SolidPrimitive of type BOX with dimensions of 0.02 meters in each direction (a 2cm cube). The pose of this box, which defines the target region, is centered at the desired goal position obtained from the parsed offset pose, with its orientation set to an identity quaternion (no rotation). This effectively creates a small cubic region around the target position that the end-effector's origin must reach.

The OrientationConstraint is similarly defined, also with its header.frame_id as "base_link" and link_name as "wrist_3_link". The desired orientation is taken directly from the orientation component of the parsed offset pose. The code also sets tolerances for the allowed deviation around each axis of the desired orientation. absolute_x_axis_tolerance, absolute_y_axis_tolerance, and absolute_z_axis_tolerance are all set to 0.05 radians (approximately 2.86 degrees). This means that the final orientation of the end-effector can deviate by up to this amount around each axis from the desired orientation. The weight for this constraint is also set to 1.0.

Both the created PositionConstraint and OrientationConstraint are then added to a moveit_msgs.msg.Constraints object, which is subsequently appended to the goal_constraints list of the MotionPlanRequest. This tells MoveIt 2 that the planned motion must end with the 'wrist_3_link' positioned within the defined 2cm cube centered at the target position and oriented within the specified 0.05 radian tolerance of the target orientation. The use of a small tolerance box for the position constraint acknowledges that achieving an exact point in space can be difficult in real-world robotics, allowing for a small margin of error. The separate

orientation constraint ensures that the end-effector also achieves the desired angular orientation, which is crucial for many manipulation tasks.

9. Collision Avoidance with CollisionObject and PlanningScene

To prevent the robot from planning motions that would result in collisions with its environment, the code adds a representation of the ground plane as a collision object to the MoveIt 2 planning scene. The planning scene is a comprehensive representation of the robot, its environment, and the current planning parameters.
The add_ground_plane method creates a moveit_msgs.msg.CollisionObject named "ground". Its header.frame_id is set to "base_link", and its operation is set to CollisionObject.ADD, indicating that this object should be added to the scene. The shape of the ground plane is defined using a shape_msgs.msg.SolidPrimitive of type BOX with dimensions 4.0 meters by 4.0 meters in the x and y directions, and a thickness of 0.02 meters in the z direction. The pose of this box is specified using a geometry_msgs.msg.PoseStamped with its header.frame_id also set to "base_link" and its position's z-coordinate set to -0.03 meters. This places the ground plane slightly below the origin of the robot's base frame. The box primitive and its pose are then added to the primitives and primitive_poses lists of the ground collision object, respectively.
Finally, a moveit_msgs.msg.PlanningScene message is created. The ground collision object is appended to the collision_objects list within the world attribute of this planning scene message. Importantly, the is_diff flag of the PlanningScene message is set to True. This indicates that this message represents a difference from the current planning scene. When this message is published to the /planning_scene topic using self.scene_publisher.publish(scene), MoveIt 2 will update its internal representation of the environment by adding the ground plane, without needing to process a complete description of the entire scene.
Adding a ground plane as a collision object is a fundamental safety measure in robotics. It ensures that the motion planner will not generate trajectories that would cause the robot to move through the floor, which is physically impossible and could lead to errors in simulation or damage in a real-world setup. By including this information in the planning scene, the motion planner can take the ground into account when searching for collision-free paths to the desired goal poses. The use of is_diff = True is an optimization that reduces the amount of data that needs to be transmitted and processed, making the system more efficient, especially when the environment is updated frequently.

10. The run_cycle Function

The run_cycle function orchestrates the main logic of the WaypointCoordinator node. It is executed periodically based on the timer created in the __init__ method.
At the beginning of each cycle, it checks the self.is_executing flag. If it is True, the function returns immediately, preventing concurrent execution of the cycle. This ensures that the steps

within the cycle are performed sequentially. The flag is then set to True to indicate that a new cycle has begun.
The function first attempts to retrieve the robot's current end-effector pose using the TF buffer. It checks if the transformation between 'base_link' and 'wrist_3_link' is available. If not, it logs a warning and returns. If available, it looks up the transformation and extracts the position and orientation into a geometry_msgs.msg.Pose object called current_pose. This operation is wrapped in a try...except block to handle potential TF lookup failures.
Next, it calculates the offset_pose by taking the current_pose and adding the self.offset value to its x-coordinate. The y and z coordinates and the orientation remain the same. Following this, the code performs a workspace boundary check on the z-coordinate of the offset_pose, ensuring it falls within the safe limits defined by SAFE_Z_MIN and SAFE_Z_MAX. If the z-coordinate is outside these bounds, a warning is logged, and the cycle is interrupted.
The current_pose and offset_pose are then written to the CSV file specified by self.csv_path. The file is opened in write mode, a header row is written, followed by the data for both poses. This operation is also within a try...except block to handle potential file writing errors. After writing, the same CSV file is opened in read mode, and the offset pose is parsed from the third row (index 2) into a geometry_msgs.msg.Pose object called parsed_pose. This parsing is also done within a try...except block to handle potential file reading or data format errors.
A final safety check is performed on the z-coordinate of the parsed_pose to ensure it is not below the SAFE_Z_MIN threshold. If it is, an error is logged, and the cycle is interrupted. If all the checks pass, an info message is logged, and the send_goal method is called with the parsed_pose to send a motion planning request to MoveIt 2. Finally, the self.is_executing flag is set back to False, allowing the next cycle to begin when the timer triggers again.
The sequential execution within run_cycle ensures a controlled flow of operations. The inclusion of checks for TF availability, workspace boundaries, and the parsed pose demonstrates a focus on safety and error handling. The TODO comment about making the offset random suggests a potential future enhancement to the node's functionality.

11. Sending Goals to the MoveIt 2 Action Server (send_goal Function)

The send_goal function is responsible for communicating the desired robot motion to the MoveIt 2 action server. It begins by checking if the action server is available using self._action_client.wait_for_server(timeout_sec=5.0). If the server does not become available within 5 seconds, an error message is logged, the is_executing flag is set to False, and the function returns.
If the server is available, a moveit_msgs.action.MoveGroup.Goal message is created. A moveit_msgs.msg.MotionPlanRequest is then instantiated and populated with the planning parameters discussed earlier, such as the group_name, num_planning_attempts, allowed_planning_time, planner_id, and the velocity and acceleration scaling factors.
Next, the function defines the goal constraints. A PositionConstraint is created, specifying a small cubic region (2cm x 2cm x 2cm) centered at the desired goal position for the 'wrist_3_link' in the 'base_link' frame. An OrientationConstraint is also created, specifying the

desired orientation of the 'wrist_3_link' in the 'base_link' frame, along with tolerances of 0.05 radians around each axis. Both of these constraints are added to a moveit_msgs.msg.Constraints object, which is then appended to the goal_constraints list of the MotionPlanRequest.
The configured MotionPlanRequest is then assigned to the request field of the goal_msg. The goal is sent asynchronously to the action server using self._send_goal_future = self._action_client.send_goal_async(goal_msg). This returns a future object that will eventually hold the server's response to the goal. A try...except block is used to handle potential errors during the goal sending process.
Finally, a callback function, self.goal_response_callback, is added to the future object using self._send_goal_future.add_done_callback(). This callback will be executed when the action server sends a response indicating whether the goal was accepted or rejected. The timeout when waiting for the action server prevents the node from hanging indefinitely if the MoveIt 2 setup is not running correctly. Sending the goal asynchronously allows the send_goal function to return quickly, and the result is handled later by the callback functions.

12. Handling Action Callbacks (goal_response_callback and get_result_callback)

The goal_response_callback function is executed when the MoveIt 2 action server responds to the goal request sent by the WaypointCoordinator. It receives the future object as an argument. The goal handle is retrieved from the future's result. The function then checks if the goal was accepted by the server using goal_handle.accepted. If the goal was not accepted, an error message is logged, and the self.is_executing flag is set to False. If the goal was accepted, the function proceeds to get the result future using goal_handle.get_result_async() and registers another callback function, self.get_result_callback, to be executed when the action server sends the final result of the motion execution.
The get_result_callback function is called when the MoveIt 2 action server finishes processing the motion goal and sends back the result. It also receives a future object as an argument. The actual result, which includes an error code, is retrieved from the future. The function checks the value of result.error_code.val. If it is equal to result.error_code.SUCCESS, it means the motion was completed successfully, and a success message is logged. Otherwise, if the error code indicates a failure, an error message is logged along with the specific error code. Finally, the self.is_executing flag is set to False, allowing the next cycle of the run_cycle function to begin. The use of these two callbacks allows the WaypointCoordinator to handle different stages of the action execution process: the initial acceptance of the goal and the final outcome of the motion. Checking the error code provides valuable information for diagnosing potential issues with the motion planning or execution.

13. main Function and Node Execution

The main(args=None) function serves as the entry point for the Python script. It begins by

initializing the rclpy library using rclpy.init(args=args), which is a necessary step before creating any ROS 2 nodes. An instance of the WaypointCoordinator node class is then created. The rclpy.spin(node) function is called within a try...except KeyboardInterrupt block. This function keeps the node alive and processing callbacks until a KeyboardInterrupt exception is raised (typically by pressing Ctrl+C). The finally block ensures that even if an exception occurs, the node.destroy_node() method is called to properly destroy the node and release any resources it might be using, followed by rclpy.shutdown() to shut down the rclpy library. The if __name__ == '__main__': block is a standard Python construct that ensures the main() function is executed only when the script is run directly. This structure provides a standard way to initialize, run, and gracefully shut down a ROS 2 Python node.

14. Conclusion

The WaypointCoordinator node effectively demonstrates a fundamental robotic task of moving to a calculated offset from the robot's current pose using ROS 2 and MoveIt 2. The node periodically retrieves the robot's end-effector pose using TF, calculates a new target pose by applying a configurable offset, and then utilizes the MoveIt 2 action server to plan and execute the motion to this new pose while adhering to specified constraints and considering potential collisions with the environment (specifically, a ground plane). The use of a CSV file as an intermediary for storing and retrieving the poses, while potentially adding a slight overhead, could serve purposes such as logging, debugging, or providing a simple interface for external interaction. Key ROS 2 concepts employed in this code include nodes, parameters for configuration, TF for coordinate transformations, actions for commanding robot motion, messages for data exchange, and the powerful MoveIt 2 framework for motion planning and execution. The inclusion of safety checks, such as workspace boundaries and verification of the parsed pose, highlights the importance of robust design in robotics software. Further development could involve implementing a more sophisticated waypoint management system, incorporating the suggested random offset functionality, or adding mechanisms for feedback and error handling during the motion execution process. Overall, this code provides a solid foundation for understanding how to integrate ROS 2 with MoveIt 2 to achieve controlled robot movements based on sensor data and predefined parameters.

Table 1: ROS 2 Messages Used in the Code

Message Type	Package	Description
Pose	geometry_msgs.msg	Represents a position and orientation in 3D space.
PoseStamped	geometry_msgs.msg	Represents a Pose with a timestamp and frame ID.
MoveGroup.Goal	moveit_msgs.action	The goal message for the

		MoveGroup action, containing a MotionPlanRequest.
MotionPlanRequest	moveit_msgs.msg	Specifies the parameters and constraints for a motion planning request.
Constraints	moveit_msgs.msg	A list of constraints that the planned motion must satisfy.
PositionConstraint	moveit_msgs.msg	Defines a constraint on the position of a link.
OrientationConstraint	moveit_msgs.msg	Defines a constraint on the orientation of a link.
CollisionObject	moveit_msgs.msg	Represents an object in the robot's environment for collision checking.
PlanningScene	moveit_msgs.msg	Describes the state of the robot, its environment, and planning parameters.
SolidPrimitive	shape_msgs.msg	Represents basic geometric shapes like boxes, spheres, etc.

Works cited

1. Learn ROS 2 - Intro: Nodes - Hadabot, accessed March 30, 2025, https://www.hadabot.com/learn-ros2-intro-nodes.html?step=ros2-node-how-the-code-works-python
2. About ROS 2 client libraries, accessed March 30, 2025, https://docs.ros.org/en/foxy/Concepts/About-ROS-2-Client-Libraries.html
3. Writing a simple publisher and subscriber (Python) — ROS 2 ..., accessed March 30, 2025, https://docs.ros.org/en/rolling/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Py-Publisher-And-Subscriber.html
4. Client libraries — ROS 2 Documentation: Rolling documentation, accessed March 30, 2025, https://docs.ros.org/en/rolling/Concepts/Basic/About-Client-Libraries.html
5. About ROS 2 client libraries - daobook 0.0.1 文档, accessed March 30, 2025, https://daobook.github.io/ros2-docs/xin/Concepts/About-ROS-2-Client-Libraries.html
6. Client libraries — ROS 2 Documentation: Rolling documentation, accessed March 30, 2025, https://docs.ros.org/en/rolling/Concepts/About-ROS-2-Client-Libraries.html
7. ros2/rclpy: rclpy (ROS Client Library for Python) - GitHub, accessed March 30, 2025, https://github.com/ros2/rclpy