Autonomy Subteam Research and Development for Fish 'n Ships

Associated Members of the Autonomy Subteam and other contributors

Abstract — This report details the advanced control strategies and research operations undertaken by the Arcturus Autonomy Subteam in implementing the software that powers Fish 'n Ships. We describe both the theoretical underpinnings of our work, its technical significance and general implications, and how each concept or idea was implemented and used in Fish 'n Ships. The scope of our work is broad–encompassing both technical performance improvements specific to the ROS architecture and more theoretical results developed as part of our research and development operations. In full, the result of our efforts has been to enable completely autonomous and efficient localization, mapping, navigation, and error-correction for a small boat based on intelligent perception.

Associated Members of the Autonomy Subteam and other contributors
Massachusetts Institute of Technology (MIT), 77 Massachusetts Ave, Cambridge, MA, email: arcturus-logistics@mit.edu

1 Software Deployment Model

This year we have investigated alternatives to our longstanding software deployment model, which has traditionally required the careful installation of certain system dependencies in addition to prehistoric technology like CMake to build various modules required for ROS nodes to work. This system suffered from numerous flaws, leading to an inevitable spiral towards what is known in the relevant literature as "dependency hell" [1] (Fig. 1.5).

1.1 Motivations

The main problem that the patchwork dependency installation scheme created was that of non-reproducible environments. It was practically impossible to construct two identical computing environments capable of running the ROS nodes. Additionally, the only way to natively run much of the Arcturus codebase required installing a specific version of Ubuntu, effectively mirroring the environment setup of the computer the nodes would eventually be run on. In addition to system dependencies required to run the ROS nodes, testing required the installation of unrelated visualization components, described by some as "pure graphical bloat," which often cluttered development environments.

Lastly, dependency specifications looked a lot like JavaScript code (metaphorically, not syntactically–they were in XML format): incomplete, non-standardized, and difficult to follow. This lead to manual installation of system dependencies (almost always unspecified) and a collection of random hacks to install remaining packages. This is the textbook definition of "technical debt," or what is now being recognized as "maintenance load" [2].

In this section, we outline how to apply a purely functional software deployment model (or a *monadic* one [3]) to a set of ROS modules intended to be run in production on a Jetson Nano. We note that this experiment builds on previous work in this space, most notably [4] and [5]. We begin by outlining some of the immediate benefits of a purely functional software deployment model in the context of the ROS ecosystem, and then move onto some of the practical challenges involved in implementing a pure build system, what progress has been made so far, and what future work remains to be done.

We believe that the experiments we have conducted so far lay the groundwork for a completely automated software deployment, management, and testing environment that can be run at scale. We also recognize much of the same has probably been said about the Java Virtual Machine, but there is strong evidence to suggest this will be different (at the very least, Nix is a product of academia and Java that of corporate tomfoolery).

1.2 Background and Related Work

A purely functional software deployment model is one that leverages the principles of purely functional software design (think of Haskell, Lean, etc.) to creating package build scripts. If this sounds like a bad idea to you already, you now understand why it has little uptake outside a niche circle of leading academics and ardent Nix theoreticians (often referred to as "flake managers"). The central idea of this model is that every package will be installed using a build function that is "pure" in the mathematical sense, by which we mean that it produces an output determined uniquely by its inputs. Running this function multiple times with the same input should yield identical outputs.

The inputs of such a function are the packages it depends on, and its output is a list of configuration settings and metadata associated with the package. Purely functional build tools can use this metadata to identify the exact version of the package to install and create an environment containing that package with those specific configuration settings applied.

The most notable purely functional build tool in use at the moment is Nix, originally proposed by Eelco Dolstra in his seminal 1978 thesis [1]. While the core idea behind purely functional software deployment–the composition of pure builder functions to create complex yet reproducible computing environments–is simple, its benefits are more far-reaching and powerful than they might seem at first glance.

We use the Nix packaging ecosystem, `nixpkgs`, a repository containing a set of pure builder functions (called "derivations"), for a plethora of common system dependencies, along with the Nix build tool, as the basis for our experiment. However, `nixpkgs` does not contain the derivations for ROS packages, so we must rely on another packaging repository to fill in this gap. We find [4] to be the most reliable provider of these packages and incorporate it into our environment setup.

1.3 Potential Benefits

While [4] does not provide a binary cache for the Humble distribution, it is not difficult to set up a Hydra build server that can distribute these binaries for the packages we need. We are currently working on setting up such a server (using NixOS, of course), and we suspect this will greatly speed up package builds, since it will be faster to use Nix's binary cache ecosystem over a local CMake build. Indeed, reports from conferences of Nix academics confirms our prediction [5], suggesting that we can speed up build times approximately five-fold and reduce resource consumption by about 90%. Aside from the obvious practical benefits of allowing faster testing iteration, this has the added benefit of being an overt display of Nix superiority over other build systems.

Of course, this is just one of the many benefits that we obtain by using a purely functional build environment. Aside from enhanced reproducibility, we gain clear and understandable dependency specifications, making it easier to locate and patch security vulnerabilities as well as enabling us to update packages without fear of unforeseen breakages later down the line thanks to instant rollbacks. We are able to create a unified specification of both system and ROS dependencies for an individual project, and we can also use such a specification to create automated runners for testing and debugging purposes. As should be quite obvious now, the ramifications of creating such a reproducible build system are large, and they can lead to an enormous reduction of maintenance loads and an enormous increase in time spent writing Nix expressions instead of writing actual code.

1.3.1 Build Speedups

The clear specification of all dependencies also allows Nix to perform "true delta" builds [5], where Nix can identify the exact subset of packages that need to be rebuilt or replaced and apply those changes without performing unnecessary work (e.g. rebuilding dependencies of updated packages that stay the same or secretly replacing your operating system with NixOS). Nix eliminates nonsensical bikeshedding [6] over ROS versions and distributions, unifying package declarations in a single monorepo that can be used as an input to a derivation for the project environment. Compared to Docker, the industry standard for reproducible builds that has been the standard build system for Arcturus on non-Ubuntu systems,

Nix offers the huge benefit (in addition to those previously described) of being able to identify and utilize system hardware resources and run code natively without virtualization overhead.

At this point, we have covered the principal advantages of Nix over currently available build tooling. Nix gives you **velocity** (faster builds, true delta comparisons, binary caches), **reproducibility** (identical computation across architectures and machines), **interpretability** (explicit and verifiable declaration of all dependencies and seamless upgrades), and **interoperability** (bundling of system and ROS-specific tooling into a single, contained development environment). There are two final benefits that are so essential to any build tool that they almost should not be mentioned; yet they are so obviously lacking from the current build process that they are worth mentioning here.

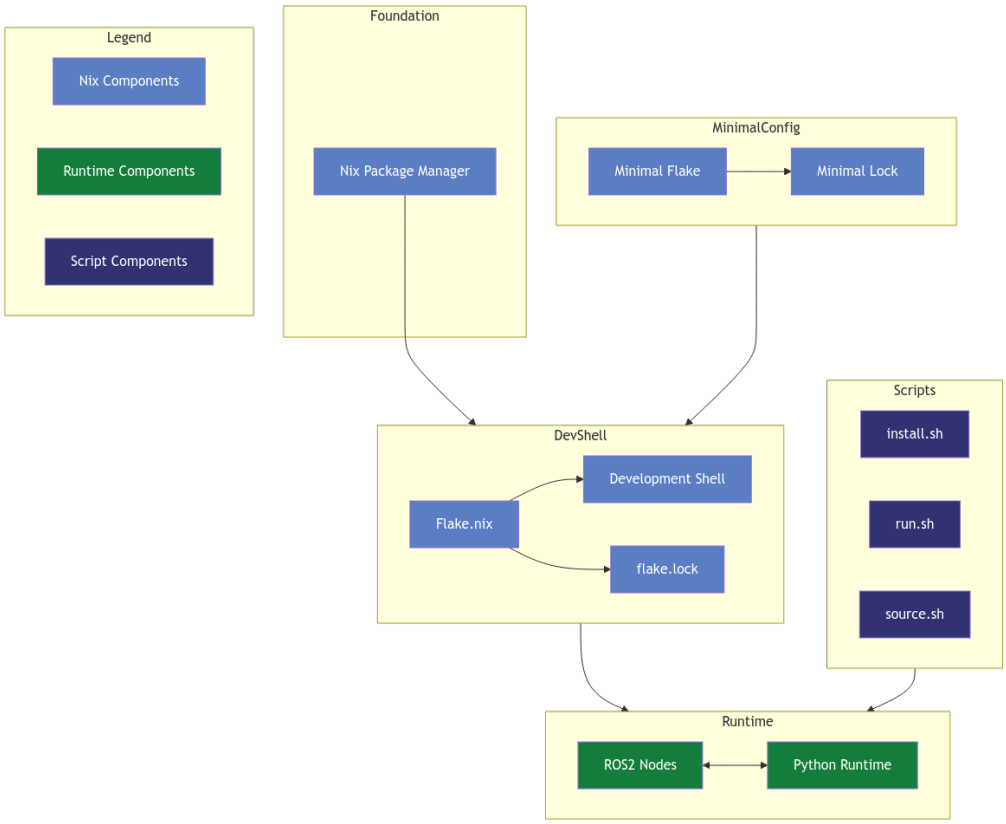
1.3.2 Additional Benefits

The first is that Nix can support *multiple* reproducible development environments. Attempting to reproduce the entire setup of the Jetson Nano onto an Ubuntu installation will result in an imperfect approximation of the boat's behavior, but, even more alarmingly, it will prevent experimentation with other packages and development of code that requires a different computing environment (some may call it a "walled garden," but this is a blatant mischaracterization of a truly sad situation). The only solution to this is to use a containerized environment like Docker, which does allow multiple reproducible development environments to be set up, but with high virtualization costs for each one.

The second benefit that even Docker cannot offer is that of composability. In Nix, it is trivial to use the output of a special type of derivation, called a flake, as the input to another flake, effectively building on top of an existing set of packages–either adding on additional packages or even modifying the environment defined in the upstream flake. An example of composition is evident even in the setup of the Nix-based build system for Arcturus–ROS packages are not listed in `nixpkgs`, but we can use an *overlay*, a special type of composition, that injects these packages into the `nixpkgs` derivation and then consume this as an input to the flake we use for setting up the ROS development environment.

Docker's containerized architecture makes these types of modifications practically impossible–you can run multiple containerized environments in parallel, but never at once [7]. Such is the travesty of "modular" containers. There simply isn't a good way to modify or extend an existing Docker environment, since Docker has no way of meaningfully tracking the packages installed or configuration settings applied in any one container.

1.4 Implementation Details



Automatically generated (read "we passed this into ChatGPT and this is what it came up with") theoretical diagram of current Nix build infrastructure

1.4.1 Flake Inputs

Implementation began by using a template *flake* provided by `nix-ros-overlay` [4]. Flakes allow for complete reproducibility in dependency specifications, since all inputs must be explicitly declared and hashes are stored in a corresponding lockfile. While [4] provides the ROS packages needed to run the `all_seaing_vehicle` Arcturus codebase [8], we still need to convert ROS dependency specifications (in XML format) into Nix package derivations.

We use `ros2nix` [9] to automate the task of converting the haphazard mess of ROS package files into meaningful Nix expressions. We do this in a way that leaves the original repository untouched (i.e. we can generate Nix package files *on demand*) because we are nice to Docker and Ubuntu adherents. For build tools and other dependencies, we import the `nixpkgs` package registry, which `nix-ros-overlay` extends. In full, the `inputs` section of the flake used to build all Arcturus code is shown below:

```
inputs = {
  ros2nix.url = "github:wentasah/ros2nix";

  # important for ros overlay to work
  nixpkgs.follows = "nix-ros-overlay/nixpkgs";
  nix-ros-overlay.url = "github:arcturusnavigation/...";
};
```

We use a specific commit of a forked version of the Nix ROS overlay to fix dependency versions in place. We can always perform automatic updates by syncing our fork with upstream and updating the flake. This fork has also allowed us to incorporate patches of critical build errors in packages that we depend on in the case the default derivations do not work as expected.

1.4.2 Development Shell Architecture

As output, we create a Nix development shell for each system type. This is of course a suboptimal architecture, since development shells are not perfectly reproducible, but they provide all the tools necessary to reproducibly build dependencies and the codebase itself. In the process of creating the development shell, we discovered numerous packages that were unspecified as dependencies and thereby created the first working dependency specification for the all_seaing_vehicle codebase, a historic achievement after a semester's worth of tumult and obloquy.

We also discovered a surprising number of system dependencies that the codebase relies upon, but which are not explicitly listed in any dependency manifests, a classic feature of most ROS codebases. Additionally, many of these packages were Python packages, which on any other system would have to be installed through a Python `venv` or similar for reproducible results. However, Nix packages these in the same way as any other application, and it is trivial to import them from `nixpkgs` and patch remaining build issues. See below for a sampling of the main flake responsible for setting up the development environment for `all_seaing_vehicle`:

```
devShells.default = pkgs.mkShell {
  name = "arcturus";
  LOCALE_ARCHIVE = "${pkgs.glibcLocales}/lib/locale/locale-archive";
  packages = [
    # basic development tasks
    pkgs.git
    pkgs.bash

    # ros build tools
    pkgs.colcon
    ros2nix.packages.${pkgs.system}.ros2nix

    # project system dependencies
    pkgs.geographiclib
    pkgs.python3Packages.rosdep
    pkgs.python3Packages.scipy
    pkgs.python3Packages.transforms3d
    pkgs.python3Packages.torch
    pkgs.python3Packages.pyserial
    pkgs.SDL
    pkgs.yaml-cpp

    # other system packages here
    (with pkgs.rosPackages.humble;
    buildEnv {
      paths = [
        # empirically determined list of project ros dependencies
        ros-core
        boost
        ament-cmake-core
        ...

        # for launch
        mavros
        robot-localization
        velodyne-pointcloud

        # other ros packages
      ]; } )
  ]; };
```

A simple `devshell` setup for managing Arcturus ROS codebases

Combining this with a series of setup functions, we can create the same development environment on any system (though the user must be enlightened enough to have Nix installed). In order for the `all_seaing_vehicle` codebase to be identified by ROS, however, each subpackage must be added to the system path along with a handful of utilities. A reproducible installer (which clones the `all_seaing_vehicle` repository into an empty directory and sets up Nix manifests on top of the existing ROS infrastructure), code discovery tool (to make the codebase discoverable by ROS tools), and runner (a wrapper around `ros2 run`) allow a practically identical setup across machines.

1.4.3 Related Concerns

The one major issue is that using a collection of shell scripts in a development shell is not verifiably reproducible, so mathematicians and other theoreticians may be unable to use it for moral reasons. However, this can be trivially solved by bundling these scripts as Nix *packages*, which can be reproducibly installed and executed in the development shell. Another concern is that `nix-ros-overlay` does not provide binaries for any distribution other than Noetic, so realizing the full speed gains made possible by Nix infrastructure will require us to set up our own binary cache. For plans related to this and other projects, see Sect. 1.5.

The full source for the Nix-based setup of the `all_seaing_vehicle` [8] codebase can be found here [10].

1.5 Future Work

We begin with a brief assessment of current plans for the future and then elaborate on the vision for the deployment of a fully reproducible environment for building modular codebases for the robot operating system. If that sounds like a container, it is absolutely *not* a container. It is a collection of pure functions implementing a monadic build procedure to create reproducible and native packages that act deterministically on an environment created by a meticulous setup process (absolutely no similarity to a container, whatsoever).

As of now, most work will be concentrated on converting ad-hoc shell scripts into proper Nix packages with appropriate derivations integrated into the main development flake. A secondary goal that can be pursued in parallel will be to get Gazebo/VRX simulation packages ("graphical bloat") working and integrated with the rest of the Nix development environment. Additionally, all binaries needed to compile the flake should be cached and regenerated with every update by an on-premises Hydra server.

Finally, the Nix environment should be used to set up robust continuous integration (CI) pipelines for testing robot code, so even a single-line diff can be subject to intensive automated experimental verification that will delay integration into upstream and artificially lengthen the review process. This should also include meta CI for managing the copious Nix expressions used throughout this project and updating flakes, upstream dependencies, etc. A fully realized Nix development environment may also include editor configuration and other developer tooling, akin to [11]. Since developers may end up writing more Nix expressions than any other language, some Nix tooling will likely be a vital addition.

1.5.1 Packaging Infrastructure

While it is common practice to run arbitrary shell scripts inside development shells, this is far from an ideal setup. Development shells are typically intended for rapid experimentation where this type of behavior may be more suitable, but for robust testing (e.g. in CI and even for local development) it will be necessary to ensure verifiable reproducibility. Shell scripts can potentially rely on packages that already exist on a specific system and utilize those configurations, making it impossible to determine whether the dependencies for a shell script have been appropriately included in the development shell and whether those specific versions are the ones being used. This can be easily solved by packaging each of the scripts and running the package (which will not even require using a development shell) to install

the `all_seaing_vehicle` repository, build subpackages, and run nodes. We can then point rigor-hungry mathematicians and other theoreticians to [1] for further research.

Launch files are also currently run from a development shell, but ideally we would like to automatically create binaries that run the launch file using `nix build` or similar. This will likely be worked on concurrently with the change to the new packaging infrastructure. We should also explore a complete Nix environment setup that directly builds entire ROS distributions, such as [12] presented by Clearpath Robotics. `nix-ros-overlay` expressions could be used as a standalone input instead of importing the entire repository as an overlay on top of `nixpkgs` and a Nix-first build infrastructure can be developed. This is an active area of research, so we expect a separate whitepaper to be published that covers the theoretical groundwork for such a system after it has been properly formulated.

1.5.2 Simulation and Tooling

One of the most requested features after the informal proposal of the Nix build system to the Arcturus Autonomy subteam has been the addition of support for Gazebo simulation, as–for better of for worse–it remains a critical part of how code is tested before being deployed. There does not exist any pre-existing Nix package for Gazebo due to the Ubuntu lobby, so this will likely need to be worked on before checking that the Gazebo package can interface with ROS nodes in the Nix environment.

Aside from simulation support, developer tooling is extremely important. As the Arcturus codebase evolves, it will greatly improve quality of life to bundle useful tools into a secondary environment that will not be built in production. To some extent, some useful packages that are not strictly necessary for the code to build are included in the `arcturus-nix` flake, but these should be separated from actual dependencies before adding more to avoid accumulating bloat and angering Helix users.

1.5.3 Hydra and Continuous Integration

Due to the lack of an existing binary cache for the Humble distribution in `nix-ros-overlay`, we will need to provide an on-premises binary cache of the packages used in the Arcturus codebase. Setting up a Hydra server and uploading build artifacts from a job on that server to a Cachix instance seems to be the most reliable way to do this, but Cachix is an external tool and running a local binary cache may be better. For now, however, Cachix seems to be the easiest solution to integrate for flake consumers, so that leaves the automatic publishing of build artifacts to Cachix as the only task to be completed. A basic Nix-based PulsarOS server configuration has already been created for this purpose.

Continuous integration can be run anywhere with identical results thanks to the Nix environment, but it will likely be more convenient to use the on-premises Hydra server for this purpose as well. However, this will require adding additional infrastructure and developer tooling in the Nix environment for running and interacting with tests.

1.6 Conclusion

We end by examining how a simple change in dependency management can lead to such a remarkable set of results. By merely leveraging pre-existing infrastructure, we are able to build on decades of research and technical innovation to increase developer productivity, eliminate non-reproducibility and related bugs, and usher in the age of distributed computing. The vision presented in this report outlines a set

of simple next steps that will allow for instant developer environments to be spun up on any machine in a matter of seconds due to an automated build server storing binary caches of the exact packages needed to build the latest version of the codebase (*not* a container!). Reproducible testing and simulation broaden access to essential tooling and make it easy to find bugs hiding in plain sight. The problem of "dependency hell" [2] and the infamous "it works on my machine" defense will cease to exist as we know them, replaced with a great deal of hysteria surrounding whether to switch to NixOS and many hours wasted writing and re-writing Nix expressions to configure fully reproducible nonsense. The end result: Fish 'n Ships is no longer a fully autonomous, omniscient boat–it's a fully *reproducible*, autonomous, omniscient boat.

References

1. Dolstra, E.: The Purely Functional Software Deployment Model, (1978)
2. Troy, C.: Stop saying "technical debt", https://stackoverflow.blog/2023/12/27/stop-saying-technical-debt/
3. jade: The postmodern build system, https://jade.fyi/blog/the-postmodern-build-system/
4. Wolsieffer, B.: lopsided98/nix-ros-overlay, https://github.com/lopsided98/nix-ros-overlay
5. Purvis, M., Wanders, I.: Better ROS Builds with Nix, http://download.ros.org/downloads/roscon/2022/Better%20ROS%20Builds%20with%20Nix.pdf
6. jerf: https://news.ycombinator.com/item?id=40701501
7. https://docs.docker.com/compose/
8. Arcturus Autonomy: ArcturusNavigation/all_seaing_vehicle, https://github.com/ArcturusNavigation/all_seaing_vehicle
9. Sojka, M.: wentasah/ros2nix, https://github.com/wentasah/ros2nix
10. Arcturus Autonomy: ArcturusNavigation/arcturus_nix, https://github.com/ArcturusNavigation/arcturus_nix
11. Ratnakumar, S.: srid/haskell-template, https://github.com/srid/haskell-template
12. Clearpath Robotics: clearpathrobotics/nix-ros-base, https://github.com/clearpathrobotics/nix-ros-base