Lessons Learned Fetching A Beer

Copyright © 2026 By Michael Wimble. Version 1.

Table of Contents

- Lessons Learned Fetching A Beer
  - Copyright © 2026 By Michael Wimble. Version 1.
  - Table of Contents
  - A couple of general observations.
  - On the Use of AI
    - AI always wants to tell you how brilliant you are.
    - Always remember that AI doesn't understand anything.
    - AI will blindly ignore your "you must" and "you must not" instructions.
    - AI will confidently say wrong things.
    - AI will often optimize for what it thinks you meant, not what you said.
    - AI is not particularly good at writing code.
    - When asking for analysis and critique, get multiple opinions.
  - Gazebo is a full physics simulator.
  - It seemed like a good idea at the time.
  - ROS and the spin cycle
  - ROS tools — know them and use them well.
  - Behavior Trees—the big hammer.
  - Groot—suddenly it's extremely costly.
  - Tool creation for behavior tree debugging.
  - Object detection.
  - Creating the simulation world.
  - Simulating the real hardware.
  - But wait, does it work on a Macintosh?
  - Articles and papers need an end.

I just spent a week grinding through the fiddly low-level details of Sigyn—my autonomous robot assistant project—and finally surfaced long enough to do something interesting with it. The Robotics Society of Southern California has issued a Can Do Challenge which is essentially a challenge to perform the "Fetch Me a Beer" task so near and dear to roboticists everywhere. During the last few days of the week I worked from 10 in the morning until 2 the following morning, with the heavy assistance of VS Code, Copilot and various AI engines. My goal was to achieve a first pass at solving the problem in the Gazebo simulator, with the solution working on the real robot sometime later. And I achieved that.

I achieved it but it was not easy. I wanted to share some lessons learned along the way as well as talk about how some of the components of the first-pass solution work.

The "Fetch A Beer" task is deceptively simple: The robot must start at a known location, navigate to a location where the beer is kept (like a kitchen table or fridge), locate the beer, pick it up, and bring it back to the user. Every one of those steps involves a dozen sub-problems in perception, navigation, manipulation, and decision making.

A couple of general observations.

- As Phil Karlton famously said, "There are only two hard things in Computer Science: cache invalidation and naming things." This also applies to naming Behavior Tree nodes and blackboard variables. Spend time thinking about the names you use. It will pay off later.
- Having a fast processor will extend your lifespan. My robot and my main development workstation both run on AMD 7900X processors, and the speed difference compared to older hardware is non-trivial when compiling or running heavy simulations.
- Debugging multiple threads is hard — start learning now. It will take a very long time to become proficient at it but it will be indispensable later on.

- VS Code is god's gift to programmers. Learn to use it well. Learn to use the extensions.
- Think about why you define any duration value. Or distance, tolerance, etc. Try not to make arbitrary choices. Think about the implications of your choices.

On the Use of AI

AI always wants to tell you how brilliant you are.

It actually always believes you're an idiot to be humored and ignored.

Always remember that AI doesn't understand anything.

It's a big dart board trying to predict the next letter to type that will please you.

AI will blindly ignore your "you must" and "you must not" instructions.

Not always but often enough. And it won't tell you it is ignoring you. Unless you follow up by asking well-crafted questions.

AI will confidently say wrong things.

See the above points. AI understands nothing and wants to flatter you.

AI will often optimize for what it thinks you meant, not what you said.

See the above points. Unless you are very diligent and check everything it is thinking and everything it does, it is going to ignore what you asked for and do what it thinks you should have asked for. It's very much a "Don't you worry your pretty little head about it, I'll take care of everything" kind of attitude.

AI is not particularly good at writing code.

The big problem is that often you have to be pretty skilled to recognize that the code it is writing is wrong. Sometimes the code is pretty subtly wrong.

When asking for analysis and critique, get multiple opinions.

Different AIs have their strengths and weaknesses. Getting multiple opinions will help you see the full picture.

A lot of my time, well over half of the week, was spent on fixing the code that AI generated.

On the other hand, AI can be absolutely brilliant at times. Those few times make up for all the wasted time, if you don't count the hair loss. I could not have done this project in a week without the help of AI; it would have taken me at least 8 times as long. Used like a fallible junior engineer—fast, energetic, and requiring review—AI is still a force multiplier.

Gazebo is a full physics simulator.

You probably wish you could just throw a URDF model of your robot into Gazebo and have it work. Surely if you defined two fingers of a gripper and you closed those fingers on a can of soda it should be able to lift the can, right? Wrong. You have to define the physics of everything in your simulation. You have to define the mass, friction coefficients, stiffness, damping, and so forth of every object in your simulation. You have to define the forces that your motors can apply to your robot. You have to define the sensors that your robot uses to perceive the world. Getting all of these values right is a huge pain.

When I first tried lifting the can, the soda can just stayed on the table while the gripper, which was tightly closed around it, tried to lift the can off the table. The gripper fingers just slipped right past the can because the friction coefficients were wrong. As a test, I changed the code to move the can to the left instead of trying to pick it up and it quite nicely knocked the can on its side and the can spun in a circle on the table. So the physics simulation was working, I just had to get the friction parameters correct.

But putting in friction, even a ridiculously high friction coefficient, wasn't enough. The next problem was that the fingers of the gripper were not defined to have any reasonable stiffness, so when the gripper fingers closed

on the can, they just deformed around the can and never actually gripped it. I had to set the stiffness of the fingers to a very high value to get them to actually grip the can.

With masses and friction coefficients changed, the next problem was that the motors could no longer lift the gripper mechanism. The new gripper was now too heavy for the motors to lift. I had to increase the motor forces to get the gripper to lift the can.

None of what was wrong was obvious at all. There was a lot of head scratching and trial and error to get the physics parameters right. This one problem took more than 10 hours to solve because of all the interrelated parameters that I've never had to deal with before.

When you tell the robot to stop moving, it probably will, eventually. You have to consider settling time and inherent movement errors. You have to think about real velocities and accelerations so the robot doesn't try to do impossible maneuvers. When you recognize that the moving arm is in the correct position to grasp the soda can, by the time the robot actually stops moving, it may have overshot the target position.

Oh, and don't get me started on manually tuning Gazebo PID controllers to get proper response.

It seemed like a good idea at the time.

The idea for finding a can of soda and getting near enough to pick it up was based on a simple idea at the time. I would have a list of likely places that the soda might exist. It might be in the refrigerator, but it might be in the pantry or on the kitchen table. The robot would go to each of these locations in turn and look for the can. If it found the can, it would try to pick it up. If it couldn't find the can at any of the locations, it would give up.

The code quite nicely picked up the first location in the list and headed there. Then the idea was that the OAKD camera would look for the can. If it didn't immediately see it, it would rotate up to 360 degrees to try to find it. Well the field of view of the OAKD camera is pretty limited, so if the can wasn't close enough to be in the field of view, the robot would never find it. That's an issue I'll have to come back to later—adding a search pattern behavior.

Once the OAKD camera found the can, it would rotate the robot to more or less face the can and then move forward until it was theoretically close enough to pick up the can. This works well if you start out from a pose that will nicely allow you to get in an ideal position to pick up the can just by rotating to point towards the can and moving forward. Again, this is an issue I'll have to come back to later—computing a better approach path to the can.

This first design produces a somewhat coarse approach. Then I needed to fine tune the position of the robot and get the gripper aligned with the can. This was done by using the Pi camera mounted on the gripper to look down at the gripper and, while slowly raising the arm holding the gripper, the robot would look for the can in the picture. Once the can finally showed up, the idea was to stop when the vertical center of the can was aligned with the vertical center of the gripper. There is not yet any code to deal with any failure here.

The expectation was that coarse positioning would guarantee the Pi camera eventually saw the can as I raised the arm, and then I’d only need a small rotation to center the can in the gripper.

It took a couple of hours of changing the position of the Pi camera and the angle of tilt of the camera to get a position that would reliably see the can when the arm was raised. Getting the Pi camera to tell when the elevator has lifted so that the center of the can aligns with the vertical center of the gripper was problematic. Some of this relies on the OAKD camera using its stereo vision to estimate the position of the can. I'm probably relying too much on the simulation of distance detection from the OAKD. Later I'll add a distance sensor to the gripper.

Once at the right height, the next step is to extend the gripper forward to the can so that the can barely touches the back of the gripper, so that the fingers can close around the can. Figuring out how far to extend the gripper was a bit of a headache. For now I rely on the distance estimate from the OAKD camera to tell how far away the can is, and extend the gripper that far minus a small offset. This seems to work reasonably well in simulation but needs more work to be reliable on the real robot.

Then, the gripper fingers close around the can and lift it off the table. Figuring out how to determine the proper closing of the gripper to grasp the can lead to pulling out one's hair, I may use force sensors in the future.

Next is verifying that the can is actually in the gripper. This will be done by both the Pi camera and the OAKD camera looking at the can in the gripper. If both cameras see the can at the expected position, then the robot assumes it has successfully grasped the can. If either camera doesn't see the can, the robot assumes it has failed to grasp the can and will have to try again. I don't currently have any retry logic implemented. Actually, even the basic verification logic is not yet implemented.

Next the gripper needs to raise the can up a little bit so it doesn't try to drag the can along the table when it starts moving backwards.

Then the arm needs to retract as the first step towards lowering the can to be near the robot's body. This is a safer position for the can to be in while the robot turns and heads back to its starting position. But you don't want to lower the can too much or it might hit the robot body.

Before you can lower the arm, though, the robot needs to back up a little bit to get clear of the table. This is done by simply reversing the path it took to get to the can. This is not very robust, but it works for now. To be correct, the robot must spin around so that all of the LIDARs and other proximity sensors have a clear view of the environment as it backs away from the table.

Finally, the robot needs to return to its starting position. This is done by simply using the Nav2 stack to compute a new path back to the starting position. This works reasonably well and has its own ability to recover from minor obstacles in the way.

There remains a myriad of problems that you wouldn't naively think of at first. Here are some of the problems I encountered along the way.

ROS and the spin cycle

No, not the spin cycle of a washing machine. ROS is built around the concept of nodes that spin, processing incoming messages and timers. If you have a long running computation in a node, you have to be careful to periodically call spin() to allow the node to process incoming messages. If you don't do this, your node will appear to hang and not respond to incoming messages. ROS will just stop working correctly.

All of the behavior of the robot is built on Behavior Trees. Behavior Trees are made up of nodes that are called periodically to perform their work. If any of these nodes take a long time to complete, you have to use some of the special features of Behavior Trees to allow the node to yield control back to the Behavior Tree engine so that, ultimately, the ROS node can call spin() to process incoming messages. I'll discuss this more later on as I talk about reactive behavior.

But there are multiple, asynchronous things going on in the robot. There is the main ROS node that is running the Behavior Tree engine. There are action servers that are running to perform movement actions. There are sensor nodes that are publishing data. There are transform nodes that are publishing transforms. Especially, there are images coming in from the simulated cameras and those images are being processed by object detectors. All of these things have to be kept in sync and working correctly.

Even when using the multi-threaded executor, you have to deal with synchronization and deadlocks. That's a topic for another article.

ROS use of quality of service (QoS) settings is a big pain. Basically, relying on "Reliable" delivery of messages can be problematic, or at least it can cause a lot of downstream problems. You should design your software to deal with missing messages. "Best Effort" delivery is usually the better choice.

Between URDF files, SDF files, config files, and code files, you have to make sure that all of the parameters are consistent. If you change a parameter in one place, you have to make sure it is changed in all the other places. ROS doesn't naturally let you define interesting values in one place, it's up to you to be aware of all the places you need to fix things. Eventually I will solve this by using the M4 macro processor to generate all of the files from a single source of truth. I prefer M4 over `xacro` because M4 is a general purpose macro processor that can generate any kind of text file (URDF, SDF, Python, C++, etc.), whereas `xacro` is XML-specific and somewhat limited in its programming capabilities.

ROS tools — know them and use them well.

If you don't know how to use the ROS tools efficiently, you'll be trying to build a house with a rock and a chisel. Make sure you understand:

- Use of the "--field" option with using "ros2 topic echo" to limit the amount of data you get when looking at images or LIDAR data.
- Use of QoS settings to make sure you get the data you need.
- Use of rqt_graph to see if the frames of reference are correctly connected.
- Use of rqt_bag to record and play back data.
- Use of all the rqt tools to help you debug your system.
- Use of tf2_ros to look at transforms and make sure they are correct.

Behavior Trees—the big hammer.

Behavior Trees are a powerful way to define complex robot behavior. They allow you to define sequences of actions, conditions, and control flow in a way that is easy to understand and modify. When you first study Behavior Trees, they seem simple enough. You have sequence nodes, selector nodes, action nodes, condition nodes, and decorator nodes. You can build up complex behavior by combining these nodes in a tree structure.

A real robot, however, has to be built around reactive behavior. The robot has to be able to respond to changes in the environment, to failures of actions, and to new information from sensors. This requires a more complex use of Behavior Trees than just simple sequences and selectors. Figuring out a reactive tree behavior is hard.

A pattern I lean on a lot is “back-chaining”: start by asking “is the goal already true?” and only do work if the answer is no. In Behavior Trees that often looks like a ReactiveFallback where the first child is the success condition, and the second child is “the plan.” If the condition is true, the tree stops doing work. If not, it executes the plan.

The problem is that real robots are asynchronous and messy. You don’t just want a straight Sequence that remembers it already did steps 1–3; you also want it to re-check whether step 4 still makes sense given the latest sensor data. That’s where the difference between a normal Sequence and a reactive one becomes important: reactive nodes re-evaluate their assumptions continuously.

In practice, this meant wrestling with return statuses. Some of my “precondition” nodes couldn’t simply return SUCCESS/FAILURE; they had to return RUNNING to keep the sequence alive while still behaving reactively. I also needed “memory” in a few places so startup actions didn’t re-run forever.

My pragmatic hack was to put “need_to_*” flags on the blackboard. Each startup action checks its flag; if it’s already done, it returns SUCCESS immediately. When the action completes, it clears its flag. It’s not elegant, but it keeps the tree both reactive *and* sane.

Designing a complex Behavior Tree that is both reactive and remembers what has been done requires careful thought. That is the principal reason I decided to get this all to run under simulation first before trying to run it on the real robot. I have run the simulation dozens of times now to get it to be reasonably reliable. This would have meant picking up the real robot and setting it back home and recharging the battery many times if I had tried to do this on the real robot first.

There are a bunch of "gotchas" when using BehaviorTree.CPP that I ran into along the way. At least one node didn't work as advertised, at least in my mind, so I had to create a couple of my own custom reactive control and decorator nodes. I kept forgetting that subtrees do not share blackboard data unless specified with __shared_blackboard="true". I also created a custom SaySomething node that would print out the current pose of the robot and other useful debug information to help me understand what was going on.

While the current behavior tree works reasonably well in simulation, there is still a lot of work to be done. I've turned off the reactive safety tests for the moment. They test for, e.g.:

- Has the robot fallen over?
- Is the robot about to fall over?
- Is the battery critically low and the system needs to shut down?
- Is the battery low and the robot needs to return to the charging station?
- Has the safety system enabled emergency stop?

A lot of recovery behavior is not yet implemented. If any action fails, the robot just gives up. I need to add in retry logic and recovery behavior for all of the actions.

When it came to debugging the behavior tree, I got a lot of "Behavior Tree failed" messages. Well, those were certainly helpful (not!). Almost all of my development time was spent solving why the next node in the tree failed, fixing that, and waiting for the next "Behavior Tree failed" message.

Hardest to fix were irreproducible results. Differences in timing of asynchronous events or dealing with arbitrary configuration tolerances that varied a bit from run to run would cause the behavior tree to fail in different places at different times. I had to add a lot of logging to figure out what was going on.

Remember to deal with settling time and inherent movement errors in the design of the behavior tree. You have to make sure that when you check a condition, the robot has actually settled into the expected position. Otherwise, you may find that the robot is never able to satisfy the preconditions of the next action.

Groot—suddenly it's extremely costly.

The only reasonable way I found to visualize the behavior tree both during the design and during execution was to use Groot, the BehaviorTree.CPP visualizer. Groot is a great tool for visualizing Behavior Trees. The original Groot package is free and open source, but it doesn't support the latest BehaviorTree.CPP features. There is a commercial version of Groot that supports the latest features, but it is not free. It's not even close to free. You will need more than a paper route to use it.

I will eventually just write my own version of Groot and share it, but for now I'm using the free version and trying to work around its limitations. It's not especially painful, but I'd rather have the full features.

Compiling Groot from source also caused a problem in that OpenCV, needed for the object detection, has a dependency on a different version of NumPy than Groot. I had to create a version of OpenCV that used the same version of NumPy as Groot.

Tool creation for behavior tree debugging.

Debugging Behavior Trees is hard. There are a lot of little details that can go wrong. I found that I needed to create a custom node that would present a user interface where I could use dependency injection to simulate hardware failures. This allowed me to test the behavior of the robot when, for example, the robot was about to fall over or the battery was low.

I also wanted to be able to step through the behavior tree one tick at a time. This was harder than I thought it would be. The problem is that the Nav2 stack can take hundreds or thousands of ticks to navigate to a goal. I think I know how to solve this problem but I will wait until I have time to deal with it again.

Object detection.

The reason for the OAKD camera is that it supports a lot of AI features that I plan to depend on. Not only do I already use the point cloud from the stereo cameras for obstacle avoidance and navigation, but I plan to build a semantic segmentation of the environment so that the robot can understand what objects are in the environment and where they are located. This will allow me, say, to avoid getting the wheels tangled up in power cords on the floor, or perhaps navigate around a floor mat instead of over it. Of course, I'll use it for object detection as well.

Sigyn is intended to be a personal assistant robot, so it will need to be able to recognize people and objects in the environment. The OAKD camera is a good choice for this because it can run AI models on board, reducing the need for a powerful onboard computer. And I can train those AI models for all of the objects of interest in my home environment.

The Pi camera is to be used primarily to add in close-up vision for grasping objects. The Pi camera is mounted above and slightly behind the gripper and is tilted downward to see within the frame of the gripper, so it can provide a close-up view of the object being grasped. This will allow me to fine tune the grasping behavior and ensure that the robot can successfully pick up objects.

For now, I used AI to create a naive object detector using the same logic I've used in the past. The object detector looks for color blobs in the image that match the expected color of the soda can. It then computes the bounding box of the largest blob and returns that as the detected object. There was a bit of tuning I had to go through to lower the rate of false positives and false negatives, but it works reasonably well in simulation.

It's a placeholder for a more sophisticated object detector that I will implement later. The naive object detector works reasonably well in simulation, but it will need to be improved for use on the real robot.

The implementation, however, was a chore to get right in the ROS and behavior tree framework. There were a lot of little details to get right, so that I could ensure that an actual new frame came in and was processed after the robot moved. Several of the behavior tree nodes had to have a precondition that ensured that a new frame had been processed since the last time the node ran. Otherwise, the robot would just keep using the same old image and never see the updated position of the can.

Creating the simulation world.

Attempting to create a Gazebo world using the Gazebo GUI is a nightmare. The GUI is clunky and slow and you end up having to do long operations over and over again. Placing objects in the world is tedious and error prone. I found that it was much easier to create the world using SDF files directly.

So I studied the SDF files created by Gazebo in some experimental worlds and learned how to create my own SDF files. I then took the blueprints of my house and created a little language of my own to describe all of the walls, doors, windows, furniture, and other objects in the house. I then used the M4 macro processor to generate the SDF files from my blueprints. All I had to do then was to create a soda can and place it on the table in the kitchen.

Simulating the real hardware.

Getting the ROS simulation to accurately reflect reality required a lot of specific tweaking. It wasn't enough to just drop in a model; I had to implement simulations for specific hardware oddities and missing Gazebo features:

- **Hardware fidelity:** I had to account for optical stops for the elevator and stepper motors for both the elevator and extender.
- **Physics tuning:** I needed proper physics parameters for all objects and accurate gripper finger collision models.
- **Visuals and State:** Simulating joint states was necessary so Rviz and Gazebo would visualize the robot correctly. I also had to simulate the TF frames for stepper motors and other moving parts.
- **Camera setup:** The Pi camera position and tilt had to be meticulously calibrated to get the proper image for the grasping algorithms. I also had to deal with getting image data in its own x,y,z optical frame of reference, which is different from the standard ROS frame of reference.
- **Simulation gaps:** There is no 'mimic' joint support in Gazebo (unlike URDF/Rviz), so I had to redesign the URDF to not use mimic joints.
- **Software Architecture:** I crafted the hardware abstraction layer to use dependency injection. This allowed me to "mock" the hardware interfaces in simulation so that the exact same Behavior Tree nodes run on both the simulator and the real hardware without modification.

But wait, does it work on a Macintosh?

I need the Macintosh support because, well, frankly, I was spending 16 hours a day (more or less) working on this project and, in the evening, I needed a comfortable environment to work in. That environment is my living room, in a comfy chair, with my MacBook Pro on my lap and reruns of TaskMaster playing on the television in the background. My robot and my main ROS workstation both use AMD processors which play nicely with ROS out of the box, but I also need ROS to run on Apple Silicon chips on the MacBook Pro.

The ROS community has a long history of not supporting Macintosh very well which is pretty damn irritating. I have spent a huge amount of time and effort to get ROS and Gazebo to work on a Macintosh. Even when that worked, I found that the LIDAR simulators in Gazebo had problems on the Macintosh. I spent more than a day just solving that problem.

Articles and papers need an end.

So here is the end. I'm pretty damn pleased with myself to get so much stuff done in a week. My throat is a bit sore from all the (actual) shouting I did at the AIs. Some are incredibly resistant to listening to my instructions. One I always avoid; oh, I test it now and then, but so far it is an outlier in its stupidity.

Sigyn is not a toy. A robot is pretty much useless to me if it isn't trustworthy and safe. There's a lot of work yet to get there, but this is a big step forward.

Have I mentioned that "Everything about robots is hard" (TM)?