1. **Interface with VESC**: Implement a ROS 2 node or nodes that interface with the VESC. This node will take high-level commands (e.g., desired speed, steering angle) from other nodes in your system and translate them into signals that the VESC understands.
    a. ***Define Communication Interface***: First, you need to define how your ROS 2 node will communicate with the VESC. This could be through a serial interface like UART, CAN bus, or USB. For example, if you're using USB, you might communicate with the VESC as if it were a serial device.
        i.

        USB will be utilized

    b. ***Create ROS 2 Node***: Create a ROS 2 node specifically for interfacing with the VESC. This node will subscribe to high-level commands, such as desired speed and steering angle, from other nodes in your system. These commands could be published on specific ROS 2 topics.
        i. Python Code

```python
import rclpy
from rclpy.node import Node
from your_msgs.msg import SpeedCmd, SteeringCmd
from vesc_msgs.msg import VescCommand

class VescInterfaceNode(Node):
    def __init__(self):
        super().__init__('vesc_interface_node')
        self.speed_sub = self.create_subscription(SpeedCmd,
'desired_speed', self.speed_callback, 10)
        self.steering_sub = self.create_subscription(SteeringCmd,
'desired_steering', self.steering_callback, 10)
        self.vesc_pub = self.create_publisher(VescCommand,
'vesc_command', 10)

    def speed_callback(self, msg):
        vesc_command = VescCommand()
        vesc_command.speed = msg.speed
        self.vesc_pub.publish(vesc_command)

    def steering_callback(self, msg):
        vesc_command = VescCommand()
        vesc_command.steering_angle = msg.steering_angle
        self.vesc_pub.publish(vesc_command)
```

```
def main(args=None):
    rclpy.init(args=args)
    vesc_interface = VescInterfaceNode()
    rclpy.spin(vesc_interface)
    vesc_interface.destroy_node()
    rclpy.shutdown()

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

ii. Explanation:

1. We import the necessary modules from `rclpy` to work with ROS 2 and define the message types we'll be using (`SpeedCmd, SteeringCmd, and VescCommand`).
2. We define a class `VescInterfaceNode` that inherits from `rclpy.node.Node`.
3. In the `__init__` method of the class, we create subscriptions to the `desired_speed` and `desired_steering` topics and a publisher for the `vesc_command` topic.
4. We define callback functions `speed_callback` and `steering_callback` that are invoked whenever messages are received on the corresponding topics. These callbacks create a `VescCommand` message with the appropriate speed or steering angle and publish it on the `vesc_command` topic.
5. The main function initializes the ROS 2 node and spins it, allowing it to process incoming messages and callbacks.
6. We call the main function if the script is executed directly.

iii. You'll need to replace `your_msgs.msg` with the actual package and message types you're using for `SpeedCmd` and `SteeringCmd`. Similarly, replace `vesc_msgs.msg` with the package and message type for `VescCommand`.

c. ***Translate Commands***: In the VESC interface node, implement logic to translate these high-level commands into low-level signals that the VESC understands. For example, you might need to convert a desired speed into PWM (Pulse Width Modulation) signals for motor control and a desired steering angle into servo position commands.

i. Python Code (Modified from Above)

```
import rclpy
from rclpy.node import Node
from your_msgs.msg import SpeedCmd, SteeringCmd
from vesc_msgs.msg import VescCommand
```

```
class VescInterfaceNode(Node):
    def __init__(self):
        super().__init__('vesc_interface_node')
        self.speed_sub = self.create_subscription(SpeedCmd,
'desired_speed', self.speed_callback, 10)
        self.steering_sub = self.create_subscription(SteeringCmd,
'desired_steering', self.steering_callback, 10)
        self.vesc_pub = self.create_publisher(VescCommand,
'vesc_command', 10)

    def speed_callback(self, msg):
        vesc_command = VescCommand()
        # Convert desired speed to PWM signal (example)
        pwm_signal = self.convert_speed_to_pwm(msg.speed)
        vesc_command.pwm = pwm_signal
        self.vesc_pub.publish(vesc_command)

    def steering_callback(self, msg):
        vesc_command = VescCommand()
        # Convert desired steering angle to servo position command
(example)
        servo_position =
self.convert_steering_to_servo(msg.steering_angle)
        vesc_command.servo_position = servo_position
        self.vesc_pub.publish(vesc_command)

    def convert_speed_to_pwm(self, speed):
        # Your implementation to convert desired speed to PWM signal
        # Example implementation: Map speed to PWM range (1000-2000)
        pwm_signal = 1000 + (speed * 100)  # Example scaling factor
        return pwm_signal

    def convert_steering_to_servo(self, steering_angle):
        # Your implementation to convert desired steering angle to servo
position
        # Example implementation: Map steering angle to servo position
range (0-180 degrees)
        servo_position = int(steering_angle * (180 / 3.14))  # Example
scaling factor
        return servo_position

def main(args=None):
```

```
    rclpy.init(args=args)
    vesc_interface = VescInterfaceNode()
    rclpy.spin(vesc_interface)
    vesc_interface.destroy_node()
    rclpy.shutdown()

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

ii. Explanation:

1. In the **`speed_callback`** and **`steering_callback`** methods, we receive the desired speed and steering angle, respectively, as messages from their corresponding topics.
2. We then convert these high-level commands into low-level signals understood by the VESC. For example, we convert the desired speed into a PWM signal and the desired steering angle into a servo position command.
3. The **`convert_speed_to_pwm`** and **`convert_steering_to_servo`** methods are placeholders for your actual conversion logic. You'll need to implement these methods based on the specifications of your motor and steering mechanism.
4. Finally, we publish the VESC command message containing the translated low-level signal (PWM or servo position) on the **`vesc_command`** topic.

d. ***Send Commands to VESC***: Once you've translated the high-level commands, send them to the VESC using the chosen communication interface. This involves formatting the commands according to the communication protocol (e.g., UART messages) supported by the VESC and sending them over the appropriate communication channel.

i. Python Code (From above, modified further)

```
import rclpy
from rclpy.node import Node
from your_msgs.msg import SpeedCmd, SteeringCmd
from vesc_msgs.msg import VescCommand
import serial

class VescInterfaceNode(Node):
    def __init__(self):
        super().__init__('vesc_interface_node')
        self.speed_sub = self.create_subscription(SpeedCmd,
'desired_speed', self.speed_callback, 10)
        self.steering_sub = self.create_subscription(SteeringCmd,
```

```
'desired_steering', self.steering_callback, 10)
        self.serial_port = serial.Serial('/dev/ttyUSB0', 115200)  #
Adjust port and baudrate
        # Add setup for receiving feedback if needed

    def speed_callback(self, msg):
        # Serialize ROS 2 message (e.g., SpeedCmd) into command format
        command = 'set_speed {}'.format(msg.speed)

        # Send command to VESC over USB serial port
        self.serial_port.write(command.encode())

    def steering_callback(self, msg):
        # Serialize ROS 2 message (e.g., SteeringCmd) into command
format
        command = 'set_steering {}'.format(msg.steering_angle)

        # Send command to VESC over USB serial port
        self.serial_port.write(command.encode())

    # Implement methods for receiving and processing feedback if needed

    def run(self):
        rclpy.spin(self)

def main(args=None):
    rclpy.init(args=args)
    vesc_interface = VescInterfaceNode()
    vesc_interface.run()
    vesc_interface.destroy_node()
    rclpy.shutdown()

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

ii. Explanation:

1. We've imported the serial module to facilitate communication over the USB serial port.
2. In the **`__init__`** method, we initialize a serial port object **`serial_port`** with the appropriate parameters (**`/dev/ttyUSB0`** for the USB serial port and **`115200`** baud rate).

3. In the `speed_callback` and `steering_callback` methods, we serialize the ROS 2 messages (`SpeedCmd` and `SteeringCmd`) into command format and send them to the VESC over the USB serial port.
4. The run method is modified to use `rclpy.spin(self)` instead of `rospy.spin()` for compatibility with ROS 2.
5. We call the main function to start the ROS 2 node.

iii. Make sure to adjust the serial port (`/dev/ttyUSB0`) and baud rate (`115200`) according to your setup. Additionally, ensure that the VESC firmware is configured to communicate over USB and recognizes the commands sent by the Jetson Nano.

e. ***Receive Feedback (Optional)***: Optionally, you can implement logic in your VESC interface node to receive feedback from the VESC, such as motor speed or status information. This feedback can be useful for monitoring the state of the vehicle and implementing closed-loop control algorithms.

i. Python Code (From above, Further modified)

```
import rclpy
from rclpy.node import Node
from your_msgs.msg import SpeedCmd, SteeringCmd, VescFeedback
from vesc_msgs.msg import VescCommand
import serial

class VescInterfaceNode(Node):
    def __init__(self):
        super().__init__('vesc_interface_node')
        self.speed_sub = self.create_subscription(SpeedCmd,
'desired_speed', self.speed_callback, 10)
        self.steering_sub = self.create_subscription(SteeringCmd,
'desired_steering', self.steering_callback, 10)
        self.vesc_feedback_pub = self.create_publisher(VescFeedback,
'vesc_feedback', 10)
        self.serial_port = serial.Serial('/dev/ttyUSB0', 115200)  #
Adjust port and baudrate
        # Add setup for receiving feedback if needed

    def speed_callback(self, msg):
        # Serialize ROS 2 message (e.g., SpeedCmd) into command format
        command = 'set_speed {}'.format(msg.speed)

        # Send command to VESC over USB serial port
        self.serial_port.write(command.encode())
```

```
    def steering_callback(self, msg):
        # Serialize ROS 2 message (e.g., SteeringCmd) into command
format
        command = 'set_steering {}'.format(msg.steering_angle)

        # Send command to VESC over USB serial port
        self.serial_port.write(command.encode())

    # Implement methods for receiving and processing feedback
    def receive_feedback(self):
        while True:
            # Read feedback from VESC over USB serial port
            feedback_data = self.serial_port.readline().decode().strip()
            # Parse feedback data and publish as ROS 2 message
            vesc_feedback = self.parse_feedback(feedback_data)
            self.vesc_feedback_pub.publish(vesc_feedback)

    def parse_feedback(self, feedback_data):
        # Your implementation to parse feedback data and create a ROS 2
message
        # Example: Parse motor speed and status information
        # Split feedback_data and extract relevant information
        motor_speed = float(feedback_data.split(',')[0])
        status = feedback_data.split(',')[1]
        # Create VescFeedback message
        vesc_feedback = VescFeedback()
        vesc_feedback.motor_speed = motor_speed
        vesc_feedback.status = status
        return vesc_feedback

    def run(self):
        rclpy.spin(self)

def main(args=None):
    rclpy.init(args=args)
    vesc_interface = VescInterfaceNode()
    vesc_interface.receive_feedback()  # Start feedback reception loop
    vesc_interface.run()
    vesc_interface.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
```

```
    main()
```

ii. Explanation:

1. We've added a new publisher `vesc_feedback_pub` for publishing feedback from the VESC.
2. In the receive_feedback method, we continuously read feedback data from the VESC over the USB serial port and parse it into meaningful information.
3. The `parse_feedback` method parses the feedback data and creates a ROS 2 message of type `VescFeedback`.
4. The `receive_feedback` method runs in a loop to continuously receive feedback from the VESC and publish it as ROS 2 messages.
5. We call the `receive_feedback` method in the main function to start the feedback reception loop.

iii. Ensure that the feedback data received from the VESC is properly formatted and that the parsing logic in the `parse_feedback` method is adapted accordingly.

2. **Choose Communication Protocol**: Decide on the communication protocol to use between the Jetson and the VESC. This could be UART, CAN bus, or USB, depending on the capabilities of your VESC and the interfaces available on your Jetson.
   a. ***Physical Connection***: Connect a USB cable from a USB port on the Jetson to a USB port on the VESC. Ensure that both devices have USB ports and support USB communication.
   b. ***USB Communication Protocol***:
      i. USB communication involves endpoints, descriptors, and transfer types. You'll need to determine the appropriate USB communication protocol based on the capabilities of your Jetson and the VESC.
      ii. For simplicity, you may use USB CDC (Communications Device Class) for serial communication over USB. This allows the Jetson to appear as a virtual serial port to the VESC.
   c. ***Software Implementation***:
      i. On the Jetson:
         1. Implement USB CDC communication in your ROS 2 node that interfaces with the VESC. This involves opening the USB serial port, sending commands, and receiving responses.
         2. You may need to install USB CDC drivers on the Jetson if they're not already included in the operating system.

a. Check for Existing Drivers:
    i. Start by connecting your VESC 6 MkV to the Jetson Nano using a USB cable.
    ii. Once connected, check if the Jetson Nano recognizes the VESC as a USB device. You can do this by running the command `lsusb` in the terminal. Look for the VESC device in the list of connected USB devices.
b. Verify USB CDC Functionality:
    i. After connecting the VESC to the Jetson Nano, verify if the USB CDC functionality is working correctly.
    ii. Check if the VESC appears as a serial device in the `/dev` directory. You can do this by running the command ls `/dev/tty*` in the terminal and looking for a device resembling `/dev/ttyUSBX`, where `X` is a number.
c. Test Communication:
    i. Once you've identified the serial port corresponding to the VESC, you can test communication with the device.
    ii. Use a serial communication tool like minicom or screen to interact with the serial port. For example, you can use the command `sudo minicom -D /dev/ttyUSBX` to open a serial connection to the VESC (replace `/dev/ttyUSBX` with the actual serial port).
    iii. Send some test commands to the VESC and verify if you receive the expected responses. This will confirm that the USB CDC communication is functioning correctly.
d. Driver Installation (if necessary):
    i. In most cases, the Jetson Nano should automatically recognize the VESC as a USB CDC device and install the necessary drivers.
    ii. However, if the VESC is not recognized or if you encounter any driver-related issues, you may need to manually install drivers.
    iii. Check the VESC manufacturer's website or documentation for any specific driver requirements or instructions. They may provide Linux drivers or instructions for installing drivers on ARM-based platforms like the Jetson Nano.
e. Troubleshooting:

i. If you encounter any issues during the process, check the system logs (`dmesg`) for any error messages related to USB or serial communication.

ii. You can also search online forums or communities for assistance. The NVIDIA Jetson developer community and the VESC user community are valuable resources for troubleshooting and support.

ii. On the VESC:

1. Implement USB CDC communication on the VESC firmware or microcontroller. This allows the VESC to communicate with the Jetson as if it were a serial device.
2. Configure the VESC to recognize and respond to commands received over USB.
   a. ***The Following are Terminal Commands:***
   b. Configure USB Gadget on Jetson Nano

```
# Load USB gadget module
sudo modprobe libcomposite

# Create USB gadget configuration directory
sudo mkdir -p /sys/kernel/config/usb_gadget/my_usb_device
cd /sys/kernel/config/usb_gadget/my_usb_device

# Set USB gadget parameters
echo 0x1d6b > idVendor  # USB Vendor ID for Linux Foundation
echo 0x0104 > idProduct # USB Product ID for CDC ACM device
echo 0x0100 > bcdDevice # USB device version

# Create USB gadget functions (e.g., CDC ACM)
sudo mkdir -p functions/acm.usb0

# Set USB gadget configuration
echo 1 > functions/acm.usb0/enable

# Create USB gadget configuration file
sudo nano
/sys/kernel/config/usb_gadget/my_usb_device/configs/c.1/stri
ngs/0x409/configuration

# Add configuration string (e.g., "CDC ACM Configuration")
# Save and exit the text editor

# Enable USB gadget configuration
```

```
echo 250 > configs/c.1/MaxPower

# Create USB gadget instance
sudo mkdir -p /dev/usb-gadget
sudo ln -s /sys/kernel/config/usb_gadget/my_usb_device
/dev/usb-gadget/my_usb_device
```

c. Implement USB CDC Communication on VESC Firmware

```
# Download VESC firmware source code
git clone https://github.com/vedderb/bldc.git
cd bldc

# Configure VESC firmware build
make clean
make

# Flash VESC firmware to the VESC 6 MkV
# Replace /dev/ttyUSB0 with the appropriate serial port
sudo make upload /dev/ttyUSB0
```

d. Testing USB CDC Communication

```
# Test USB CDC communication on Jetson Nano
# Replace /dev/ttyACM0 with the appropriate serial port
minicom -D /dev/ttyACM0

# Send and receive data over the virtual serial port
# Use Ctrl+A, Z to access minicom menu for configuration
```

e. Integrating with ROS 2

```
# Install ROS 2 on Jetson Nano (if not already installed)
# Follow official ROS 2 installation instructions for ARM64
architecture

# Create ROS 2 workspace
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws/src

# Clone ROS 2 packages for VESC communication
git clone <your_ros2_packages_repo_url>

# Build ROS 2 packages
cd ~/ros2_ws
```

```
colcon build

# Source ROS 2 setup script
source install/setup.bash

# Run ROS 2 nodes for VESC communication
ros2 run <your_ros2_package_name> <your_ros2_node_name>
```

f. These commands provide a basic outline for configuring USB gadget on the Jetson Nano, implementing USB CDC communication on the VESC firmware, testing USB CDC communication, and integrating with ROS 2. Please replace placeholders like **<your_ros2_packages_repo_url>** and **<your_ros2_package_name>** with actual values specific to your project.

3. **Implement Communication Protocol**: Implement the chosen communication protocol in your ROS 2 node that interfaces with the VESC. This involves sending commands to the VESC and receiving feedback, such as motor speed or status information.
   a. ***Sending Commands to the VESC***:
      i. Define Message Types: Define ROS 2 message types for the commands you want to send to the VESC. These messages could include commands for setting the speed, steering angle, or other parameters.
         1. Create a new ROS 2 package specifically for defining custom message types.
         2. Use the following command to create the package (Terminal/Bash)

```
ros2 pkg create <message_package_name> --build-type ament_python
--dependencies rclpy
```

         3. Define custom message types. For example, you can define messages for setting the speed and steering angle (Python):

```
# Speed command message
# File: speed_cmd.msg
float64 speed

# Steering command message
# File: steering_cmd.msg
float64 steering_angle
```

         4. Build the ROS 2 message package to generate Python code for the custom message types:

```
cd <path_to_message_package>
```

```
colcon build
```

5. After building the message package, source the ROS 2 setup script to make the custom message types available:

```
source <path_to_ros2_ws>/install/setup.bash
```

6. Once you've defined the ROS 2 message types, you can use them in your ROS 2 nodes for communicating with the VESC. These message types will allow you to publish commands, such as desired speed and steering angle, on specific ROS 2 topics, which can then be subscribed to by the node responsible for interfacing with the VESC

ii. Publish Commands: In your ROS 2 node, publish these command messages on the appropriate topics whenever you want to control the vehicle. For example, you might publish a **`SetSpeed`** message on the **`desired_speed`** topic to control the vehicle's speed.

1. Within your ROS 2 node, import the necessary modules to work with ROS 2 and your custom message types.
2. Create a publisher object that will publish messages on the appropriate topic. For example, if you want to control the vehicle's speed, you'll publish **`SetSpeed`** messages on the **`desired_speed`** topic.
3. Populate instances of your custom message type with the desired command values (e.g., desired speed) and publish them at the appropriate rat
4. Example Python Code:

```
import rclpy
from your_custom_msgs.msg import SetSpeed  # Import your custom
message type

def publish_commands():
    # Initialize ROS 2 node
    rclpy.init()
    node = rclpy.create_node('command_publisher')

    # Create publisher for SetSpeed messages
    publisher = node.create_publisher(SetSpeed, 'desired_speed',
10)

    # Create SetSpeed message and populate with desired speed
value
    speed_msg = SetSpeed()
```

```
    speed_msg.speed = 0.5  # Example value, replace with your
desired speed

    # Publish SetSpeed message
    node.get_logger().info('Publishing command:
{}'.format(speed_msg.speed))
    publisher.publish(speed_msg)

    # Spin ROS 2 node
    rclpy.spin(node)

    # Clean up
    node.destroy_node()
    rclpy.shutdown()

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

a. We import rclpy for working with ROS 2 in Python.
b. We import our custom message type **`SetSpeed`** from the appropriate package (**`your_custom_msgs.msg`**).
c. We define a function **`publish_commands()`** to encapsulate the publishing logic.
d. Inside this function, we initialize the ROS 2 node, create a publisher for the **`SetSpeed`** messages on the **`desired_speed`** topic, create a **`SetSpeed`** message, populate it with a desired speed value, publish the message, and then spin the node to keep it active.
e. Finally, we call **`publish_commands()`** to execute the publishing logic when the script is run.
f. Make sure to replace '**`your_custom_msgs.msg`**' with the actual package name where your custom message types are defined, and adjust the message fields and values according to your specific requirements.

iii. Serialize Messages: Serialize the ROS 2 message into a format that can be transmitted over USB. This may involve converting the message into a byte array or string representation.

1. In the provided code snippet for publishing commands, serialization is implicitly handled by ROS 2. When you publish a message using ROS 2's publisher, it takes care of serializing the message into a format suitable for transmission over the ROS

2 middleware, which can include serialization to binary format for transport over USB.

2. Therefore, you don't need to explicitly handle serialization in the code snippet provided earlier. ROS 2 takes care of serialization and deserialization transparently as part of its communication mechanisms

iv. Send Commands: Send the serialized command data to the VESC over USB using the appropriate USB communication interface. This typically involves writing the data to the USB serial port.

1. Open USB Serial Port: Use appropriate libraries (e.g., `pyserial` in Python) to open the USB serial port on the Jetson Nano. This establishes a connection to the VESC.
2. Serialize Commands: As mentioned earlier, ROS 2 handles serialization of messages when publishing. So, you'll have the serialized command data ready to be sent.
3. Write Data to Serial Port: Once the USB serial port is open and you have the serialized command data, you can write this data to the serial port. This sends the commands to the VESC.
4. Simplified example in Python using `pyserial`:

```
import serial

# Open USB serial port
ser = serial.Serial('/dev/ttyUSB0', 115200)  # Adjust port and
baudrate

# Serialized command data (replace with actual serialized data)
serialized_data = b'your_serialized_data_here'

# Send serialized data to VESC over USB serial port
ser.write(serialized_data)

# Close serial port when done
ser.close()
```

   a. We use the `serial.Serial()` function from the `pyserial` library to open the USB serial port (`/dev/ttyUSB0` is an example; you may need to adjust it based on your system).
   b. We assume that `serialized_data` contains the serialized command data obtained from ROS 2.

c. We use the `write()` method of the serial object (`ser`) to send the serialized data to the VESC over the USB serial port.
d. Finally, we close the serial port using the `close()` method.

5. You need to integrate this code into your ROS 2 node where you're handling the command publishing. When you want to send commands to the VESC, you'll call this code to send the serialized command data over USB.

b. ***Receiving Feedback from the VESC***:

i. Receive Data: Implement logic in your ROS 2 node to receive data from the VESC over USB. This could include motor speed, status information, or any other feedback relevant to your application.

1. Open USB Serial Port: Similar to sending commands, you'll need to open the USB serial port on the Jetson Nano to establish a connection to the VESC.
2. Read Data from Serial Port: Continuously read data from the USB serial port to receive feedback from the VESC. You may need to define a protocol for the data exchange between the Jetson Nano and the VESC to interpret the received data correctly.
3. Parse Received Data: Parse the received data to extract relevant information, such as motor speed, status information, or any other feedback provided by the VESC.
4. Process and Publish Feedback: Once you've parsed the received data and extracted the relevant information, you can process it as needed and publish it as ROS 2 messages on appropriate topics within your ROS 2 node. Other nodes in your ROS 2 system can then subscribe to these topics to access the feedback data.
5. Example in Python using pyserial for reading data from the serial port:

```
import serial

# Open USB serial port
ser = serial.Serial('/dev/ttyUSB0', 115200)  # Adjust port and
baudrate

# Continuously read data from serial port
while True:
    # Read data from serial port
    data = ser.readline().strip()  # Adjust read size and format
as needed

    # Parse received data and process as needed
```

```
    # Example: Print received data
    print("Received data:", data)

# Close serial port when done (this won't be reached in the
example)
ser.close()
```

a. We continuously read data from the USB serial port using the `readline()` method of the serial object (`ser`).
b. The received data is then processed as needed. You would replace the print statement with logic to parse and handle the received data according to your application requirements.
c. The `while True` loop ensures that the code continuously listens for and processes incoming data.
d. The serial port is closed when the program exits (although in this example, the loop is infinite so it won't reach the close statement)

ii. Parse Data: Parse the received data to extract the relevant information. Depending on the communication protocol and data format used by the VESC, this may involve decoding byte sequences or parsing structured messages.

1. Understand the Data Format: Determine how the VESC encodes the feedback data. This could involve consulting the VESC documentation or inspecting the data received from the VESC.
2. Define Parsing Logic: Based on the data format, write parsing logic in your ROS 2 node to extract the relevant information. This could involve techniques such as splitting strings, decoding binary data, or using regular expressions.
3. Extract Relevant Information: Parse the received data to extract the relevant information, such as motor speed, status flags, or any other feedback provided by the VESC.
4. Publish Parsed Feedback: Once you've extracted the relevant information, publish it as ROS 2 messages on appropriate topics. This allows other nodes in your ROS 2 system to subscribe to and process the feedback as needed.
5. Example of parsing feedback data received over a serial connection:

```
import serial

# Open serial port
```

```
ser = serial.Serial('/dev/ttyUSB0', 115200)

# Continuously read and parse data
while True:
    # Read data from serial port
    data = ser.readline().strip()  # Adjust read size and format
as needed

    # Parse received data
    parsed_data = parse_data(data)  # Implement parsing logic

    # Publish parsed data as ROS 2 message
    publish_parsed_data(parsed_data)  # Implement publishing logic

# Close serial port when done
ser.close()
```

6. In this example, **parse_data()** is a placeholder for your parsing logic, where you extract relevant information from the received data. Similarly, **publish_parsed_data()** is a placeholder for your publishing logic, where you publish the parsed data as ROS 2 messages on appropriate topics

iii. Publish Feedback: Publish the parsed feedback data as ROS 2 messages on appropriate topics. This allows other nodes in your ROS 2 system to subscribe to and process the feedback as needed.

1. Define ROS 2 Message Types: First, define ROS 2 message types to represent the parsed feedback data. These message types should capture the relevant information extracted from the feedback data. You can create custom message types using ROS 2 message definitions.
2. Modify Parsing Logic: Update the parsing logic in your ROS 2 node to extract the relevant information from the received data. Once parsed, store this information in variables or data structures.
3. Publish Parsed Feedback: After parsing the feedback data, publish it as ROS 2 messages on appropriate topics. Use the ROS 2 publishing API to create publishers for each topic and publish the parsed feedback data as messages.
4. Subscribe to Published Topics: If other nodes in your ROS 2 system need to process the feedback data, they can subscribe to the topics where the parsed feedback messages are published. These nodes can then receive and process the feedback data as needed.

5. Example of how you might implement these steps in your ROS 2 node:

```
import rclpy
from your_custom_msgs.msg import ParsedFeedback  # Import your
custom message type
import serial

def parse_feedback(data):
    # Implement parsing logic to extract relevant information from
data
    parsed_data = ...  # Parse data and store parsed information

    return parsed_data

def publish_feedback(parsed_data, feedback_publisher):
    # Create a ROS 2 message with parsed feedback data
    feedback_msg = ParsedFeedback()
    feedback_msg.field1 = parsed_data.field1
    feedback_msg.field2 = parsed_data.field2
    # Populate other fields as needed

    # Publish the message on the feedback topic
    feedback_publisher.publish(feedback_msg)

def main():
    rclpy.init()

    # Create a node
    node = rclpy.create_node('vesc_interface_node')

    # Create a publisher for the parsed feedback messages
    feedback_publisher = node.create_publisher(ParsedFeedback,
'parsed_feedback_topic', 10)

    # Open serial port for communication with VESC
    ser = serial.Serial('/dev/ttyUSB0', 115200)

    # Continuously read and parse feedback data
    while rclpy.ok():
        data = ser.readline().strip()  # Read data from serial
port
        parsed_data = parse_feedback(data)  # Parse feedback data
        publish_feedback(parsed_data, feedback_publisher)  #
```

```
Publish parsed feedback
        rclpy.spin_once(node)

    # Close serial port when done
    ser.close()
    rclpy.shutdown()

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

a. `parse_feedback()` function is responsible for parsing the received feedback data and returning the parsed information.
b. `publish_feedback()` function creates a ROS 2 message containing the parsed feedback data and publishes it on the appropriate topic using the feedback publisher.
c. `The main()` function initializes the ROS 2 node, creates a publisher for parsed feedback messages, opens a serial port for communication with the VESC, continuously reads and parses feedback data, and publishes parsed feedback messages.
d. You'll need to replace `your_custom_msgs.msg` with the actual package name and message type containing the parsed feedback data. Additionally, customize the parsing logic to extract the relevant information from the received feedback data

4. **Test and Debug**: Test your ROS 2 nodes on the Jetson to ensure they can effectively communicate with the VESC. Debug any issues that arise during testing, such as incorrect command signals or communication failures.
    a. ***Unit Testing***:
        i. Write unit tests for individual components of your ROS 2 nodes, such as message serialization, command processing, and feedback parsing.
        ii. Use mock objects or simulation environments to simulate the behavior of the VESC and test your nodes in isolation from the actual hardware.
    b. ***Integration Testing***:
        i. Test the integration of your ROS 2 nodes with the VESC in a controlled environment.
        ii. Use a test setup with the actual hardware connected, but in a controlled environment where you can easily monitor and troubleshoot issues.
    c. ***Functional Testing***:

   i. Conduct functional tests to verify that your ROS 2 nodes can effectively control the vehicle's speed, steering, and other parameters.
   ii. Test various scenarios and edge cases to ensure robustness and reliability of your system.

d. ***Debugging Techniques***:
   i. Monitor ROS 2 topics and messages using tools like rostopic and rqt.
   ii. Use logging and debugging statements in your ROS 2 nodes to track the flow of data and identify potential issues.
   iii. Check for error messages and exceptions raised during operation and investigate their causes.
   iv. Use hardware debugging tools such as oscilloscopes or logic analyzers to monitor the communication between the Jetson and the VESC.

e. ***Troubleshooting Common Issues***:
   i. Check the physical connections between the Jetson and the VESC to ensure they are properly connected.
   ii. Verify that the USB serial port is configured correctly and that the baud rate matches between the Jetson and the VESC.
   iii. Check for compatibility issues between the USB drivers on the Jetson and the VESC firmware.
   iv. Ensure that the VESC is powered on and initialized correctly before attempting communication.

f. ***Iterative Development***:
   i. Iterate on your ROS 2 nodes and communication protocol based on feedback from testing and debugging.
   ii. Refactor and optimize your code as needed to improve performance and reliability.

g. ***Example Debugging Workflow***:
   i. Deploy your ROS 2 nodes to the Jetson and start the system.
   ii. Monitor the ROS 2 topics related to communication with the VESC using rostopic.
   iii. Send test commands to the VESC and observe the response.
   iv. Check for any error messages or unexpected behavior in the ROS 2 node logs.
   v. Use debugging statements to track the flow of data through your ROS 2 nodes and identify potential issues.
   vi. Investigate and troubleshoot any issues identified during testing, making necessary adjustments to your code or configuration.

vii. Repeat the testing and debugging process until your system operates reliably and meets your requirements.

5. **Integrate with Vehicle Control System**: Integrate the ROS 2 nodes responsible for communicating with the VESC into your overall vehicle control system. Ensure that they work seamlessly with other components, such as perception and planning modules.
    a. ***Define Interfaces***:
        i. Clearly define the interfaces between different components of your vehicle control system, including perception, planning, and VESC communication.
        ii. Determine the ROS 2 topics, services, and actions that will be used for communication between these components.
        iii. Document the message types and data formats exchanged between components to facilitate integration.
    b. ***Implement Interoperability***:
        i. Implement ROS 2 nodes for perception and planning modules that publish relevant information, such as obstacle detection, localization, and trajectory planning, on designated topics.
        ii. Ensure that the ROS 2 nodes responsible for communicating with the VESC can subscribe to these topics and incorporate the information into their control algorithms.
    c. ***Coordinate Actions***:
        i. Coordinate actions between different components of the vehicle control system to achieve desired behavior.
        ii. For example, the perception module might detect obstacles and publish this information on a `obstacle_detection` topic. The planning module can then use this information to generate collision-free trajectories, which are sent to the VESC communication node for execution.
    d. ***Error Handling and Recovery***:
        i. Implement error handling and recovery mechanisms to handle unexpected situations, such as communication failures or sensor errors.
        ii. Define strategies for gracefully recovering from errors and ensuring the safety of the vehicle and its surroundings.
    e. 5. Simulation and Testing:
        i. Use simulation environments to test the integration of your vehicle control system components in a virtual environment.
        ii. Verify that the perception, planning, and VESC communication modules interact correctly and produce the desired behavior in simulated scenarios.

f. ***Real-world Testing***:
    i. Conduct real-world testing to validate the performance of your integrated vehicle control system in real-world conditions.
    ii. Collect data and evaluate the system's performance against predefined metrics, such as accuracy, robustness, and safety.
g. ***Iterative Development***:
    i. Iterate on the integration of your vehicle control system components based on feedback from testing and real-world deployment.
    ii. Continuously improve and refine the integration to enhance the overall performance and reliability of the system.
h. ***Example Integration Scenario***:
    i. The perception module detects obstacles using onboard sensors and publishes obstacle positions on an `obstacle_detection` topic.
    ii. The planning module generates collision-free trajectories based on the detected obstacles and publishes them on a `trajectory_planning` topic.
    iii. The VESC communication node subscribes to the `trajectory_planning` topic, receives trajectory commands, and executes them using the VESC to control the vehicle's speed and steering.

6. **Calibration and Tuning**: Calibrate and tune the control parameters of your VESC to achieve desired performance characteristics, such as smooth acceleration and responsive steering.
    a. ***Understand VESC Parameters***:
        i. Familiarize yourself with the various control parameters available in the VESC firmware or configuration software.
        ii. Understand the purpose and effect of each parameter on the behavior of your vehicle, such as motor control, throttle response, and braking.
    b. ***Motor Configuration***:
        i. Configure the VESC settings to match the specifications of your motor, including motor type (BLDC, FOC), pole pairs, phase resistance, and inductance.
        ii. Ensure that the motor parameters are accurately entered to enable proper motor control and efficient operation.
    c. ***Current and Voltage Limits***:
        i. Set appropriate current and voltage limits to protect your motor, battery, and other electrical components from damage.

   ii. Determine the maximum safe current and voltage levels based on the specifications of your motor and battery, and configure the VESC accordingly.

d. ***Throttle and Brake Calibration***:
   i. Calibrate the throttle and brake inputs to ensure smooth and precise control over acceleration and deceleration.
   ii. Perform throttle calibration to establish the mapping between throttle input and motor output, ensuring consistent and linear response.
   iii. Calibrate the brake settings to adjust the braking force and regenerative braking behavior to suit your preferences and requirements.

e. ***PID Tuning***:
   i. Fine-tune the PID (Proportional-Integral-Derivative) control parameters to achieve stable and responsive motor control.
   ii. Start by adjusting the proportional, integral, and derivative gains to achieve desired performance characteristics, such as quick response to throttle input and minimal overshoot.
   iii. Use iterative testing and observation to refine the PID parameters until you achieve optimal performance without oscillations or instability.

f. ***Sensor Calibration***:
   i. Calibrate any sensors used by the VESC, such as temperature sensors or current sensors, to ensure accurate measurement and monitoring of critical parameters.
   ii. Follow the manufacturer's instructions for sensor calibration procedures and perform calibration as needed to maintain accuracy and reliability.

g. ***Test and Iteration***:
   i. Test the performance of your vehicle under various operating conditions, such as different speeds, loads, and terrain.
   ii. Monitor the behavior of the VESC and observe any anomalies or areas for improvement.
   iii. Iterate on the calibration and tuning process based on test results and feedback, making incremental adjustments to fine-tune the performance of your vehicle.

h. ***Safety Considerations***:
   i. Prioritize safety throughout the calibration and tuning process to prevent accidents or damage to equipment.
   ii. Conduct testing in a controlled environment with appropriate safety measures in place, such as barriers, safety gear, and emergency stop mechanisms.
   iii. Monitor temperature and voltage levels during testing to ensure that they remain within safe limits.

Python File Breakdown

---

1. **vesc_interface_with_feedback.py**

```
import rclpy
from rclpy.node import Node
from your_msgs.msg import SpeedCmd, SteeringCmd, VescFeedback
from vesc_msgs.msg import VescCommand
import serial

class VescInterfaceNode(Node):
    def __init__(self):
        super().__init__('vesc_interface_node')
        self.speed_sub = self.create_subscription(SpeedCmd, 'desired_speed',
self.speed_callback, 10)
        self.steering_sub = self.create_subscription(SteeringCmd,
'desired_steering', self.steering_callback, 10)
        self.vesc_feedback_pub = self.create_publisher(VescFeedback,
'vesc_feedback', 10)
        self.serial_port = serial.Serial('/dev/ttyUSB0', 115200)  # Adjust port and
baudrate
        # Add setup for receiving feedback if needed

    def speed_callback(self, msg):
        # Serialize ROS 2 message (e.g., SpeedCmd) into command format
        command = 'set_speed {}'.format(msg.speed)

        # Send command to VESC over USB serial port
        self.serial_port.write(command.encode())

    def steering_callback(self, msg):
        # Serialize ROS 2 message (e.g., SteeringCmd) into command format
        command = 'set_steering {}'.format(msg.steering_angle)

        # Send command to VESC over USB serial port
        self.serial_port.write(command.encode())

    # Implement methods for receiving and processing feedback
    def receive_feedback(self):
        while True:
            # Read feedback from VESC over USB serial port
```

```
            feedback_data = self.serial_port.readline().decode().strip()
            # Parse feedback data and publish as ROS 2 message
            vesc_feedback = self.parse_feedback(feedback_data)
            self.vesc_feedback_pub.publish(vesc_feedback)

    def parse_feedback(self, feedback_data):
        # Your implementation to parse feedback data and create a ROS 2 message
        # Example: Parse motor speed and status information
        # Split feedback_data and extract relevant information
        motor_speed = float(feedback_data.split(',')[0])
        status = feedback_data.split(',')[1]
        # Create VescFeedback message
        vesc_feedback = VescFeedback()
        vesc_feedback.motor_speed = motor_speed
        vesc_feedback.status = status
        return vesc_feedback

    def run(self):
        rclpy.spin(self)

def main(args=None):
    rclpy.init(args=args)
    vesc_interface = VescInterfaceNode()
    vesc_interface.receive_feedback()  # Start feedback reception loop
    vesc_interface.run()
    vesc_interface.destroy_node()
    rclpy.shutdown()

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

2. **command_publisher.py**

```
import rclpy
from your_custom_msgs.msg import SetSpeed  # Import your custom message type

def publish_commands():
    # Initialize ROS 2 node
    rclpy.init()
    node = rclpy.create_node('command_publisher')

    # Create publisher for SetSpeed messages
    publisher = node.create_publisher(SetSpeed, 'desired_speed', 10)
```

```
    # Create SetSpeed message and populate with desired speed value
    speed_msg = SetSpeed()
    speed_msg.speed = 0.5  # Example value, replace with your desired speed

    # Publish SetSpeed message
    node.get_logger().info('Publishing command: {}'.format(speed_msg.speed))
    publisher.publish(speed_msg)

    # Spin ROS 2 node
    rclpy.spin(node)

    # Clean up
    node.destroy_node()
    rclpy.shutdown()

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

3. **send_serial_data_to_vesc.py**

```
import serial

# Open USB serial port
ser = serial.Serial('/dev/ttyUSB0', 115200)  # Adjust port and baudrate

# Serialized command data (replace with actual serialized data)
serialized_data = b'your_serialized_data_here'

# Send serialized data to VESC over USB serial port
ser.write(serialized_data)

# Close serial port when done
ser.close()
```

4. **serial_data_render.py**

```
import serial

# Open USB serial port
ser = serial.Serial('/dev/ttyUSB0', 115200)  # Adjust port and baudrate

# Continuously read data from serial port
while True:
```

```
    # Read data from serial port
    data = ser.readline().strip()  # Adjust read size and format as needed

    # Parse received data and process as needed
    # Example: Print received data
    print("Received data:", data)

# Close serial port when done (this won't be reached in the example)
ser.close()
```

5.

```

```

6.