

# COMP0213 Coursework: Using LLMs to simplify human-robot interactions

Andrew Sanmori-Gwozdz<sup>1</sup> and Hassan Shahzad<sup>1</sup>

<sup>1</sup>Department of Computer Science, University College London

## ABSTRACT

The rigidity of robots is the first thing people notice. In front of them lies a miracle in engineering and a product of every advancement in engineering and computer science over the last 80 years and yet their movement is too rigid; too robotic. We can easily distinguish between creatures and robots even though both can appear to look similar and be of similar shapes. Why is this the case? It isn't that robots lack enough degrees of freedom or that their hardware isn't up to date; these problems have been solved countless times decades ago by great mechanical engineers. What lies at the heart of the problem is a lack of advanced software to improve human robot interaction and help robots interact with the world. Recent advancements in LLM's have allowed for powerful language models to be run in the cloud and be utilised to better bridge the gap between man and machine. Enabling the future of robots to longer appear robotic in nature but lifelike, excelling in their ability to navigate the world in a manner we are used to

Our paper proposes combining this new concept of LLM agents; those will the ability to perform tool calling in the right order and sequence to complete a users query with robust SLAM techniques to provide a low computational cost, accurate and robust means of transport and interaction for any ROS2 robot. A scalable and reproducible solution which we aim to offer can be implemented across numerous autonomous robots to bring the robotics era one step closer

Keywords: LLM, LangChain, LiveKit, tool calling, SLAM

## CONTENTS

<b>1 Methods</b>	<b>2</b>
1.1 First approach: LangChain . . . . .	2
Usage of tools with LangChain • Tools used • OOP structure	
1.2 Second approach: LiveKit . . . . .	4
Usage of tools with LiveKit • Tools used	
1.3 Simulation . . . . .	6
Modeling the Robots • Simulating the Robot • The relevance of ROS2 control	
<b>2 Results</b>	<b>11</b>
2.1 LangChain and LiveKit Comparison . . . . .	11
Results for the LangChain LLM • Results for the LiveKit LLM • Tool chaining	
2.2 Simulation results . . . . .	17
Mapping the robot with slam_toolbox • Localizing with AMCL • Navigating inside the world • Using the Nav2 Simple Commander API • Additional Services programmed	
2.3 Overall Results with high level tools and ROS2 bindings . . . . .	24
Tool to take an image and save it • Tool to describe the contents of an image • Robot instructions	
<b>3 Full package Layout for a24</b>	<b>29</b>
3.1 Nodes . . . . .	29
3.2 Launch Files . . . . .	30
3.3 Configuration Files . . . . .	30
3.4 Maps and World Files . . . . .	30
3.5 URDF and Mesh Files . . . . .	31
3.6 Visualizations . . . . .	31

3.7 Testing . . . . .	31
3.8 Miscellaneous . . . . .	31
<b>4 Discussion</b>	<b>31</b>
<b>5 Contributions</b>	<b>31</b>

## INTRODUCTION

Our paper is divided into 2 overall sections. The first diving into how LLMs are capable of transforming human queries into robot instructions, where we describe 2 different approaches we have taken: Langchain and LiveKit, the first providing robust and reliable results but text based, and the second a method of using cutting edge webRTC and LLM to get results enabling speech to speech interaction with the robot. Section 1 will also feature an overview of our robots simulated in Gazebo and using ros2.

Section 2 shows the results we have obtained from our two approaches to LLM agents: LangChain and LiveKit, interacting with a Gazebo environment of a home equipped with SLAM and mapping to go from point A to B along with numerous other features like image recognition.

## 1 METHODS

We have taken two different approaches to human-robot interactions with LLMs and tool-calling. The first one uses LangChain, a Python library that helps in developing projects that use LLMs. That would mean "text to text" interactions. The second uses LiveKit, an open-source platform for user-AI interactions. That would allow for "speech to speech" interactions.

### 1.1 First approach: LangChain

We have first made use of the [LangChain](#) python library, which allows to add tools to a given LLM (we used openAI's 'gpt-4o' model). The interaction with the user is text to text: the user writes down a query, which is passed to the LLM, and the LLM then calls the correct tools to complete the query, as well as answer the user.

#### 1.1.1 Usage of tools with LangChain

Tools are functions that were written by a human prior to the program running, and that the LLM not only has access to, but also understands (thanks to a provided description). In this first approach, we use *LangChain* to connect tools to our LLMs (which we use openAI for).

```
@tool
def get_time() -> int:
    """Gets the number of seconds that passed since the first of january 1970"""
    seconds = time.time()
    return seconds
```

**Figure 1.** Simple example of a LangChain tool

Fig.1 shows an example for a LangChain tool. This tool outputs a timestamp and takes no inputs. The docstring is what the LLM uses to understand what the tool does, so clear explanations are very important.

#### 1.1.2 Tools used

Our LangChain code uses 2 files for its tools. The first one, *ros\_tools.py*, is a collection of general use ros2 tools that allow the LLM to execute a ros2 command, gather a list of all active topics, echo a given topic, etc. Those tools are not ours, they were copied from the [ROSA project](#) (which aimed to give an LLM ros and ros2 tools to control a robot).

The second file, *ros\_robot\_tools.py*, is a collection of tools (that we made ourselves) which we estimated the LLM needed to perform the set of actions we wanted it to perform. There are tools to give the robot any linear or angular velocity, stop the robot, or gather data on the current state of the robot.

```

@tool
def give_robot_velocity(x_linear: float = 0.0, y_linear: float = 0.0,
                       z_linear: float = 0.0, x_angular: float = 0.0,
                       y_angular: float = 0.0, z_angular: float = 0.0,
                       rate: float = 1) -> dict:
    """
    Make the robot move with the given x,y,z linear velocities and x,y,z angular velocities

    :param x_linear: the linear velocity along the x axis that the robot will get.
    :param y_linear: the linear velocity along the y axis that the robot will get.
    :param z_linear: the linear velocity along the z axis that the robot will get.
    :param x_angular: the angular velocity along the x axis that the robot will get.
    :param y_angular: the angular velocity along the y axis that the robot will get.
    :param z_angular: the angular velocity along the z axis that the robot will get.
    :param rate: the rate, in Hz, at which this information will be passed to the robot.
    """
    cmd = f'ros2 topic pub {(" -r " + str(rate)) if rate != 1 else ""} + \
          f' /cmd_vel {geometry_msgs/msg/Twist "{{linear: {{x: {x_linear}, y: \
          {y_linear}, z: {z_linear}}}}, angular: {{x: {x_angular}, y: \
          {y_angular}, z: {z_angular}}}}}" --once'

    success, output = execute_ros_command(cmd)

    if not success:
        return {"error": output}

    return {'output': output} # output of the terminal

```

**Figure 2.** Tool for giving the robot any linear or angular velocity

```

@tool
def ros2_topic_echo_info(
    topic: str,
    timeout: float = 1.0,
) -> dict:
    """
    Echoes the contents of a specific ROS2 topic, and always returns the message.

    :param topic: The name of the ROS topic to echo.
    :param timeout: Max time to wait for a message before timing out.

    :note: Do not use this tool if the number of messages is large.
    | This will cause the response to be too large and may cause the tool to fail.
    """
    cmd = f"ros2 topic echo {topic} --once --spin-time {timeout}"

    success, output = execute_ros_command(cmd)

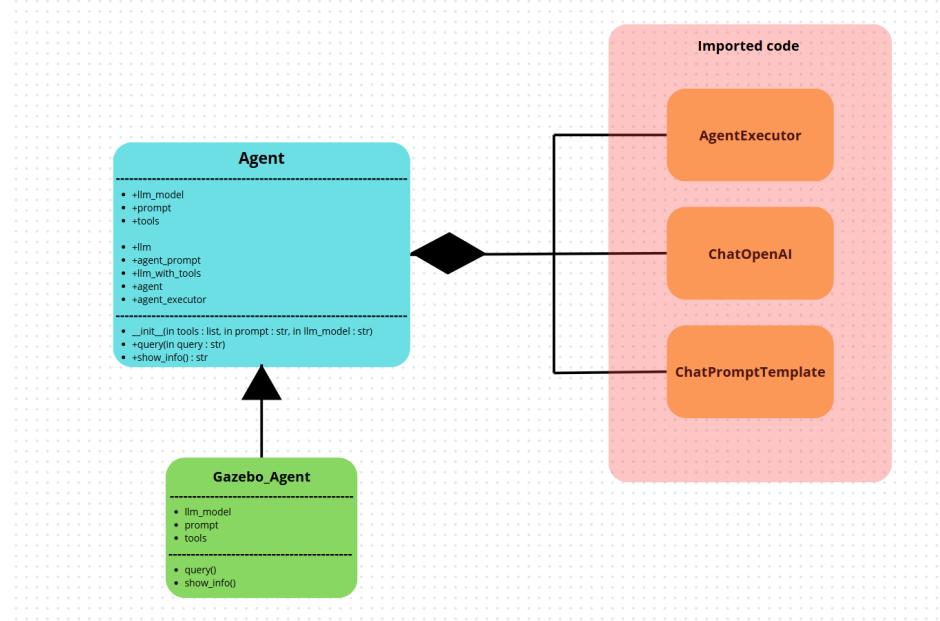
    if success:
        return {"echoes": output}

    return {"error": output}

```

**Figure 3.** Tool for listening to any active ros2 topic

### 1.1.3 OOP structure



**Figure 4.** UML diagram of the LangChain agent

In Fig. 4, the 'imported code' section corresponds to classes belonging to external libraries: *LangChain* and *LangChain-openai*.

Class *Gazebo\_agent* inherits from *Agent* and is practically identical, except for its modified prompt (where it is given a detailed description of the environment it is in, such as what information each ros2 topic can give), and its modified tools (it is given the tools that correspond to our Gazebo simulation).

Thus, *Gazebo\_agent* is the agent that was created to specifically interact with our Gazebo simulation, and *Agent* is more of a generic agent. *Agent* can be used on different Gazebo simulations, provided its prompt is changed to a description of the environment, and its tools are adequate to the tasks it is to perform.

```

gazebo_prompt = """
You are a robot in a gazebo simulation. You have access to topics to know
the current state of the robot.
Prioritize ros2_topic_echo_info over ros2_topic_echo tool.
Here are some of the topics you have access to:
'/camera/camera_info' gives you information on the calibration of the camera,
'/camera/depth/camera_info' gives you information on the calibration of the depth camera,
'/camera/depth/image_raw' gives you each pixel value for the image of the depth camera,
'/clock' tells you how much time has passed inside the simulation,
'/joint_states' gives you the current angle of each of the wheels,
'/odom' gives you odometry of the robot: under 'pose:', 'pose:', 'position:', you can find the x y z position of the robot,
'/scan' gives you information from the 2D ladar you have.
"""
  
```

**Figure 5.** Modified prompt of *Gazebo\_agent*

## 1.2 Second approach: LiveKit

In this second approach, we have made use of [LiveKit](#), an open-source platform for user-AI interactions. Contrary to the first approach, this allows for speech to speech interactions with humans. More precisely, we make use of a pipeline agent, which consists of a STT (Speech To Text), an LLM and a TTS(Text To Speech). Thus, the audio query of the user will be converted into text, then plugged into an LLM which has tools, and finally the LLM's answer is converted back into speech.

Originally, we were hoping to replace that LiveKit LLM with our LangChain LLM, but it unfortunately proved much more complicated than we thought, and so we haven't been able to use LangChain within LiveKit.

### 1.2.1 Usage of tools with LiveKit

```
@llm.ai_callable()
async def get_time(
    self,
):
    """Gets the number of seconds that passed since epoch (the first of january 1970)"""
    logger.info(f"Fetching time with get_time function")
    seconds = time.time()
    return f"{str(seconds)}s have passed since 1st jan 1970"
```

**Figure 6.** Example tool of LiveKit

Fig.6 shows the same tool as Fig.1 except for the fact that it is now in a 'LiveKit format'. The function's docstring is still used as the description that the LLM uses.

"logger" can be ignored for now, it serves as a print statement in the terminal.

### 1.2.2 Tools used

The tools used for the LiveKit LLM are very similar to the tools used for the LangChain LLM. We simply copied the tools we developed and adapted them to the correct "LiveKit format" for the LiveKit LLM to be able to use them.

```
@llm.ai_callable()
async def give_robot_velocity(
    self,
    x_linear: Annotated[
        int, llm.TypeInfo(
            | description="the linear velocity along the x axis that the robot will get."
        )] = 0.0,
    y_linear: Annotated[
        int, llm.TypeInfo(
            | description="the linear velocity along the y axis that the robot will get."
        )] = 0.0,
    z_linear: Annotated[
        int, llm.TypeInfo(
            | description="the linear velocity along the z axis that the robot will get."
        )] = 0.0,
    x_angular: Annotated[
        int, llm.TypeInfo(
            | description="the angular velocity along the x axis that the robot will get."
        )] = 0.0,
    y_angular: Annotated[
        int, llm.TypeInfo(
            | description="the angular velocity along the y axis that the robot will get."
        )] = 0.0,
    z_angular: Annotated[
        int, llm.TypeInfo(
            | description="the angular velocity along the z axis that the robot will get."
        )] = 0.0,
    rate: Annotated[
        int, llm.TypeInfo(
            | description="the rate, in Hz, at which this information will be passed to the robot."
        )] = 10,
) -> dict:
    """Make the robot move with the given x,y,z linear velocities and x,y,z angular velocities"""

    logger.info(f"called give_robot_velocity function")

    cmd = f'ros2 topic pub' + ((f' -r {rate}') if rate != 1 else '') + \
          f' /cmd_vel geometry_msgs/msg/Twist [{"linear: {x: {x_linear}, y: \
              {y_linear}, z: {z_linear}}}, angular: {x: {x_angular}, y: \
              {y_angular}, z: {z_angular}}}]' --once'

    success, output = ros_tools_mini.execute_ros_command(cmd)

    return 'success'
```

**Figure 7.** Tool for giving the robot any linear or angular velocity, for LiveKit

```

@llm.ai_callable()
async def ros2_topic_echo_info(
    self,
    topic: Annotated[
        str, llm.TypeInfo(description="The name of the ROS topic to echo.")
    ],
    timeout: Annotated[
        float, llm.TypeInfo(
            description="Max time to wait for a message before timing out."
        )
    ] = 3.0
) -> dict:
    """
    Echoes the contents of a specific ROS2 topic, and always returns the message.

    :note: Do not use this tool if the number of messages is large.
    | This will cause the response to be too large and may cause the tool to fail.
    """

    logger.info(f"called ros2_topic_echo_info function")

    cmd = f"ros2 topic echo {topic} --once --spin-time {timeout}"

    success, output = ros_tools_mini.execute_ros_command(cmd)

    return output

```

**Figure 8.** Tool for listening to any active ros2 topic, for LiveKit

It is important to note that we have not copied all of the LangChain tools over to LiveKit, as many of those tools are not vital to the robot functioning. This is especially true for the *ros\_tools.py* tools from ROSA, as they were developed for user queries regarding the state of the current ros2 framework, whereas our aim was for a simpler LiveKit agent.

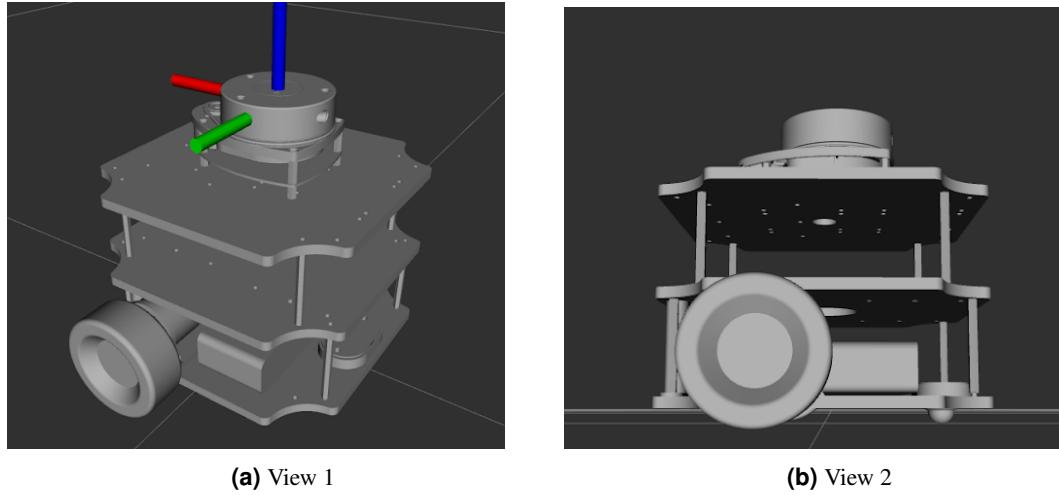
## 1.3 Simulation

### 1.3.1 Modeling the Robots

In this project we simulated 2 robots: namely A24 and OpenBase2. Both robots are custom made and not available for public use. This meant an additional step is needed before we can use these robots within our simulation and that is creating the URDF/xacro file.

The URDF file stands for unified robotics description format. It is a xml file used to describe a robot by its links joints and transforms between them. There are additional tags like inertia and collision which are required by simulation software to accurately model the robot in real life. The problem is that for any physical robot, the design of the links can be very difficult to model and code by hand. Especially measurements like the Inertia tensor and the center of mass for each link must be done computationally. The solution to this is to use some software to go from your accurate CAD assembly model of the robot to the URDF. Both robots were modeled in Fusion 360 and a plugin called **fusion2urdf** was used to go from my file to urdf.

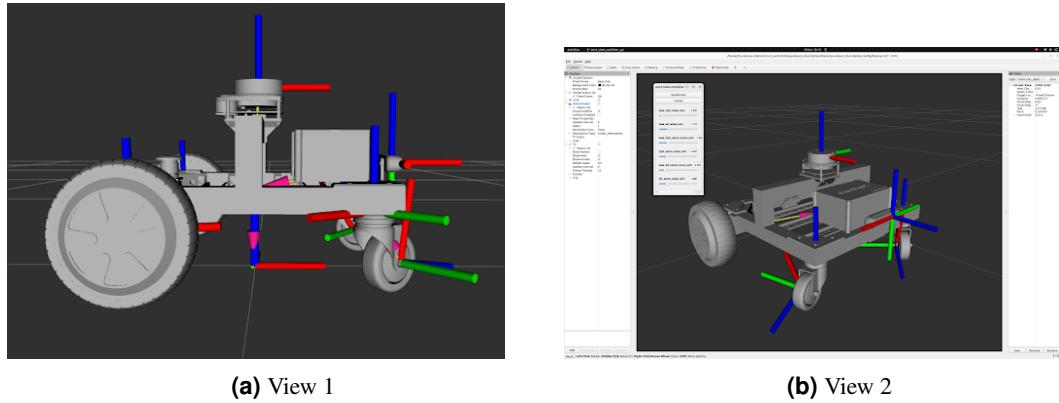
After getting the urdf we can view it in RViz by running the built in **robot state publisher** node in ROS2 and optionally the **joint state publisher** to update the transforms if there are any desired changes to the joint angles.



(a) View 1

(b) View 2

**Figure 9.** Robot A24



(a) View 1

(b) View 2

**Figure 10.** Robot OpenBase2

The generated URDF file also includes data for the inertia values that can be used to accurately model the robots properties.

```
<link name="base_footprint"/>
<link name="base_link">
  <inertial>
    <origin xyz="3.882070344410716e-14 -0.0018708886856651943 4.98641867654471e-18" rpy="0 0 0"/>
    <mass value="1.3439256828712636"/>
    <inertia ixx="0.003108" iyy="0.003286" izz="0.006387" ixz="-0.0" iyx="0.0" izy="0.0" ixy="0.0"/>
  </inertial>
  <visual>
    <origin xyz="0 0 0" rpy="0 0 -1.5708"/>
    <geometry>
      <mesh filename="file://${find a24}/meshes/base_link_shell_unrotated.stl" scale="0.001 0.001 0.001"/>
    </geometry>
    <material name="silver"/>
  </visual>
  <collision>
    <origin xyz="0 0 0" rpy="0 0 -1.5708"/>
    <geometry>
      <mesh filename="file://${find a24}/meshes/base_link_shell_unrotated.stl" scale="0.001 0.001 0.001"/>
    </geometry>
  </collision>
</link>
```

**Figure 11.** URDF file

Meshes are used in places of simple shapes to represent complex geometries

### 1.3.2 Simulating the Robot

To bring the robot into the Gazebo simulation with the ROS2 bindings we use the gazebo ros bridge. Gazebo is a standalone piece of software like other simulation engines however bindings to ROS2 can be installed to provide easy control of the robot.

We can also further modify our URDF file by replacing it with xacro, enabling things like macros and better control over our robots description and add gazebo tags to the model. These tags allow us to change the friction of each link to better match the real world and simulate sensors like cameras, depth cameras and 2d lidars. All of which will be used within the robots

```
<gazebo reference="caster_wheel_1">
  <material>${body_color}</material>
  <mu1>0.001</mu1>
  <mu2>0.001</mu2>
  <self_collide>true</self_collide>
</gazebo>
```

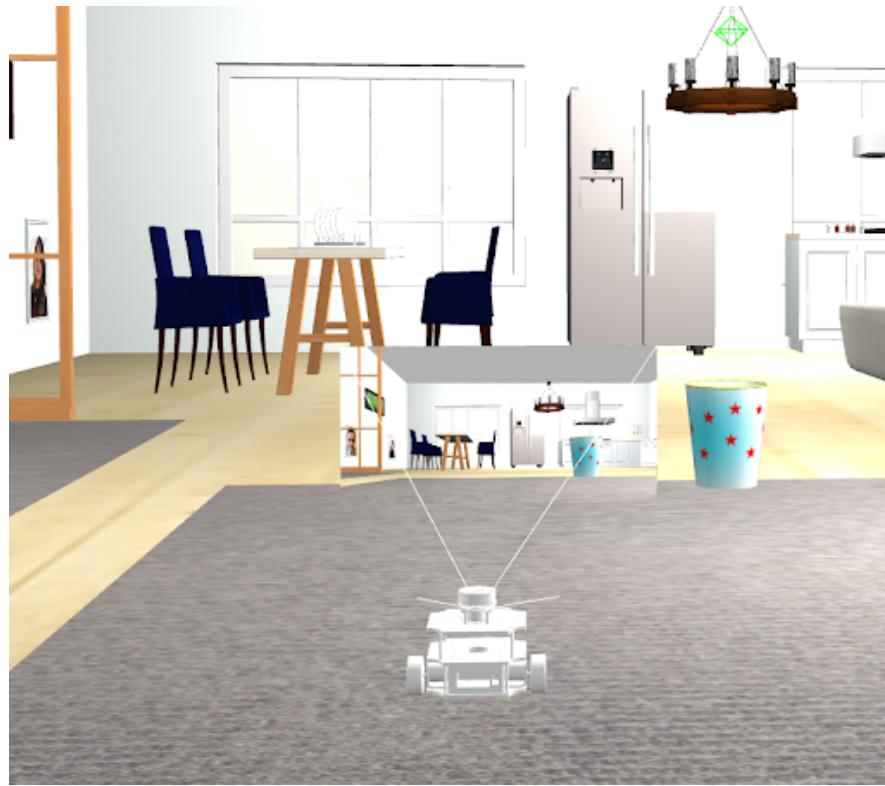
**Figure 12.** Links like the castor wheel have a low friction to slide easily along the surface

```
<gazebo reference="right_wheel_1">
  <material>${body_color}</material>
  <mu1>1</mu1>
  <mu2>1</mu2>
  <self_collide>true</self_collide>
</gazebo>

<gazebo reference="left_wheel_1">
  <material>${body_color}</material>
  <mu1>1</mu1>
  <mu2>1</mu2>
  <self_collide>true</self_collide>
</gazebo>
```

**Figure 13.** The wheels are modeled with a frictional coefficient of 1 along both axis to reduce wheel slippage leading to a more accurate odometry which will be discussed later

A world file of a home is also loaded in. This file was developed by the team at **AWS robomaker** and was chosen due to its rich variety of objects and resemblance to an environment these robots may one day be in. The feature complexity within the world also makes it better for SLAM and the open spaces between rooms give us a mixture of spaces to test the robots performance



**Figure 14.** A24 in a house environment



**Figure 15.** House environment

### 1.3.3 The relevance of ROS2 control

ROS2 control is a software package with a variety of useful applications. Although there was a steep learning curve to fully learn everything the stack had to offer, it is incredibly beneficial for scaling the robots and shortening development time in the long run. The biggest benefit for me with ROS2 control is its innate **Sim2Real** capability. Once a robot has been fully developed in ROS2 with ros2\_control, simply by swapping the hardware interface to one that can communicate with the necessary hardware on the robot that the physical robot can come to life

Why this is relevant here is that ros2 control comes with a gazebo hardware interface so that joint movements in Gazebo can also update the joint states of the robot and any command given to the robot feeds forward into the simulation.

Here we add to the xacro file for the robots with Fig.16:

```

<ros2_control name="GazeboSystem" type="system">
  <hardware>
    <plugin>gazebo_ros2_control/GazeboSystem</plugin>
  </hardware>
  <joint name="left_wheel_joint">
    <command_interface name="velocity">
      <param name="min">-10</param>
      <param name="max">10</param>
    </command_interface>
    <state_interface name="velocity"/>
    <state_interface name="position"/>
  </joint>
  <joint name="right_wheel_joint">
    <command_interface name="velocity">
      <param name="min">-10</param>
      <param name="max">10</param>
    </command_interface>
    <state_interface name="velocity"/>
    <state_interface name="position"/>
  </joint>
</ros2_control>

```

Figure 16

It is saying that there are 2 joints that we need to control. One for the right wheel and for the left. Each joint has 1 command interface being velocity and 2 state interfaces being position and velocity. Limits have also been set.

Our 2 wheeled robot has kinematics which translate some desired linear and rotational velocity into angular wheel velocities. While the math is not complex, ros2\_control provides a controller called the **diff drive controller** which given the parameters for our robot can compute the necessary angular velocity the wheels must be at.

```

controller_manager:
  ros_parameters:
    update_rate: 50
    use_sim_time: true

  diff_cont:
    type: diff_drive_controller/DiffDriveController

  joint_broad:
    type: joint_state_broadcaster/JointStateBroadcaster

  diff_cont:
    ros_parameters:

      publish_rate: 50.0

      base_frame_id: base_footprint

      left_wheel_names: ['left_wheel_joint']
      right_wheel_names: ['right_wheel_joint']
      wheel_separation: 0.226
      wheel_radius: 0.035

      use_stamped_vel: false
      enable_odom_tf: true

```

**Figure 17**

There is also another controller called **joint state broadcaster** and that is responsible for updating the transforms for the robot given changes in the joint states. All the controllers are running at 50hz which is reasonable enough to provide accuracy and not overload the system given all the other nodes running.

## 2 RESULTS

### 2.1 LangChain and LiveKit Comparison

#### 2.1.1 Results for the LangChain LLM

The LangChain LLM is successful at understanding user queries, correctly using the right tools, and answering the user.

```

(venv) andy@Andyscomputer:~/project_oop$ /home/andy/project_oop/venv/bin/python /home/andy/project_oop/test_demands.py
*Agent_OOP* created new agent
Ask agent (or Ctrl+C to quit): go forward

> Entering new None chain...

Invoking: 'give_robot_velocity' with '{'x_linear': 1.0}'

-> ros2 topic pub /cmd_vel \geometry_msgs/msg/Twist "[linear: {x: 1.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.0}]" --once
{'output': 'publisher: beginning Loop\\nPublishing #1: geometry_msgs.msg.Twist[linear=geometry_msgs.msg.Vector3(x=1.0, y=0.0, z=0.0), angular=geo
metry_msgs.msg.Vector3(x=0.0, y=0.0, z=0.0)]\\n\\n'The robot is now moving forward with a Linear velocity of 1.0.

> Finished chain.

```

**Figure 18.** LLM successfully answering "go forward" query

For a full breakdown of Fig.18 above: we can first see the user query on line 3 ("go forward"). Then the LLM, in response to the query, calls a tool (green text : *Invoking: 'give\_robot\_velocity' with '{'xlinear': 1.0}'*) which is the tool responsible for giving the robot a linear or angular velocity. '{'xlinear': 1.0}' corresponds to the parameters passed to the tool. Here, it means the robot will start moving with a velocity of 1.0 m/s along the x axis. Right after that, the gray text which starts by an arrow (*ros2 topic pub ...*) corresponds to the command that the tool executed in terminal. Finally, the blue text corresponds to the

output of the terminal command and the final green text corresponds to the LLM's answer to the query: *The robot is now moving forward with a linear velocity of 1.0.* It confirms that the robot has indeed completed the query successfully.

```
Ask agent (or Ctrl+C to quit): stop

> Entering new None chain...

Invoking: `stop_robot` with `{}'

-> ros2 topic pub /cmd_vel geometry_msgs/msg/Twist "{linear: {x: 0, y: 0, z: 0}, angular: {x: 0, y: 0, z: 0}}" --once
{'output': 'publisher: beginning loop\npublishing #1: geometry_msgs.msg.Twist{linear:geometry_msgs.msg.Vector3(x=0.0, y=0.0, z=0.0), angular=geometry_msgs.msg.Vector3(x=0.0, y=0.0, z=0.0)}\n\nThe robot has been stopped successfully. All Linear and angular velocities are set to zero.'}

> Finished chain.
```

**Figure 19.** LLM successfully answering "stop" query

Similar to Fig.18, Fig.19 shows the user query (*stop*) followed by the tool used (*stop\_robot*, with no parameters). We can also see the terminal command in gray as well as the LLM output in green (the second green line), which confirms the successful stop of the robot, as per requested by the user.

We can see that Fig.19 shows the LLM called a "stop\_robot" tool, when the terminal command looks identical to the *give\_robot\_velocity* tool with all 0 velocities. Indeed the *stop\_robot* tool is identical to the *give\_robot\_velocity* tool for all 0 velocities, but testing has proven that the LLM will stop the robot much more reliably if it has a tool specifically for it. We thus decided to add a theoretically useless tool to not only improve performance but also safety (real robots need to be able to stop any current action quickly and reliably at any moment if they start behaving unexpectedly).

**Figure 20.** LLM returning coordinates to "where are you" query

In Fig.20, the LLM is asked the position of the robot ("where are you"). To answer this query, the LLM uses the `ros2_topic_echo_info` tool, which echoes(reads) a given ros2 topic, and then feeds the result to the LLM. Here, the topic echoed is `/odom`, which corresponds to the robot's odometry (the position of the robot calculated from encoders on the wheels, it isn't its true position).

The pink text corresponds to the information returned by the ros2 command to the LLM: it is one echo from the `/odom` topic. Finally, the LLM answers the query with the green text by clearly displaying the X, Y and Z coordinates of the robot.

```

Ask agent (or Ctrl+C to quit): go forward

> Entering new None chain...

Invoking: 'give_robot_velocity' with '{'x_linear': 1.0}'

-> ros2 topic pub /cmd_vel geometry_msgs/msg/Twist "[linear: {x: 1.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.0}]" --once
{'output': 'publisher: beginning Loop\npublishing #1: geometry_msgs.msg.Twist(linear=geometry_msgs.msg.Vector3(x=1.0, y=0.0, z=0.0), angular=geo
metry_msgs.msg.Vector3(x=0.0, y=0.0, z=0.0))\n\nThe robot is now moving forward with a Linear velocity of 1.0.'}

> Finished chain.

Ask agent (or Ctrl+C to quit): stop

> Entering new None chain...

Invoking: 'stop_robot' with '{}'

-> ros2 topic pub /cmd_vel geometry_msgs/msg/Twist "[linear: {x: 0.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.0}]" --once
{'output': 'publisher: beginning Loop\npublishing #1: geometry_msgs.msg.Twist(linear=geometry_msgs.msg.Vector3(x=0.0, y=0.0, z=0.0), angular=geo
metry_msgs.msg.Vector3(x=0.0, y=0.0, z=0.0))\n\nThe robot has been stopped successfully. All Linear and angular velocities are set to zero.'}

> Finished chain.

Ask agent (or Ctrl+C to quit): where are you

> Entering new None chain...

Invoking: 'ros2_topic_echo_info' with {'topic': '/odom'}
```

-> ros2 topic echo /odom --once --spin-time 1.0

{'echoes': 'header:\n stamp:\n sec: 159\n nanosec: 16000000\n frame\_id: odom\n child\_frame\_id: base\_footprint\n pose:\n position:\n x: 9.596427585589527\n y: 0.2706368067834804\n z: -0.00036529347712066597\n orientation:\n x: 0.00232253832516135\n y: -0.00707166895204734\n z: 0.03180604687546085\n w: 0.9994690152641238\n covariance:\n linear:\n x: -1.0e-05\n y: -0.01\n z: -0.01\n angular:\n x: -0.01\n y: -0.01\n z: -1.0e-05\n twist:\n linear:\n x: 0.001861288986222776\n y: -4.3849007253415e-06\n z: 0.01\n angular:\n x: 0.01\n y: 0.01\n z: 3.968428195366561e-05\n covariance:\n linear:\n x: -1.0e-05\n y: -0.01\n z: -0.01\n angular:\n x: -0.01\n y: -0.01\n z: -1.0e-05\n twist:\n linear:\n x: 0.0000000000000000\n y: -1000000000000000.0\n z: 0.0\n angular:\n x: -0.01\n y: -0.01\n z: -0.01\n covariance:\n linear:\n x: 0.0000000000000000\n y: -1000000000000000.0\n z: 0.0\n angular:\n x: -0.01\n y: -0.01\n z: -0.01\n I am located at the following coordinates in the simulation:\n\n- \*\*\*X\*\*\* 9.5964\n- \*\*\*Y\*\*\* 0.2706\n- \*\*\*Z\*\*\* -0.0004\nThis position is relative to the frame of reference Labeled "odom."'

> Finished chain.

**Figure 21.** Full conversation for the examples given

In Fig.21 above, we can see the entire conversation user/LLM for the set of examples we have showcased in figures 18, 19 and 20.

### 2.1.2 Results for the LiveKit LLM

Similarly to the LangChain LLM, the LiveKit LLM is successful at understanding user queries, correctly using the right tools, and answering the user.

However the big difference between our LangChain and LiveKit approach is that LiveKit is speech to speech: a python file is executed in terminal, and the user can enter a [virtual room](#) and speak with the LLM (if the python file isn't running, the virtual room will not work). Another benefit of LiveKit is that the virtual room can be run on a separate device, like a smartphone.

```

(venv) andys@Andyscomputer:~/project_oop/secure-syscall-2e6768$ python3 hagent.py dev
2024-12-10 10:07:13,371 - DEBUG asyncio - Using selector: EpollSelector
2024-12-10 10:07:13,373 - DEV livekit.agents - Watching /home/andys/project_oop/secure-syscall-2e6768
2024-12-10 10:07:14,820 - DEBUG asyncio - Using selector: EpollSelector
2024-12-10 10:07:14,827 - INFO livekit.agents - starting worker {"version": "0.11.3", "rtt-version": "0.18.1"}
2024-12-10 10:07:14,887 - INFO livekit.agents - livekit.agents - registered worker {"id": "AW_caQxfsmX3c", "region": "UK", "protocol": "1.5", "node_id": "NC_OLONDON1B_Jun76dtX8Gup1"}
2024-12-10 10:07:22,653 - INFO Livekit.agents - received job request {"job_id": "AJ_CJMxiKwfVact", "dispatch_id": "", "room_name": "sbx-2e6768-bl9QTimFmPgBrfNEDzLsn", "agent_name": "", "resuing": false}
2024-12-10 10:07:24,266 - DEBUG asyncio - Using selector: EpollSelector {"pid": 2176}
2024-12-10 10:07:24,268 - INFO livekit.agents - initializing process {"pid": 2176}
2024-12-10 10:07:24,386 - INFO livekit.agents - process initialized {"pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:24,390 - INFO voice-agent - connecting to room sbx-2e6768-bl9QTimFmPgBrfNEDzLsn {"pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:24,656 - INFO voice-agent - starting voice assistant for participant immutable-interface {"pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:26,554 - DEBUG livekit.agents.pipeline - speech playout started {"speech_id": "5c78f0cfdf6", "pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:28,362 - DEBUG livekit.agents.pipeline - speech playout finished {"speech_id": "5c78f0cfdf6", "interrupted": false, "pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:28,362 - DEBUG livekit.agents.pipeline - committed agent speech {"agent_transcript": "Hey, how can I help you today?", "interrupted": false, "speech_id": "5c78f0cfdf6", "pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:31,927 - DEBUG livekit.agents.pipeline - received user transcript {"user_transcript": "Make the robot move forward", "pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:33,094 - DEBUG livekit.agents.pipeline - validated agent reply {"speech_id": "8bd8db7c363d", "pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:33,094 - DEBUG livekit.agents.pipeline - speech playout finished {"speech_id": "8bd8db7c363d", "interrupted": false, "pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:33,095 - DEBUG livekit.agents.pipeline - executing ai function {"function": "give_robot_velocity", "speech_id": "8bd8db7c363d", "pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:33,095 - INFO voice-agent - called give_robot_velocity function {"pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:36,531 - DEBUG livekit.agents.pipeline - speech playout started {"speech_id": "8bd8db7c363d", "pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:40,976 - DEBUG livekit.agents.pipeline - speech playout finished {"speech_id": "8bd8db7c363d", "interrupted": false, "pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:40,976 - DEBUG livekit.agents.pipeline - committed agent speech {"agent_transcript": "The robot is now moving forward. Is there anything else I can help you with?", "interrupted": false, "speech_id": "8bd8db7c363d", "pid": 2176, "job_id": "AJ_CJMxiKwfVact"}

```

**Figure 22.** LLM successfully answering "make the robot move forward" query

For a full breakdown of Fig.22: the first line, `python3 hagent.py dev`, runs the python file responsible for the LiveKit code. Then, the first 10 or so lines correspond to the virtual room and worker(the AI) starting up. From that point onward, the speech to speech user-LLM conversation starts with the message *Hey, how can I help you today?*, which is the line right above the line underlined in red.

Underlined in red is the user query: *Make the robot move forward*. In response to the query, the LLM calls the `give_robot_velocity` tool, underlined in blue. We can see two different calls as the first one is the LiveKit agent reporting that it is using a tool (the first one) and the second one is due to a print statement that we added (the "logger" mentioned in Fig6 and visible in Figs.7 and 8), to make it easier to see when the agent uses a tool. Finally, underlined in green, is the LLM's answer to the user (which is transformed into speech for the user in the virtual room).

```

2024-12-10 10:07:40,976 - DEBUG Livekit.agents.pipeline - committed agent speech {"agent_transcript": "The robot is now moving forward. Is there anything else I can help you with?", "interrupted": false, "speech_id": "8bd8db7c363d", "pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:42,812 - DEBUG Livekit.agents.pipeline - received user transcript {"user_transcript": "Stop the robot", "pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:43,189 - DEBUG livekit.agents.pipeline - validated agent reply {"speech_id": "ela207919e28", "pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:43,866 - DEBUG livekit.agents.pipeline - speech playout finished {"speech_id": "ela207919e28", "interrupted": false, "pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:43,867 - DEBUG livekit.agents.pipeline - executing ai function {"function": "stop_robot", "speech_id": "ela207919e28", "pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:43,888 - INFO voice-agent - called stop_robot function {"pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:46,963 - DEBUG livekit.agents.pipeline - speech playout started {"speech_id": "ela207919e28", "pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:49,128 - DEBUG livekit.agents.pipeline - speech playout finished {"speech_id": "ela207919e28", "interrupted": false, "pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:49,129 - DEBUG livekit.agents.pipeline - committed agent speech {"agent_transcript": "The robot has stopped. What's next?", "interrupted": false, "speech_id": "ela207919e28", "pid": 2176, "job_id": "AJ_CJMxiKwfVact"}

```

**Figure 23.** LLM successfully answering "stop the robot" query

Fig.23 shows the result of a *stop the robot* query (red), to which the LLM calls the `stop_robot` tool (blue). The robot successfully stops, and the LLM answers the user (green).

This is identical to the same test performed with LangChain (Fig.19).

```

2024-12-10 10:07:49 129 - DEBUG livekit.agents.pipeline - committed agent speech {"agent_transcript": "The robot has stopped. What's next?", "interrupted": false, "speech_id": "e1a207919e28", "pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:52, 804 - DEBUG livekit.agents.pipeline - received user transcript {"user_transcript": "What are the coordinates of the robot?"}
2024-12-10 10:07:53, 180 - DEBUG livekit.agents.pipeline - validated agent reply {"speech_id": "e65374139ae0", "pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:53, 619 - DEBUG livekit.agents.pipeline - speech playout finished {"speech_id": "e65374139ae0", "interrupted": false, "pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:53, 619 - DEBUG livekit.agents.pipeline - executing ai function {"function": "ros2_topic_echo_info", "speech_id": "e65374139ae0", "pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:53, 619 - INFO voice-agent - called ros2_topic_echo_info function {"pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:55, 750 - WARNING livekit.agents - Running <Task finished name='Task-108' core=<AssistantFn.ros2_topic_echo_info() done at /home/andys/project_oop> took too long: 2.13 seconds {"pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:57, 803 - DEBUG livekit.agents.pipeline - speech playout started {"speech_id": "e65374139ae0", "pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:08:03, 448 - DEBUG livekit.agents.pipeline - speech playout finished {"speech_id": "e65374139ae0", "interrupted": false, "pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:08:03, 448 - DEBUG livekit.agents.pipeline - committed agent speech {"agent_transcript": "The robot's coordinates are x: 11.27, y: 0.31, z: 0.00. What's next?", "interrupted": false, "speech_id": "e65374139ae0", "pid": 2176, "job_id": "AJ_CJMxiKwfVact"}

```

**Figure 24.** LLM successfully answering "What are the coordinates of the robot?" query

Fig.24 shows the result of a *What are the coordinates of the robot?* query (red), to which the LLM calls the *ros2\_topic\_echo\_info* tool (blue). The output of that command isn't written down in the terminal we're looking at, unlike in the LangChain example (Fig.20) but the LLM still has access to it as we can see in its answer (green) with the correct coordinates.

```

2024-12-10 10:07:13,373 - DEBUG asynco - Using selector: EpollSelector
2024-12-10 10:07:13,373 - DEV livekit.agents - Watching at /home/andys/project_oop/secure-syscall-2e6768 python hagent.py dev
2024-12-10 10:07:14,420 - DEBUG asynco - Using selector: EpollSelector
2024-12-10 10:07:14,420 - DEBUG asynco - rttc version: "0.10.1"
2024-12-10 10:07:14,487 - INFO livekit.agents - registered worker {"id": "#M_caJQfrsWxzc", "region": "UK", "protocol": 15, "node_id": "NC_OLONDON1B.Jun76dtXgUpP"}
2024-12-10 10:07:17,22,653 - INFO livekit.agents - received job request {"job_id": "AJ_CJMxiKwfVact", "dispatch_id": "", "room_name": "sbx-2e6768-bl9TmFaBgfRfNdz8Lsn", "agent_name": "AJ_CJMxiKwfVact", "version": "0.11.3", "pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:24,266 - DEBUG asynco - Using selector: EpollSelector {"pid": 2176}
2024-12-10 10:07:24,268 - INFO livekit.agents - initializing process {"pid": 2176}
2024-12-10 10:07:24,309 - DEBUG asynco - Using selector: EpollSelector {"pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:24,390 - INFO voice-agent - connecting to room sbx-2e6768-bl9TmFaBgfRfNdz8Lsn {"pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:24,456 - INFO voice-agent - starting voice assistant for participant immutable-interface {"pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:24,456 - DEBUG livekit.agents.pipeline - speech playout finished {"speech_id": "e65374139ae0", "pid": 2176, "job_id": "AJ_CJMxiKwfVact", "interrupted": false, "speech_id": "e65374139ae0", "pid": 2176, "job_id": "AJ_CJMxiKwfVact", "version": "0.10.1", "rtt": 0.001}
2024-12-10 10:07:27,33,953 - DEBUG livekit.agents.pipeline - received user transcript {"user_transcript": "Make the robot move forward.", "pid": 2176, "job_id": "AJ_CJMxiKwfVact", "version": "0.10.1", "rtt": 0.001}
2024-12-10 10:07:28,362 - DEBUG livekit.agents.pipeline - validated agent reply {"speech_id": "#bdd8d07c363d", "pid": 2176, "job_id": "AJ_CJMxiKwfVact", "version": "0.10.1", "rtt": 0.001}
2024-12-10 10:07:31,927 - DEBUG livekit.agents.pipeline - speech playout finished {"speech_id": "#bdd8d07c363d", "pid": 2176, "job_id": "AJ_CJMxiKwfVact", "interrupted": false, "speech_id": "#bdd8d07c363d", "pid": 2176, "job_id": "AJ_CJMxiKwfVact", "version": "0.10.1", "rtt": 0.001}
2024-12-10 10:07:33,034 - DEBUG livekit.agents.pipeline - speech playout finished {"speech_id": "#bdd8d07c363d", "pid": 2176, "job_id": "AJ_CJMxiKwfVact", "interrupted": false, "speech_id": "#bdd8d07c363d", "pid": 2176, "job_id": "AJ_CJMxiKwfVact", "version": "0.10.1", "rtt": 0.001}
2024-12-10 10:07:33,034 - DEBUG livekit.agents.pipeline - executing ai function {"function": "give_robot_velocity", "speech_id": "#bdd8d07c363d", "pid": 2176, "job_id": "AJ_CJMxiKwfVact", "version": "0.10.1", "rtt": 0.001}
2024-12-10 10:07:33,035 - INFO voice-agent - called give_robot_velocity function {"pid": 2176, "job_id": "AJ_CJMxiKwfVact", "version": "0.10.1", "rtt": 0.001}
2024-12-10 10:07:33,035 - DEBUG livekit.agents.pipeline - received user transcript {"user_transcript": "Move the robot move forward.", "pid": 2176, "job_id": "AJ_CJMxiKwfVact", "version": "0.10.1", "rtt": 0.001}
2024-12-10 10:07:48,976 - DEBUG livekit.agents.pipeline - speech playout finished {"speech_id": "#bdd8d07c363d", "pid": 2176, "job_id": "AJ_CJMxiKwfVact", "interrupted": false, "speech_id": "#bdd8d07c363d", "pid": 2176, "job_id": "AJ_CJMxiKwfVact", "version": "0.10.1", "rtt": 0.001}
2024-12-10 10:07:48,976 - DEBUG livekit.agents.pipeline - committed agent speech {"agent_transcript": "The robot is now moving forward. Is there anything else I can help you with?", "pid": 2176, "job_id": "AJ_CJMxiKwfVact", "version": "0.10.1", "rtt": 0.001}
2024-12-10 10:07:48,976 - DEBUG livekit.agents.pipeline - received user transcript {"user_transcript": "Stop the robot.", "pid": 2176, "job_id": "AJ_CJMxiKwfVact", "version": "0.10.1", "rtt": 0.001}
2024-12-10 10:07:48,976 - DEBUG livekit.agents.pipeline - validated agent reply {"speech_id": "#a1e07919e28", "pid": 2176, "job_id": "AJ_CJMxiKwfVact", "version": "0.10.1", "rtt": 0.001}
2024-12-10 10:07:48,976 - DEBUG livekit.agents.pipeline - speech playout finished {"speech_id": "#a1e07919e28", "pid": 2176, "job_id": "AJ_CJMxiKwfVact", "interrupted": false, "speech_id": "#a1e07919e28", "pid": 2176, "job_id": "AJ_CJMxiKwfVact", "version": "0.10.1", "rtt": 0.001}
2024-12-10 10:07:48,976 - DEBUG livekit.agents.pipeline - executing ai function {"function": "stop_stop_robot", "speech_id": "#a1e07919e28", "pid": 2176, "job_id": "AJ_CJMxiKwfVact", "version": "0.10.1", "rtt": 0.001}
2024-12-10 10:07:48,976 - INFO voice-agent - called stop_stop_robot function {"pid": 2176, "job_id": "AJ_CJMxiKwfVact", "version": "0.10.1", "rtt": 0.001}
2024-12-10 10:07:48,976 - DEBUG livekit.agents.pipeline - speech playout started {"speech_id": "#a1e07919e28", "pid": 2176, "job_id": "AJ_CJMxiKwfVact", "interrupted": false, "speech_id": "#a1e07919e28", "pid": 2176, "job_id": "AJ_CJMxiKwfVact", "version": "0.10.1", "rtt": 0.001}
2024-12-10 10:07:48,976 - DEBUG livekit.agents.pipeline - speech playout finished {"speech_id": "#a1e07919e28", "pid": 2176, "job_id": "AJ_CJMxiKwfVact", "interrupted": false, "speech_id": "#a1e07919e28", "pid": 2176, "job_id": "AJ_CJMxiKwfVact", "version": "0.10.1", "rtt": 0.001}
2024-12-10 10:07:49,129 - DEBUG livekit.agents.pipeline - committed agent speech {"agent_transcript": "The robot has stopped. What's next?", "pid": 2176, "job_id": "AJ_CJMxiKwfVact", "version": "0.10.1", "rtt": 0.001}
2024-12-10 10:07:53,904 - DEBUG livekit.agents.pipeline - received user transcript {"user_transcript": "What are the coordinates of the robot?", "pid": 2176, "job_id": "AJ_CJMxiKwfVact", "version": "0.10.1", "rtt": 0.001}
2024-12-10 10:07:53,904 - DEBUG livekit.agents.pipeline - validating agent reply {"speech_id": "#e65374139ae0", "pid": 2176, "job_id": "AJ_CJMxiKwfVact", "version": "0.10.1", "rtt": 0.001}
2024-12-10 10:07:55,750 - WARNING livekit.agents - Running <Task finished name='Task-108' core=<AssistantFn.ros2_topic_echo_info() done, defined at /home/andys/project_oop/secure-syscall-2e6768/hagent.py:158> took too long: 2.13 seconds {"pid": 2176, "job_id": "AJ_CJMxiKwfVact"}
2024-12-10 10:07:55,750 - DEBUG livekit.agents - shutting down job task {"pid": 2176, "job_id": "AJ_CJMxiKwfVact", "version": "0.10.1", "rtt": 0.001}
2024-12-10 10:08:14,586 - INFO livekit.agents - shutting down job task {"pid": 2176, "job_id": "AJ_CJMxiKwfVact", "version": "0.10.1", "rtt": 0.001}
2024-12-10 10:08:14,586 - DEBUG livekit.agents - shutting down job task {"pid": 2176, "job_id": "AJ_CJMxiKwfVact", "version": "0.10.1", "rtt": 0.001}
2024-12-10 10:08:14,586 - DEBUG livekit.agents - http_session() closing the httpclient ctx {"pid": 2176, "job_id": "AJ_CJMxiKwfVact", "version": "0.10.1", "rtt": 0.001}

```

**Figure 25.** Full conversation of the above examples

Fig.25 corresponds to the full user-agent discussion used for the examples discussed in figures 22, 23 and 24.

### 2.1.3 Tool chaining

Both the LiveKit approach and the LangChain approach we have used work well. In both cases the LLM understands the user's query and calls the correct tools for the task at hand. The biggest difference between both approaches is that LiveKit allows speech-to-speech interactions whereas LangChain is only text-to-text.

However, extensive testing of both approaches has proven that there is a big difference in how both agents can use tools: the LangChain agent has shown to be much better at completing queries that required multiple tools.

Consider the following example:

Each agent has access to 2 tools: *get\_time()*, which returns the total number of seconds which have passed since epoch (01/01/1970); and *timestamp\_convert()*, which converts that number of seconds into a human readable format, such as "11-02-2024 13:43:26".

If those agents were then queried "What time is it?", they would first have to call *get\_time()*, then pass

the output of that tool as input to `timestamp_convert()`, and finally read the output of that tool to be able to answer the query.

For simplicity of notation, let's call the action of using multiple tools which each need the prior tool's outputs as inputs "Tool chaining". Thus, here we have a chain of 2 tools.

Our LangChain agent has proven to be capable of tool chaining, contrary to our LiveKit agent, severely limiting how much it is capable of doing.

```
from langchain.agents import tool
import time
from datetime import datetime

@tool
def get_time() -> int:
    """Gets the number of seconds that passed since the first of january 1970"""
    seconds = time.time()
    return seconds

@tool
def timestamp_convert(timestamp : int) -> str:
    """Converts passed timestamp into year,month,day,hour,minute,second """
    dt_object = datetime.fromtimestamp(timestamp)

    return str(dt_object)
```

**Figure 26.** The tools the LangChain agent has access to

```
*Agent_OOP* created new agent
Ask agent (or Ctrl+C to quit): what time is it?

> Entering new None chain...

Invoking: `get_time` with `{}`

1733842820.1534846
Invoking: `timestamp_convert` with `{'timestamp': 1733842820}`

2024-12-10 15:00:20The current time is 15:00:20 on December 10, 2024.

> Finished chain.
Ask agent (or Ctrl+C to quit): █
```

**Figure 27.** LangChain agent successful tool chain

Fig.27 shows the agent chaining tools, using the tools from Fig.26. It first calls `get_time` (first green text), which outputs a timestamp (blue). Then, the agent uses this timestamp output as an input for another tool, `timestamp_convert`. This returns the date and time (in yellow), which the LLM reads to answer the user (last green text).

```

@llm.ai_callable()
async def get_time(
    self,
):
    """Gets the number of seconds that passed since epoch (the first of january 1970)"""
    logger.info(f"Fetching time with get_time function")
    seconds = time.time()
    return f"{str(seconds)}s have passed since 1st jan 1970"

@llm.ai_callable()
async def timestamp_convert(
    self,
    timestamp: Annotated[
        int, llm.TypeInfo(
            | description="the number of seconds that passed since epoch (1st jan 1970)"
        ],
    ) -> str:
    """Converts passed timestamp into year,month,day,hour,minute,second format"""
    dt_object = datetime.date.fromtimestamp(timestamp)

    return str(dt_object)

```

**Figure 28.** Tools the LiveKit agent has access to

```

elp you today?", "interrupted": false, "speech_id": "e37ef02a7056", "pid": 3888, "job_id": "AJ_UGTvsmvvmL06"}  

2024-12-08 16:43:27,913 - INFO livekit.agents - job exiting {"reason": "", "pid": 3686, "job_id": "AJ_PWRoTvfNsNLpD"}  

2024-12-08 16:43:27,911 - WARNING livekit - livekit::rtc_engine - received session close: "signal client closed: \\\"stream closed\\\" UnknownReason Residue {"pid": 3686, "job_id": "AJ_PWRoTvfNsNLpD"}  

2024-12-08 16:43:27,912 - DEBUG livekit.agents - shutting down job task {"reason": "", "user_initiated": false, "pid": 3  

686, "job_id": "AJ_PWRoTvfNsNLpD"}  

2024-12-08 16:43:27,915 - DEBUG livekit.agents - http_session(): closing the httpclient ctx {"pid": 3686, "job_id": "AJ_  

PWRoTvfNsNLpD"}  

2024-12-08 16:43:27,915 - DEBUG livekit.agents - http_session(): creating a new httpclient ctx {"pid": 3686, "job_id": "  

AJ_PWRoTvfNsNLpD"}  

2024-12-08 16:43:28,234 - WARNING livekit.agents - failed to connect to livekit, retrying in 0s: worker connection close  

d unexpectedly  

2024-12-08 16:43:28,278 - INFO livekit.agents - registered worker {"id": "AW_jGzthBPUJFGC", "region": "UK", "protocol":  

15, "node_id": "NC_OLONDON1B_6GX9t5d9jFW0"}  

2024-12-08 16:43:28,515 - DEBUG livekit.agents.pipeline - received user transcript {"user_transcript": " What day is it  

today?", "pid": 3888, "job_id": "AJ_UGTvsmvvmL06"}  

2024-12-08 16:43:28,893 - DEBUG livekit.agents.pipeline - validated agent reply {"speech_id": "celed5b04d3f", "pid": 388  

8, "job_id": "AJ_UGTvsmvvmL06"}  

tried to print  

2024-12-08 16:43:29,303 - DEBUG livekit.agents.pipeline - speech playout finished {"speech_id": "celed5b04d3f", "interru  

pted": false, "pid": 3888, "job_id": "AJ_UGTvsmvvmL06"}  

2024-12-08 16:43:29,303 - DEBUG livekit.agents.pipeline - executing ai function {"function": "get_time", "speech_id": "c  

eled5b04d3f", "pid": 3888, "job_id": "AJ_UGTvsmvvmL06"}  

2024-12-08 16:43:29,304 - INFO voice-agent - Fetching time with get_time function {"pid": 3888, "job_id": "AJ_UGTvsmvvmL06"}  

2024-12-08 16:43:29,745 - DEBUG livekit.agents.pipeline - speech playout finished {"speech_id": "celed5b04d3f", "interru  

pted": false, "pid": 3888, "job_id": "AJ_UGTvsmvvmL06"}  

2024-12-08 16:43:29,745 - DEBUG livekit.agents.pipeline - committed agent speech {"agent_transcript": "", "interrupted":  

false, "speech_id": "celed5b04d3f", "pid": 3888, "job_id": "AJ_UGTvsmvvmL06"}

```

**Figure 29.** LiveKit agent unsuccessful tool chain

Similar to Fig.27, Fig.29 shows the LiveKit agent attempt tool chaining, using the tools in Fig.28.

Underlined in red is the user query (*What day is it today?*). The agent tries to answer the query by first calling the *get\_time* tool (underlined in blue). After that, the agent seems to freeze: it is unable to answer the query because it is unable to chain tools.

These results show that the LiveKit approach allows for speech-to-speech interaction but has limited functionalities as far as converting queries into robot instructions go. On the contrary, LangChain only supports text-to-text but it is capable of tool chaining: this means, provided it has access to basic tools, it could perform very complex actions by chaining up several simple tools together.

## 2.2 Simulation results

### 2.2.1 Mapping the robot with *slam\_toolbox*

Before we can get our robot navigating inside the world, we must first generate a map through which it can later localise and navigate autonomously. The most popular approach is to use a ROS2 package called **slam\_toolbox** which as the name suggests, performs SLAM. It only needs to subscribe to the /scan topic which our 2d lidar is publishing to and the necessary transforms are made.

## **Map -> odom -> base\_footprint -> base\_link**

**Figure 30.** Operation order of our robot

There are 2 main choices for performing mapping with slam\_toolbox: **online\_async** and **online\_sync launch**

**Online async** is an asynchronous mapping algorithm that only uses the most recent /scan data to generate the map. It works well in resource constrained environments where some messages may be lost and has lower latency than the synchronous slam

Online sync slam is the opposite. It uses all the scan data even if the message received is old. This can in turn make a more accurate map however depending on the speed of the robot and how resource constrained the system, also make a worse map and take too much computation leaving less time for other tasks.

For our setup we went with the first choice: online async launch due to our computers not being that powerful.

In order to map out the world i drive around my sending **Twist** messages to the **/cmd\_vel** topic. This is a vital part of how the vast majority of mobile robots move around. A twist message is composed of 2 vectors, 1 for the linear and the other angular. Each vector has 3 elements: velocity about the x,y and z axis. Hence a Twist message can fully represent the state of some object in 3d space.

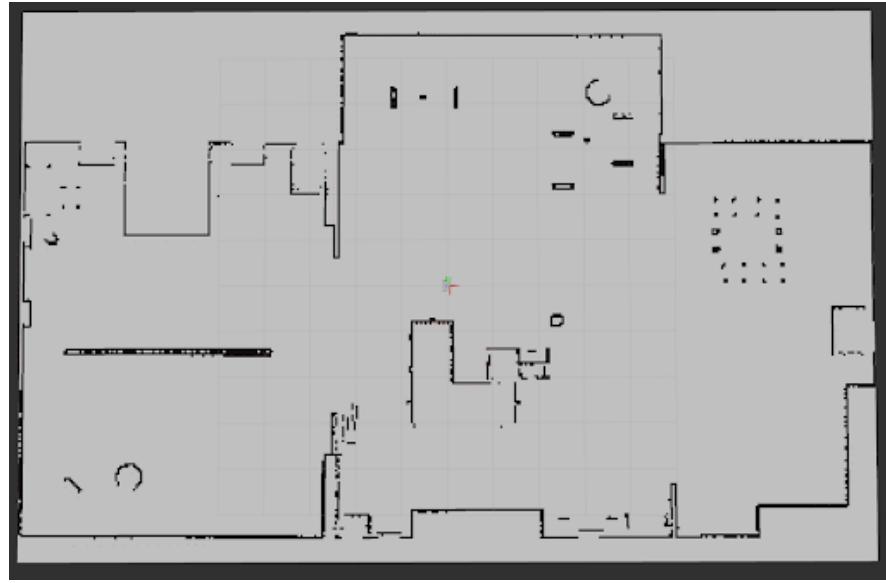
In practice we only utilize 2 of the values and leave the rest set to 0. We use the values **linear.x** and **angular.z**

This means we can drive the robot forward with some velocity (can be negative) in metres per second and revolve it around the robots z axis in **radians per second**.

After driving the robot around the house a few times we were able to generate the map in Fig.32 from the environment in Fig.31.



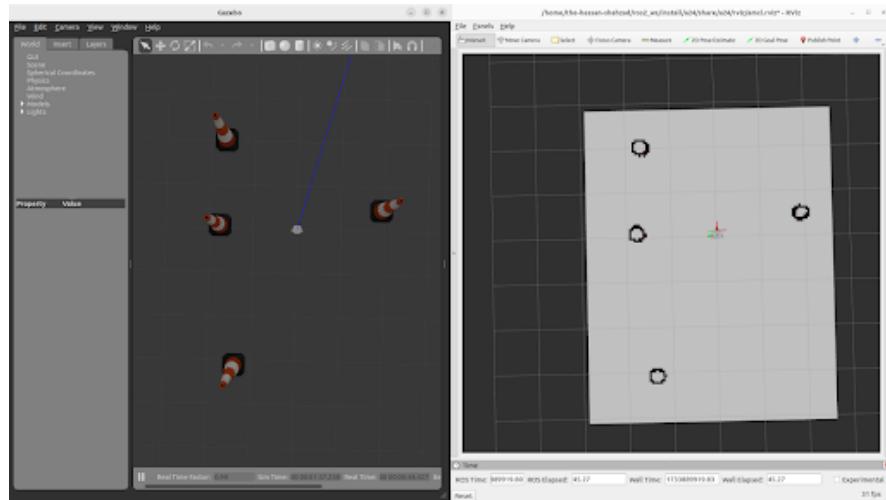
**Figure 31.** Real environment



**Figure 32.** Mapped environment

Now this map being generated is being published to the `/map` topic, however if we were to close this program then the map would disappear and we would have to generate it again. In order to save the map we run the command: `ros2 run nav2_map_server map_saver_cli -f name_of_map` This will save the map in the directory we ran the command in. It generates 2 files which are a `map_name.pgm` and `map_name.yaml`.

These 2 files must be inside the same directory of the same name for the map to be reused. The `pgm` file can be thought of as the raw uncompressed, unserialized data for the map. It is a 2d occupancy grid of 3 states: definitive occupied (when the probability it is occupied is above the threshold), definitive unoccupied and uncertain. The threshold values along with data about the maps centre is stored in the `yaml` file.



**Figure 33.** Another environment mapped

### 2.2.2 Localizing with AMCL

Now that a map has been generated and saved to the maps folder of the package, we can begin trying to localise the robot within the map. A technique called AMCL short for Adaptive Monte Carlo Localization is employed to do this.

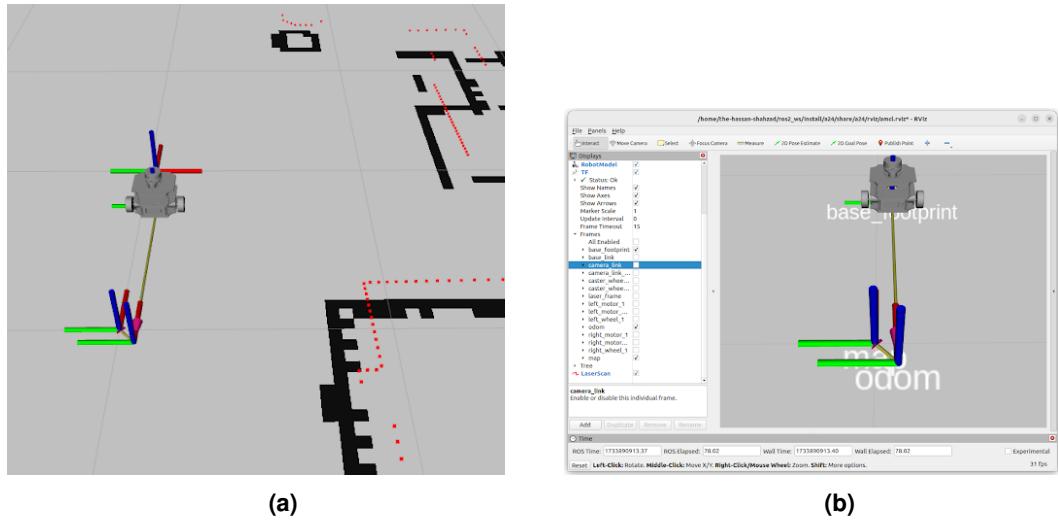
In order for ROS2 to have access to the map we run `nav2_map_server` and specify the `.yaml` file for the map we generated and this will load the map.

Then we run `ros2 launch nav2_amcl amcl` in order to localize. The important thing is that we must first set an initial pose. This tells AMCL where we think the robot is on the map. If we select a initial pose on the map that is far off its true value then the quality of the localization will suffer but over time resolve itself. If we choose a good guess for the initial pose then our localization will also be good because our odometry is accurate.

It is time to mention what **odometry** is and its relevance in autonomous mobile robots. Odometry is where the robot thinks it is in the world. It can be calculated by integrating over the robots wheel encoders or performing position tracking on a robots IMU if it has one. There are also techniques like visual odometry and laser odometry which estimate how far a robot has been displaced simply by observing sensor data.

The key problem is that odometry drifts and quickly strays from its true value. It can be caused by a multitude of reasons but for wheeled mobile robots, the key culprit is wheel slippage between the ground. This causes the robot to think it has traveled further than it actually has. This is especially a problem in areas that have slippage like carpeted rugs and transitioning between different textured surfaces.

Once we have set the initial pose we can begin driving the robot around manually and observe how far the transform between **map** and **odom** is. The further it is, the more the SLAM software is trying to correct the odometry and the closer it is, the more reliable the localization.



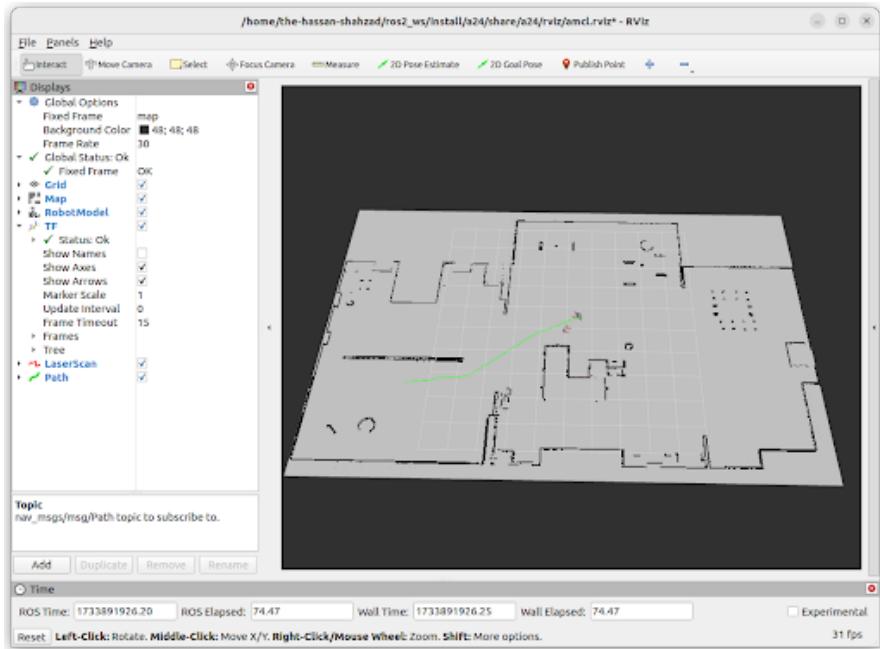
**Figure 34.** Odometry and map transforms close to one another

### 2.2.3 Navigating inside the world

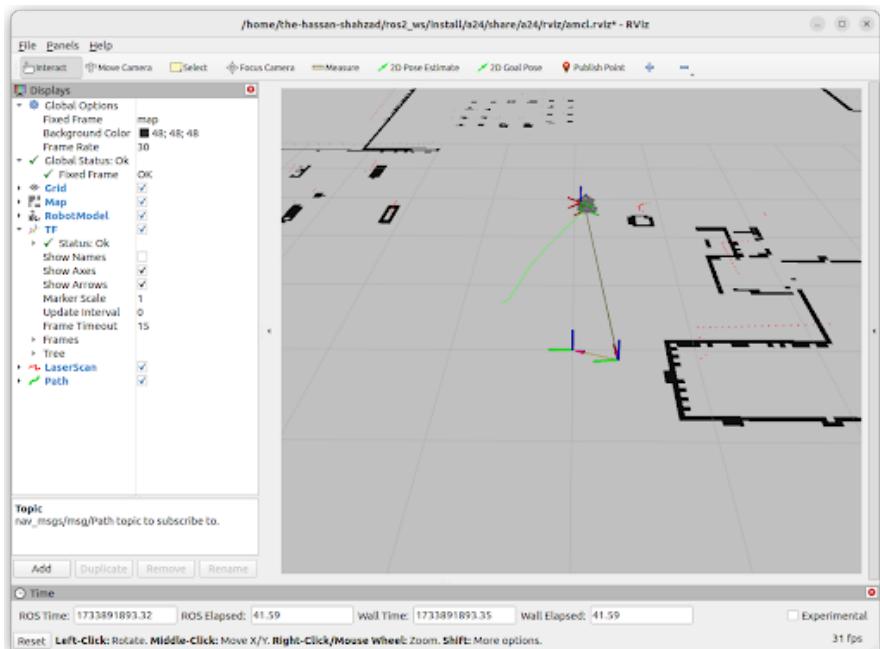
Now that we can localise the robot inside the map, the next step is to make the robot navigate. In the previous stages we manually published twist messages to the robot however now the software will send the commands by itself to follow a path that it generates.

To do this I use the **nav2 stack** and run the following launch file `ros2 launch nav2_bringup navigation.launch.py` along with specifying a parameters file to configure the behaviour of the navigator.

After setting the initial pose of the robot, we can send a goal pose and the robot will try to get to that match that pose. The pose consists of a point in x,y,z along with a quaternion to represent the angles. A quaternion is used because it is more computationally efficient than euler angles.



**Figure 35**



**Figure 36**

#### 2.2.4 Using the Nav2 Simple Commander API

In order to interact programmatically with the nav2 stack we can use an api which greatly speeds up the complexity of programming autonomous robots to follow waypoints and complete missions.

The basic layout is as follows:

```

import rclpy
from nav2_simple_commander.robot_navigator import BasicNavigator
from geometry_msgs.msg import PoseStamped
import tf_transformations

def main():
    rclpy.init()
    navigator = BasicNavigator()

    # Define the quaternion from Euler angles
    qx, qy, qz, qw = tf_transformations.quaternion_from_euler(0.0, 0.0, 0.0)

    # Create the initial pose message
    initial_pose = PoseStamped()
    initial_pose.header.frame_id = 'map'
    initial_pose.header.stamp = navigator.get_clock().now().to_msg()

    # Set the position and orientation
    initial_pose.pose.position.x = 0.0
    initial_pose.pose.position.y = 0.0
    initial_pose.pose.position.z = 0.0
    initial_pose.pose.orientation.x = qx
    initial_pose.pose.orientation.y = qy
    initial_pose.pose.orientation.z = qz
    initial_pose.pose.orientation.w = qw

    # Set the initial pose
    navigator.setInitialPose(initial_pose)

```

**Figure 37.** Set the initial pose

```

def create_pose_stamped(self, navigator, position_x, position_y, rotation_z):
    q_x, q_y, q_z, q_w = tf_transformations.quaternion_from_euler(0.0, 0.0, rotation_z)
    goal_pose = PoseStamped()
    goal_pose.header.frame_id = 'map'
    goal_pose.header.stamp = navigator.get_clock().now().to_msg()
    goal_pose.pose.position.x = position_x
    goal_pose.pose.position.y = position_y
    goal_pose.pose.position.z = 0.0
    goal_pose.pose.orientation.x = q_x
    goal_pose.pose.orientation.y = q_y
    goal_pose.pose.orientation.z = q_z
    goal_pose.pose.orientation.w = q_w
    return goal_pose

def goal_callback(self, msg):
    if len(msg.data) == 3:
        goal_x = msg.data[0]
        goal_y = msg.data[1]
        goal_theta = msg.data[2]

        self.get_logger().info(f"Goal: x={goal_x}, y={goal_y}, theta={goal_theta}")

        goal_pose = self.create_pose_stamped(self.navigator, goal_x, goal_y, goal_theta)
        self.navigator goToPose(goal_pose)
        while not self.navigator.isTaskComplete():
            feedback = self.navigator.getFeedback()
            self.get_logger().info(f"Feedback: {feedback}")

        self.get_logger().info(f"Result: {self.navigator.getResult()}")

    else:
        self.get_logger().error("Invalid goal pose message received")

```

**Figure 38.** Go to global pose

The function **create\_pose\_stamped** is responsible for generating the correct pose given only a desired x,y coordinate and rotation around the z axis in radians

Using these 2 nodes, we can move the the robot to any unoccupied space in the map and even enabling

motions like spinning on the spot by altering the goal theta. This provided a higher level of control for the agent which other mobile robots controlled by LLM's do not have.

### 2.2.5 Additional Services programmed

In ROS2 we not only have nodes and launch files but also services and actions. The service in particular is incredibly useful for this project since it is a function that takes in some inputs and returns an output in one go, not constantly like how nodes typically work. This can be used to perform tasks inside ROS2 like check if a coordinate on a map is occupied or what the distance measurement is from the depth camera for a given x,y pixel. That's exactly what has been implemented. We made the following services in ROS2 to give the LLM agent a higher level of control over ROS2 and give it more insight.

- IsOccupied

- Take\_image

- Coord\_2\_depth

The first service sees if the robot can actually go to the point the user asks to go to. It returns 2 booleans: one to indicate if the command to check has gone through and the second to actually determine if that coordinate is occupied or not with a True or False value. There is a custom **srv** message to go along with this service:

```
srv > ≡ CheckCoordinates.srv
1   float64 x
2   float64 y
3   ---
4   bool success
5   bool is_occupied
```

(a)

```
# Check occupancy
if occupancy_value == -1:
    self.get_logger().info('Cell is unknown.')
    response.success = False
    response.is_occupied = False
elif occupancy_value >= 50:
    self.get_logger().info('Cell is occupied.')
    response.success = True
    response.is_occupied = True
else:
    self.get_logger().info('Cell is free.')
    response.success = True
    response.is_occupied = False

return response
```

(b)

Figure 39

```

def image_callback(self, msg):
    """Callback to store the latest image."""
    try:
        self.latest_image = self.bridge.imgmsg_to_cv2(msg, "bgr8")
    except Exception as e:
        self.get_logger().error(f"Error processing image: {e}")

def handle_save_image(self, request, response):
    """Service handler to save the latest image."""
    if self.latest_image is None:
        response.success = False
        response.message = "No image received yet."
        return response

    try:
        # Generate filename
        filename = os.path.join(self.output_directory, f"image.png")

        # Save the image
        cv2.imwrite(filename, self.latest_image)
        self.get_logger().info(f"Saved image as {filename}")

        response.success = True
        response.message = f"Image saved as {filename}"
    except Exception as e:
        response.success = False
        response.message = f"Error saving image: {e}"

    return response

```

**Figure 40.** The takeImage service takes an image of what the camera sees and saves it as .png for other functions/tools to use

The next service is **coord\_2\_depth**. This is used to find how far a RGB pixel is in the depth image. While this service provides low level inference from the ROS2 system, if coupled with more advanced computer vision or other advanced tools, it could be used to go towards items within the house

### 2.3 Overall Results with high level tools and ROS2 bindings

To finally merge the work done with LangChain with this ROS2 package, we create 6 distinct tools which can be used to effectively interact with the simulated world.

```

def execute_ros_command(command: str) -> Tuple[bool, str]:
    """
    Execute a ROS2 command.

    :param command: The ROS2 command to execute.
    :return: A tuple containing a boolean indicating success and the output of the command.
    """

    # Validate the command is a proper ROS2 command
    cmd = command.split(" ")
    valid_ros2_commands = ["node", "topic", "service", "param", "doctor"]

    if len(cmd) < 2:
        raise ValueError(f"{command} is not a valid ROS2 command.")
    if cmd[0] != "ros2":
        raise ValueError(f"{command} is not a valid ROS2 command.")
    if cmd[1] not in valid_ros2_commands:
        raise ValueError(f"ros2 {cmd[1]} is not a valid ros2 subcommand.")

    try:
        output = subprocess.check_output(command, shell=True).decode()
        return True, output
    except Exception as e:
        return False, str(e)

```

**Figure 41.** Function used to execute a ROS2 command

The following function executes a ROS2 command in terminal using the subprocess function. It first validates it to make sure this is a valid ROS2 command by splitting the message and performing a basic syntax analysis.

```

@tool
def go_to_goal(
    x: float,
    y: float,
    theta: float
):
    """
    Move the robot to a specific point in the world with an x,y coordinate and angle theta.
    Check if the point is occupied before sending the goal

    :param x: The x coordinate of the point to move to in meters.
    :param y: The y coordinate of the point to move to in meters.
    :param theta: The angle to move to in radians.
    """

    cmd = f"ros2 topic pub /nav2_goal_pose std_msgs/Float64MultiArray \"data: [{(x), (y), (theta)}]\" --once"
    success, output = execute_ros_command(cmd)

    if success:
        print(f"Successfully sent command to move to ({x}, {y}, {theta})")
    else:
        print(f"Failed to send command ({x}, {y}, {theta}): {output}")

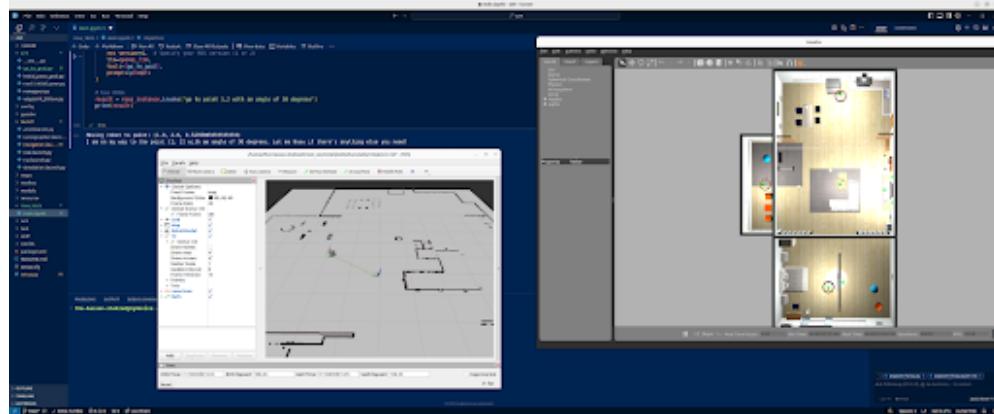
    # Implement the logic to move the robot to the specified point
    print(f"Moving robot to point: ({x}, {y}, {theta})")

```

**Figure 42.** Tool to move the robot to some Nav2 goal pose

This tool moves the robot so some point on the map with a desired pose denoted by an angle theta about the z axis.

When calling the tool using the LLM we got the following results:



**Figure 43**

```

# Use ROSA
result = rosa_instance.invoke("go to point 1,2 with an angle of 30 degrees")
print(result)

```

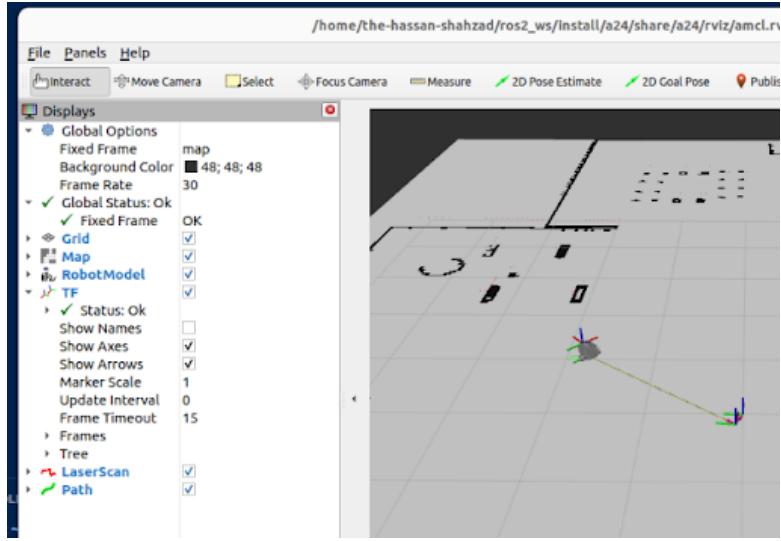
**Figure 44.** query

```

Moving robot to point: (1.0, 2.0, 0.5235983333333333)
I am on my way to the point (1, 2) with an angle of 30 degrees. Let me know if there's anything else you need!

```

**Figure 45.** query results



**Figure 46.** robot going to the goal

Figs.44, 45, 46 show the robot being queried to go to  $x=1$ ,  $y=2$ , with 30 degrees orientation and succeeding in answering that query.

```
@tool
def check_map_occupancy(x: float, y: float) -> bool:
    """
    Check if a specific point in the map is occupied. Called when a user wants to move the robot to a point. Cannot go to a point that is occupied.
    output is a boolean true or false. true means the coordinate is occupied. false means it isn't
    """
    cmd = "ros2 service call /check_coordinate a24_interfaces/srv/CheckCoordinates '{(x: %f, y: %f)}'"% (x,y)
    print(cmd)
    success, output = execute_ros_command(cmd)
    print("Checking map occupancy for point: (%f, %f)"% (x,y))
    print("Output: %s"% output)
    is_occupied_val = "is_occupied=True" in output
    return is_occupied_val # tells us if the map is occupied
```

**Figure 47.** Tool to check if the map is occupied

This tool publishes a service call to our custom service and gets the results. This tells if the robot can even go to the point the user says to go to. Improving the reliability of the system. Here are the results:

```
# Use ROSA
# result = rosa_instance.invoke("check if the point 1,1 is occupied")
result = rosa_instance.invoke("move to point 1,1")
print(result)

✓ 9.1s
ros2 service call /check_coordinate a24_interfaces/srv/CheckCoordinates "x: 1.0, y: 1.0"
Checking map occupancy for point: (1.0, 1.0)
Output: requester: making request: a24_interfaces.srv.CheckCoordinates_Request(x=1.0, y=1.0)

response:
a24_interfaces.srv.CheckCoordinates_Response(success=True, is_occupied=False)

Successfully sent command to move to (1.0, 1.0, 0.0)
Moving robot to point: (1.0, 1.0, 0.0)
I have successfully moved to the point (1, 1). If you have any other instructions, feel free to let me know!
```

**Figure 48**

As you can see it first checks if the coordinate is occupied before attempting to go there. We did not have to explicitly mention to check if it is occupied, the robot did so by itself.

### 2.3.1 Tool to take an image and save it

In order to save what the robot is seeing in its camera, another tool is developed which takes an image. The output is an **image.png** file saved in a folder called **images** for other functions to have access to it

```

@tool
def take_image():
    """
    Takes an image of what the robot is seeing in its camera and saves it as a .jpg image to this directory of the same "image.jpg"
    """
    cmd = f"ros2 service call /save_image std_srvs/srv/Trigger"
    success, output = execute_ros_command(cmd)
    print(f"Success: {success}")
    print(f"Output: {output}")

```

**Figure 49.** The tool use the custom service responsible for taking an image based on a trigger message. It returns if the image has been taken successfully

### 2.3.2 Tool to describe the contents of an image

This tool is essential and has 2 parts to it. On one side we made a python function saved in a separate file. This function uses the OpenAI API to take describe an image and just return the string of the image contents:

```

import base64
from openai import OpenAI

# Initialize OpenAI client
client = OpenAI()

def analyze_image(image_path, model="gpt-4o-mini"):
    """
    Analyze an image using the OpenAI API and return the content of the response.

    Args:
        image_path (str): Path to the image file (.png or .jpg).
        model (str): The model to use for the analysis.

    Returns:
        str: The content of the assistant's response.
    """

    # Encode the image as base64
    def encode_image(path):
        with open(path, "rb") as image_file:
            return base64.b64encode(image_file.read()).decode("utf-8")

    # Determine MIME type based on file extension
    if image_path.lower().endswith(".png"):
        mime_type = "image/png"
    elif image_path.lower().endswith(".jpg") or image_path.lower().endswith(".jpeg"):
        mime_type = "image/jpeg"
    else:
        raise ValueError("Unsupported file format. Please use .png, .jpg, or .jpeg.")

    # Encode the image
    base64_image = encode_image(image_path)

    # Send the request to the OpenAI API
    response = client.chat.completions.create(
        model=model,
        messages=[
            {
                "role": "user",
                "content": [
                    {
                        "type": "text",
                        "text": "What is in this image?",
                    },
                    {
                        "type": "image_url",
                        "image_url": {
                            "url": f"data:{mime_type};base64,{base64_image}"
                        },
                    },
                ],
            }
        ],
    )

    # Return just the content
    return response.choices[0].message.content

```

**Figure 50.** As you can see we just ask the LLM “what is in this image” and return only the relevant parts of its response

```

@tool
def describe_image() -> str:
    """
    Describes the image that the robot took. before using this tool take an image with the take_image tool to get the up to date image
    """

    cmd = f"ros2 service call /save_image std_srvs/srv/Trigger"
    success, output = execute_ros_command(cmd)

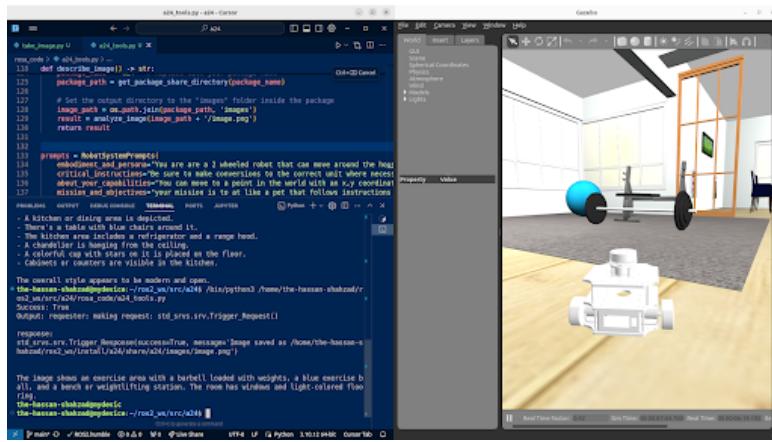
    package_name = 'a24' # Replace with your package name
    package_path = get_package_share_directory(package_name)

    # Set the output directory to the "images" folder inside the package
    image_path = os.path.join(package_path, 'images')
    result = analyze_image(image_path + '/image.png')
    return result

```

**Figure 51.** The take image tool accesses the image taken by the previous tool and then passes that path to image into the function and returns the response

Here are the results for the tool:



**Figure 52**

```

The image shows an exercise area with a barbell loaded with weights, a blue exercise ball, and a bench or weightlifting station. The room has windows and light-colored flooring.

```

**Figure 53.** Image contents being described correctly

Finally the last higher level tool made was to find how far a pixel is in the depth camera:

```

@tool
def coord_2_depth(x: float, y: float) -> float:
    """
    Find the how far away the robot is from a point in the world by specifying an x,y coordinate and retrieving the depth value from the depth camera.
    Units of distance are metres and the coordinates are in pixels along and up the camera image.
    Specify the distance to 3 significant figures.
    """
    cmd = f"ros2 service call /calculate_distance a24_interfaces/srv/CoordDistance '{x} {y}'"
    success, output = execute_ros_command(cmd)

    if success:
        match = re.search(r"distance:(\d\.\d+)", output)
        if match:
            return float(match.group(1))
        else:
            return float('nan')
    else:
        return f"Failed to calculate distance: {output}"

```

**Figure 54.** Tool for depth of image

This tool also uses our custom service and returns the distance the tool gives us here are the results for the tool:

```
/bin/python3 /home/the-hassan-shahzad/ros2_ws/src/a24/rosa_code/a24_tools.py
the-hassan-shahzad@mydevice:~/ros2_ws/src/a24$ /bin/python3 /home/the-hassan-shahzad/ros2_ws/src/a24/rosa_code/a24_tools.py
The distance to the coordinate (100, 200) on the camera is approximately 7.75 meters.
the-hassan-shahzad@mydevice:~/ros2_ws/src/a24$ /bin/python3 /home/the-hassan-shahzad/ros2_ws/src/a24/rosa_code/a24_tools.py
The distance to the coordinate (100, 200) on the camera is approximately 8.91 meters.
the-hassan-shahzad@mydevice:~/ros2_ws/src/a24$ /bin/python3 /home/the-hassan-shahzad/ros2_ws/src/a24/rosa_code/a24_tools.py
The distance to the coordinate (100, 200) on the camera is approximately 9.05 meters.
the-hassan-shahzad@mydevice:~/ros2_ws/src/a24$ /bin/python3 /home/the-hassan-shahzad/ros2_ws/src/a24/rosa_code/a24_tools.py
The distance to the coordinate (100, 200) on the camera is approximately 4.69 meters.
the-hassan-shahzad@mydevice:~/ros2_ws/src/a24$ /bin/python3 /home/the-hassan-shahzad/ros2_ws/src/a24/rosa_code/a24_tools.py
The distance to the coordinate (100, 200) on the camera is approximately 4.817 meters.
the-hassan-shahzad@mydevice:~/ros2_ws/src/a24$
```

**Figure 55.** results

We drove the robot before calling the next tool so the distance changed. Small changes in the value are normal to due the dispersion within the depth camera. As you can see all the values are to 3 significant figures, as requested by the tool:

```
"""
Find the how far away the robot is from a point in the world by specifying an x,y coordinate and retrieving the depth value from the depth camera.
Units of distance are metres and the coordinates are in pixels along and up the camera image.
Specify the distance to 3 significant figures.
"""
```

**Figure 56**

### 2.3.3 Robot instructions

The Robot system prompts are vital to getting your robot to behave we intended and imitate the personality you set it out to have. For our robot we put the following:

```
prompts = RobotSystemPrompts(
    embodiment_and_persona=(
        "You are a 2-wheeled robot that can move around the house using SLAM with a 2D lidar. "
        "You can see the world around you and navigate to specific points in the house."
    ),
    critical_instructions=(
        "Be sure to make conversions to the correct unit where necessary. "
        "Do not go to a point that is occupied in the map."
    ),
    about_your_capabilities=(
        "You can move to a point in the world with an x, y coordinate and angle theta. "
        "You have a camera to take pictures and infer from the images. "
        "You also have a depth camera to find the distance to objects in the world."
    ),
    mission_and_objectives=(
        "Your mission is to act like a pet that follows instructions from the homeowners."
    )
)
```

**Figure 57.** robot prompts

The robot system prompts are straight forward and make sure that the robot is well aware of what it can and cannot do

## 3 FULL PACKAGE LAYOUT FOR A24

The a24 ROS 2 package is designed for voice-to-action interaction, leveraging SLAM Toolbox, the Nav2 stack, and the Rosa software. Below is an overview of the package's layout, focusing on nodes, launch files, configurations, and other relevant components:

### 3.1 Nodes

- **Voice-to-Action Nodes:**

- coord\_2\_depth.py: Converts coordinate data to depth values.
- go\_to\_goal.py: Handles navigation to a specific goal location.
- initial\_pose\_goal.py: Sets the initial pose of the robot for localization.
- nav2\_initial\_pose.py: Integrates Nav2 stack for setting the robot's pose.

- `isOccupied.py`: Checks if a specific map location is occupied.
- `remapper.py`: Remaps topics or data for compatibility between components.
- `take_image.py`: Captures images, possibly for debugging or perception.
- `waypoint_follow.py`: Implements waypoint-based navigation for the robot.

- **Agent Code (Integration with LangChain and LiveKit):**

- LangChain-based tools and agents are structured in `agent_code/langchain_agent`, providing language-driven control and task automation using tools like `agent_OOP.py` and `ros_tools.py`.
- LiveKit agents in `agent_code/livekit_agent` include `agent_livekit.py` and other supporting files like `livekit_tools_ros.py`, enhancing ROS2 and voice interaction integration.

- **Rosa Integration:**

- `rosa_code/a24_tools.py` and `rosa_code/image_describer.py` provide tools for interpreting images and managing Rosa-related functionality.

### 3.2 Launch Files

The launch files, located in the `launch` directory, orchestrate the package's subsystems:

- **SLAM and Navigation:**

- `amcl.launch.py`: Launches the Adaptive Monte Carlo Localization (AMCL) stack.
- `cartographer.launch.py`: Initializes Cartographer for SLAM-based mapping.
- `navigation.launch.py`: Sets up Nav2 for navigation.

- **Simulation and Real Robot Operation:**

- `simulation.launch.py`: Configures the simulation environment, including Gazebo.
- `real.launch.py`: Sets up the package for operation on a physical robot.
- `rsp.launch.py`: Launches robot state publisher for URDF publishing.

### 3.3 Configuration Files

Located in the `config` directory, these YAML files provide parameters for critical functionalities:

- **Navigation:** `nav2_params.yaml` configures the Nav2 stack.
- **SLAM:** `mapper_params.yaml` defines parameters for SLAM Toolbox.
- **Controller Settings:** `controller.yaml` specifies robot control parameters.

### 3.4 Maps and World Files

- **Maps (in `maps` directory):**

- `cones.pgm` and `cones.yaml`: Map and metadata for a cones environment.
- `home.pgm` and `home.yaml`: Map and metadata for a home environment.

- **Worlds (in `worlds` directory):**

- `cones.world`: A Gazebo simulation world for navigation testing.
- `home.world`: Simulated home environment.

### 3.5 URDF and Mesh Files

- **URDF Models (in `urdf` directory):**

- Includes robot description files for simulation (`a24_sim.xacro`) and real operation (`a24_real.xacro`)
- Sensor-specific files like `depth_camera.xacro` and `lidar.xacro`.

- **Meshes (in `meshes` directory):** 3D models of robot components, such as wheels, motors, and the base.

### 3.6 Visualizations

- **RViz Configuration (in `rviz` directory):**

- Visualization setups for various scenarios: `amcl.rviz`, `cartographer.rviz`, and `navigation.rviz`.

### 3.7 Testing

- **Unit Tests (in `test` directory):**

- Scripts like `test_flake8.py` and `test_pep257.py` ensure code quality and adherence to standards.

### 3.8 Miscellaneous

- **Simulation Parameters:** `gazebo/gazebo_params.yaml` for Gazebo simulation settings.
- **Agent Integration:** The `agent_code` directory facilitates advanced AI and task management with external libraries and frameworks like LangChain and LiveKit.
- **Images and Models:** Placeholder for additional resources or assets in `images` and `models` directories.

This layout supports a modular design, allowing seamless integration of voice commands, SLAM, and navigation functionalities.

## 4 DISCUSSION

As discussed in section 2.1.3, our LiveKit approach and our LangChain approach each have their own strengths and weaknesses: LiveKit supports speech-to-speech but not tool chaining, and LangChain is text-to-text but does support tool chaining.

Both approaches have been capable of transforming human queries into function calls, thus greatly simplifying the interactions between the user and the simulated robot. We have succeeded in our objective for this project, but we believe this subject has more potential than what we have covered here. We were limited by the report's deadline, but had we had more time, we could have tried using our LangChain and LiveKit agents on a real robot.

## 5 CONTRIBUTIONS

### 1 Methods:

#### 1.1 First approach: LangChain

- research: 50/50 (equal contributions from both authors)
- agent python files: Andrew
- tools: Andrew

#### 1.2 Second approach: LiveKit

- research: 50/50

- agent files: 50/50
- tools: 50/50

### **1.3 Simulation**

- all Hassan

## **2 Results**

### **2.1 LangChain and LiveKit Comparison**

- Andrew

### **2.2 Simulation results**

- Hassan

### **2.3 Overall Results with high level tools and ROS2 bind**

- Hassan

### **3 Full package Layout for a24**

- Hassan

**Project overall:** 50/50

## **ACKNOWLEDGMENTS**

Once again, this report was made possible by [LangChain](#), [LiveKit](#), and the [ROSA](#) project.