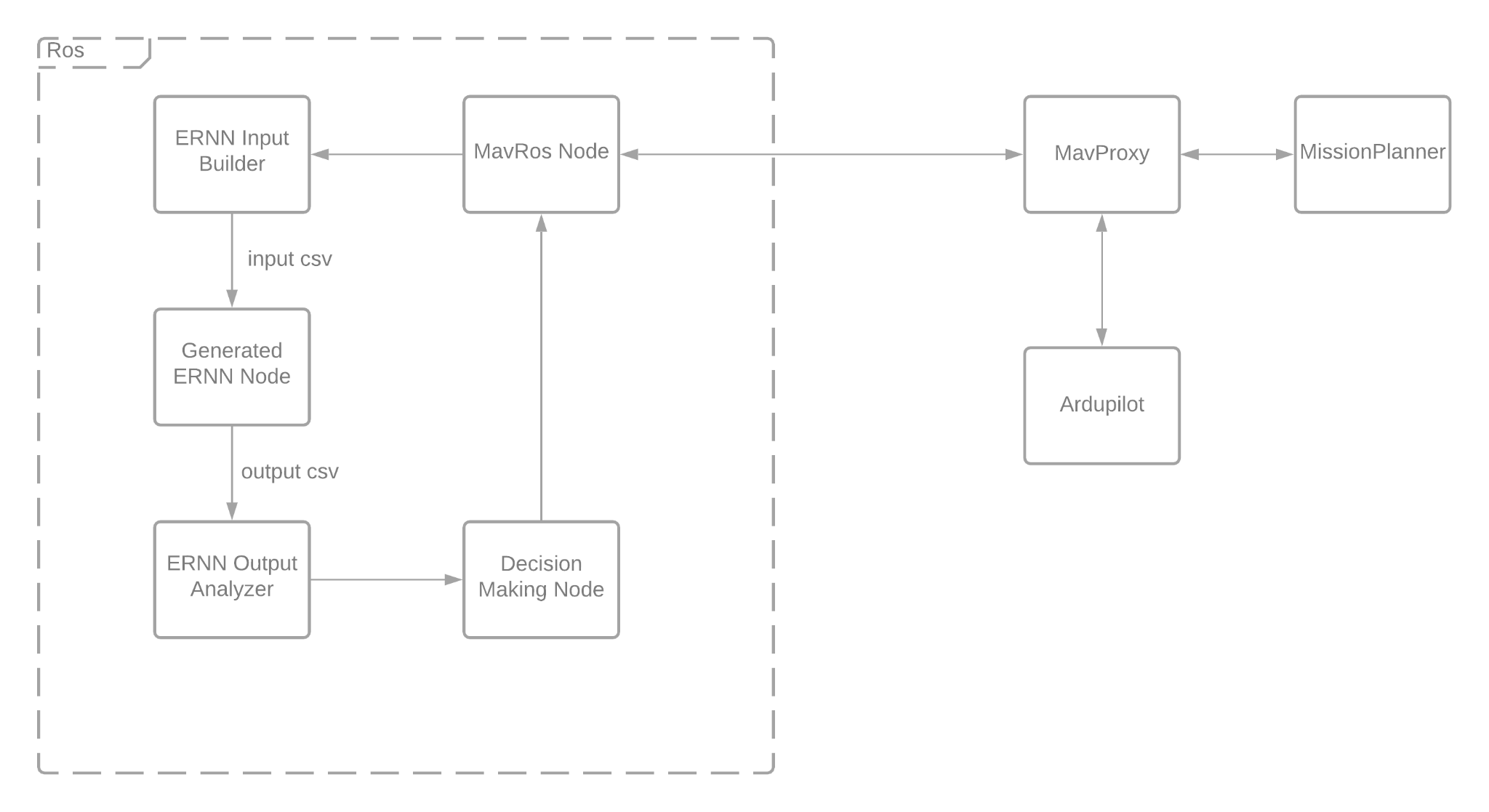
# **Vehicle AI - Tools Paper**

### **AItomotives**

|  |  |  |
| --- | --- | --- |
| Brandon Goodhue | Hunter Kelley | Christopher Salvati |
| Stephen Cosco | Trevor Lezynski |  |

### **System Design**



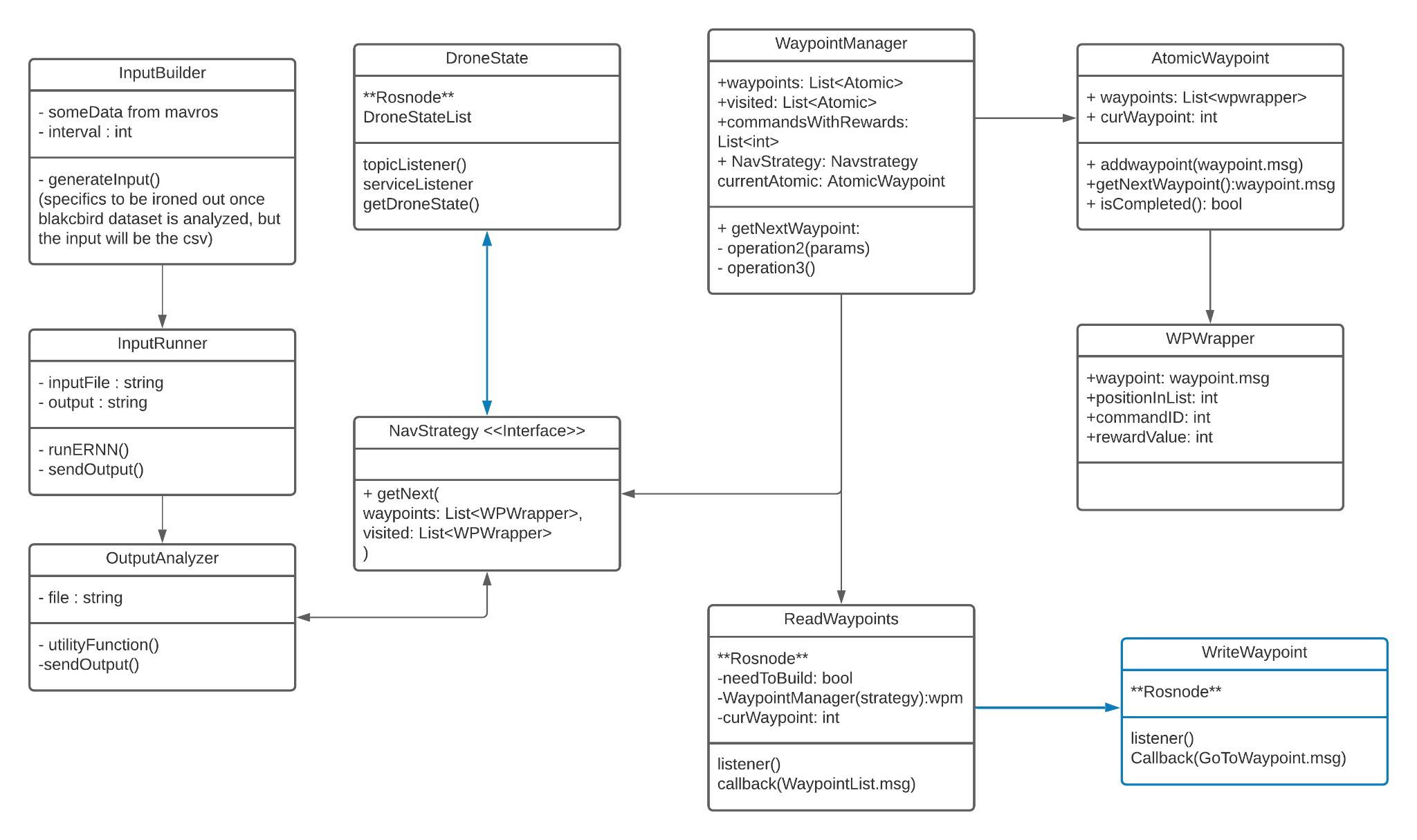
**Mission Planner -** A ground control software that is able to to create missions for the drone as well as interact with some drone parameters. Their parameters include things that allow for variance including wind speed and direction and battery percentage. This software is like a front end for the user to see what the simulated drone is doing and information about it in real time. Our system uses this as the place to feed a drone test scenarios and is also where we have a script to edit these parameters.

**Ardupilot -** An open source drone software that handles a lot of the actual drone piloting and simulating. Everything but the brain of the operation. Center point of all communication between the tools.

**Mavlink -** This is the communication protocol that drone commands and messages are sent with. This is the method through which all of our different tools and components are able to communicate with each other. We had to modify this protocol to be able to send and receive different types of messages, as well as modify other parts of our system to utilize the customizations we added.

**ROS -** Robot operating system is a node based publisher/subscriber framework. We are able to have multiple nodes or programs running that can communicate with each other in a pubsub fashion and through a library communicate with ardupilot. This communication goes both ways as our nodes read things such as the drones currently programmed mission and current state, and it can write back to the drone what decisions it should be making. Within our ROS system is where most of our effort was directed and it's the majority of our system.

Since this was the majority of our system this is also where most of our design decisions were made. Within the ROS architecture there are three ROS nodes of our own creation, each one having its own purpose, and then a library called mavros. This library has multiple nodes and services that act similarly to an api for ardupilot. Because of the pub/sub nature of ROS architecture the team’s nodes are able to subscribe to the nodes of the library and receive data from the drone. The team’s 3 nodes have different purposes and interact with different parts of mavros.

****

One of the nodes is responsible for writing to the drone a waypoint that it should be going to next. The team decided that this should be a separate node so that it has the potential to subscribe to multiple different sources. This node being separate also gives it the single responsibility of having to talk to mavros, without being coupled to any logic.

Another node is responsible for reading and remembering all of the drone’s state. This node has slightly more responsibilities as it reads from multiple different mavros nodes and compiles information. Since the neural network needs data from more than just one instant, this node also keeps track of a number of the most recent readings from the drone.

Not every mavros node updates at the same rate. For instance the position of the drone is updated 30 times every second, while the drone’s battery status is updated twice per second. Since this node is tasked with reading from multiple nodes and compiling them together, it works on its own time. Within the node there is a dictionary that is constantly updated with the most recently published data on all of the mavros topics. On a rate that can be changed, the node makes a copy of the dictionary and adds it to a list of all the recorded readings. Since the neural net needs more than one instant of data to be able to make predictions, this list of dictionaries is a good representation within python of time series data. The list also has a variable maximum size before it starts to remove the oldest entries and replace them with newer readings. Testing the neural net showed that anything less than around 90 instances of data wouldn’t allow it to make a prediction, so for most of the testing the 200 most recent readings at 0.25 second intervals were used.

In terms of design this functionality was decided on being its own node due to the amount of communication with mavros that was necessary, and the specific purpose it had. While this node was responsible for both communication and logic unlike the waypoint writing node, the functionality within this node is very specific to the information being read. This led to an overall very cohesive piece that didn't need to be abstracted or broken apart further.

The final node is the driver node. It is responsible for reading all of the mission commands or waypoints from the drone and keeping an internal representation of them. When starting a mission the node creates a new waypoint manager class that takes the mavros mission data, and pulls out specific important pieces to build custom waypoint objects. These can be managed, tracked, and grouped within our system, which solves the exact problem that we initially had with ardupilot, and the reason we switched to ros. This node reads the mission every time a “decision point” is reached. For this project it was decided that decision points were every time the drone reaches a waypoint, but in the future it would be possible to have a decision point be based on time or a number of other things. Since this node has access to the manager with all of the waypoint data, this node is also responsible for passing that data to where a decision is made. There is a strategy pattern implemented so any python class with a “getNext” function can be used to decide where the drone should go to after reaching a waypoint. Then once a decision is made this node only has to communicate with the first mentioned ROS node about what was decided, and communication back to the simulation is handled by that node.

The strategy pattern that is used to make decisions isn't a perfect strategy pattern for a notable reason. Python is not a typed language so it is up to the user to use a correct type for everything. The system is architected so that it works well with an injected dependency of a strategy class that follows the paradigm, but in theory someone could break this as python has no strict requirement of what classes are used.

There exists in the codebase a few different strategies for making decisions for the drone. These currently include a random strategy that navigates the drone randomly, a smart strategy that navigates the drone to the waypoints with the highest rewards, not taking cost into account, and a utility strategy. The utility strategy is currently in an unworking state due to it not calculating cost correctly, but there are still a lot of working components to the strategy. The utility strategy is able to query the ernn and get the predictions to use in cost calculation. It attempts to calculate the distance to other points to use cost, but currently the distance calculation isn't correct. However there are safeguards that would work if the calculation was correct, such as having an infinite cost if the drone to go somewhere too far that would leave it unable to return to home. One final thing the utility strategy is accounting for is lookahead. When calculating cost and reward its important to look not just at the current location, but where the drone could go after that. For example the drone could go to one really far waypoint worth 5 points, or it could go to a group of three close waypoints worth two each. The utility function takes this into account by being able to look at waypoint costs recursively, and decides where the drone should go currently by looking at what could be the best path in the future. Lookahead calculations start to get very costly so the sponsor recommended not to look anything farther than three points into the future. Changing the amount of points looked ahead is changeable by a single variable, but currently the lookahead depth being used is only two.

### **Product State at Time of Delivery**

At the time of delivery, the state of the project is significantly different from the product we intended at the outset. Instead of having a system capable of making on the fly path planning, we've built the tool that can be used to accomplish this. This is to say, our system has all the components to *make* path planning decisions autonomously, but it doesn't actually do this.

Due to the issues in establishing and utilizing a suitable dataset for the development of this project, we were never able to adequately train the eRNN to make actually useful predictions. Because the system cannot make those predictions, it also means we've failed to create a suitable utility function to be used in the decision making process. This is not to say what we've constructed is useless though.

If provided with an adequate dataset and utility function, the project should work as intended. All that's actually required to make decisions are those two things, and providing them would be relatively simple due to the architure we choose for implementing decision making. The way the drone decides to visit different waypoints is handled by an implementation of the Strategy pattern, so making modifications to the existing code base is as simple as it possibly could be.

**Setup instructions**

*Note:* These setup instructions assume you’re running on a Windows machine, and require you to have Ubuntu 18.04 Windows Subsystem for Linux 2, which adds additional capabilities (such as Docker) over base WSL.

* Clone [AItomotives/Ardupilot](https://github.com/AItomotives/Ardupilot) repository
* Clone [AItomotive/ros](https://github.com/AItomotives/ros) repository
* Clone the [EXACT](https://github.com/travisdesell/exact) repository
* Install [MissionPlanner](https://ardupilot.org/planner/docs/mission-planner-installation.html)
* Enable [WSL2](https://docs.microsoft.com/en-us/windows/wsl/install-win10) on your computer. Make sure you follow the steps to enable WSL2.
* Install [Docker](https://docs.docker.com/docker-for-windows/install/) for Windows.

**Run the docker container**

1. cd to the ardupilot directory
2. Build the docker container with:

|  |
| --- |
| docker build . -t ardupilot |

If this is the first time, this may take a while, as the docker container will be cloning and downloading Ros, which is a few gigabytes, so you can estimate based on your internet speed.

1. Run this command. Replace {path\_to\_exact} with the location of your EXACT clone and {path\_to\_ros} with the location of your ROS clone.

|  |
| --- |
| docker run --rm -it -v {path\_to\_exact}/exact/build/rnn\_examples:/exact -v {path\_to\_ros}/ros:/catkin\_ws -v `pwd`:/ardupilot ardupilot:latest bash |

1. Your docker container should now be running. You will have to have more terminals connected to this container, so if you’re comfortable using things like tmux, you can do that. Alternatively, you can open a new terminal and connect to the container. To do this, get the name of the container from the Docker gui or the command docker container ls. Then use the command docker exec -it {container name} /bin/bash to enter the container.

**Start Ardupilot**

1. In a non-docker terminal run ip config to find the ip address of the currently running WSL instance.
2. Start MissionPlanner
3. In your docker instance cd ArduCopter
4. ./run.sh {IP\_Address} with your WSL ip address. Mission Planner should automatically connect but if it doesn’t then connect to your MissionPlanner instance with the connect button in the top right. Feel free to go into the script and just put in your IP address as long as you don't commit it, especially if you're running this multiple times. Otherwise, I recommend you keep it on hand. Ardupilot should continue

**Start ROS**

1. Open a new terminal and connect to the Docker instance
2. cd /catkin\_ws and ./roscore\_build.sh. For some reason this fails the first time it's run, run it again. This was happening for a little bit, but shouldn't. If it keeps failing, there may be additional issues.
3. Open a new terminal and connect to the Docker instance
4. cd /catkin\_ws and ./rosrun\_node.sh.

**Starting a Mission Planner mission**

1. With MissionPlanner and Ardupilot running, the simulated drone should automatically connect.
2. In the Plan tab on MissionPlanner, either create your own mission with waypoints, or run an existing waypoints file by Loading one. Write the waypoints when you're finishing editing them. Reward commands need to be added manually by changing the command type to UNKNOWN, entering the command ID of 26, and setting the first field to the value of the reward. You can also set the coordinates and altitude of the reward waypoint in the last three boxes of the command to exactly the same as the waypoint command to make the visuals more appealing.
3. On the Actions tab, make sure Mission Start is selected in the top left drop down box, select Arm/Disarm to arm the drone, then select Do Action to begin the mission.