Packaging Medical Imaging Software with Nix

https://github.com/ryanorendorff/medical-imaging-nix

Ryan Orendorff

December 12th, 2019

Overview of the talk

- Introduction to Magnetic Particle Imaging (MPI).
- How to set up systems with tons of dependencies like MPI.
- What worked well, what did not work as well.

Disclaimer: The Nix work was done at Magnetic Insight, my employer.

Magnetic Particle Imaging: Detecting Iron Nanoparticles using Magnetic Fields

MPI detects iron in the blood

Magnetic Particle Imaging (MPI) is an emerging medical imaging technology that images iron particle distribution in a body.

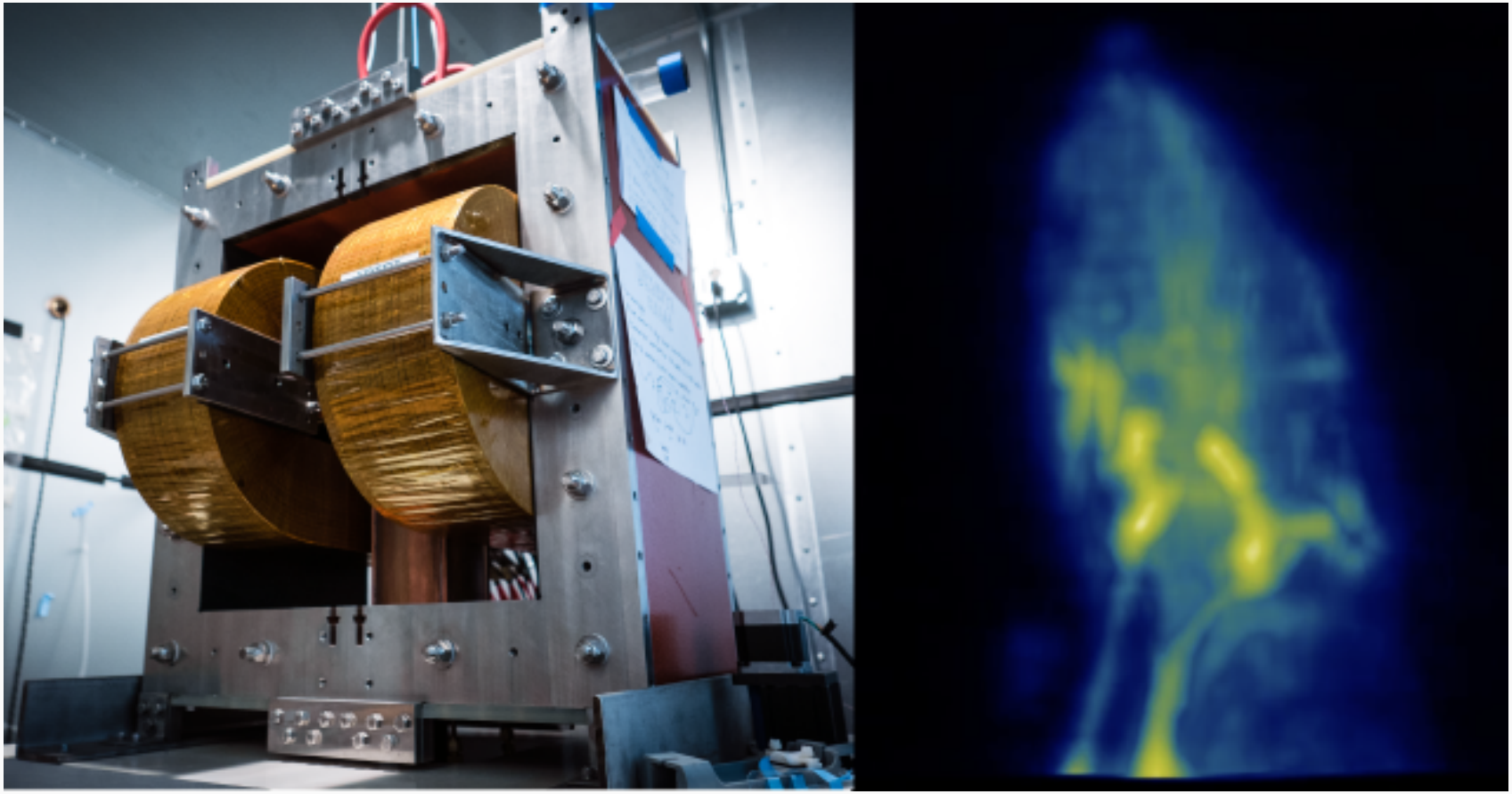
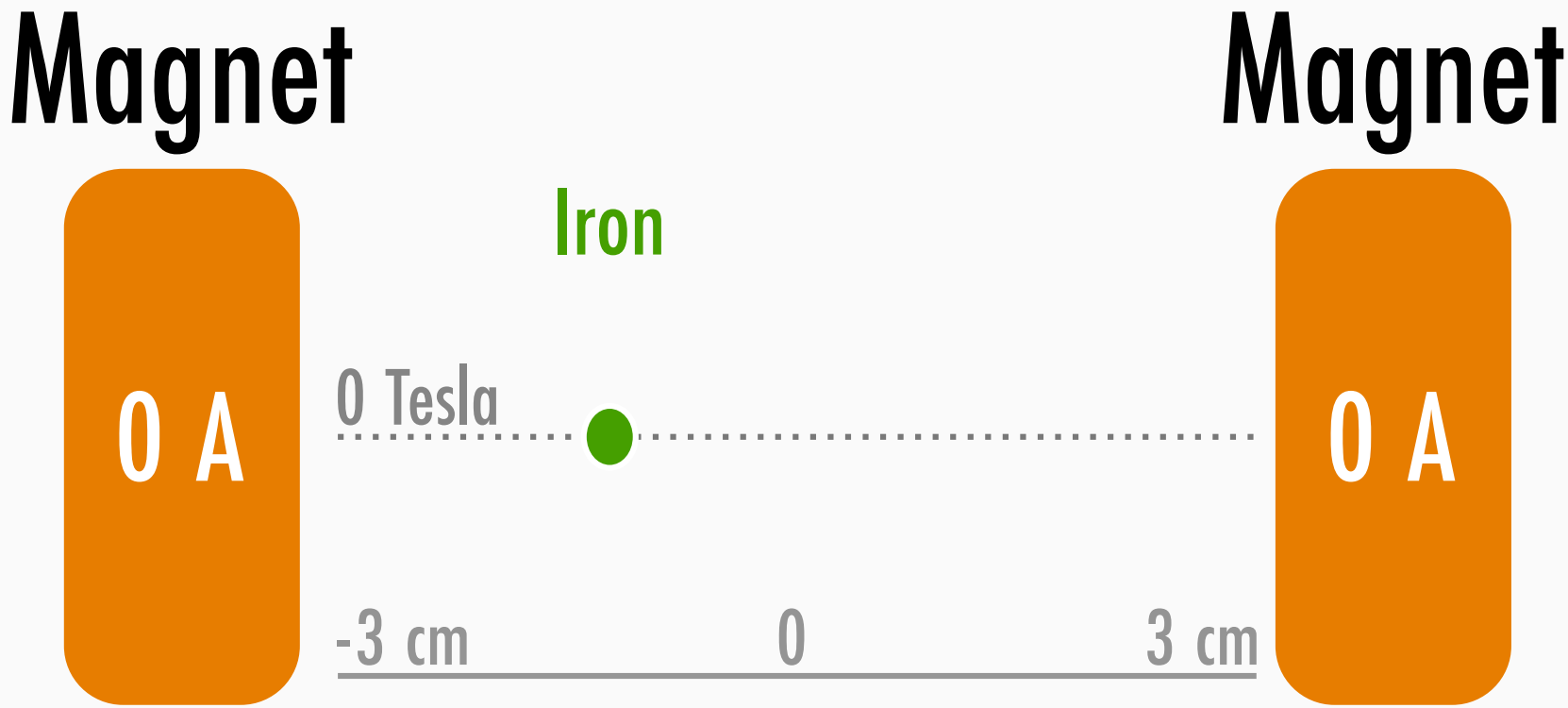


Figure 1: Left: MPI Hardware. Right: Can you guess? Courtesy Conolly Lab, UC Berkeley

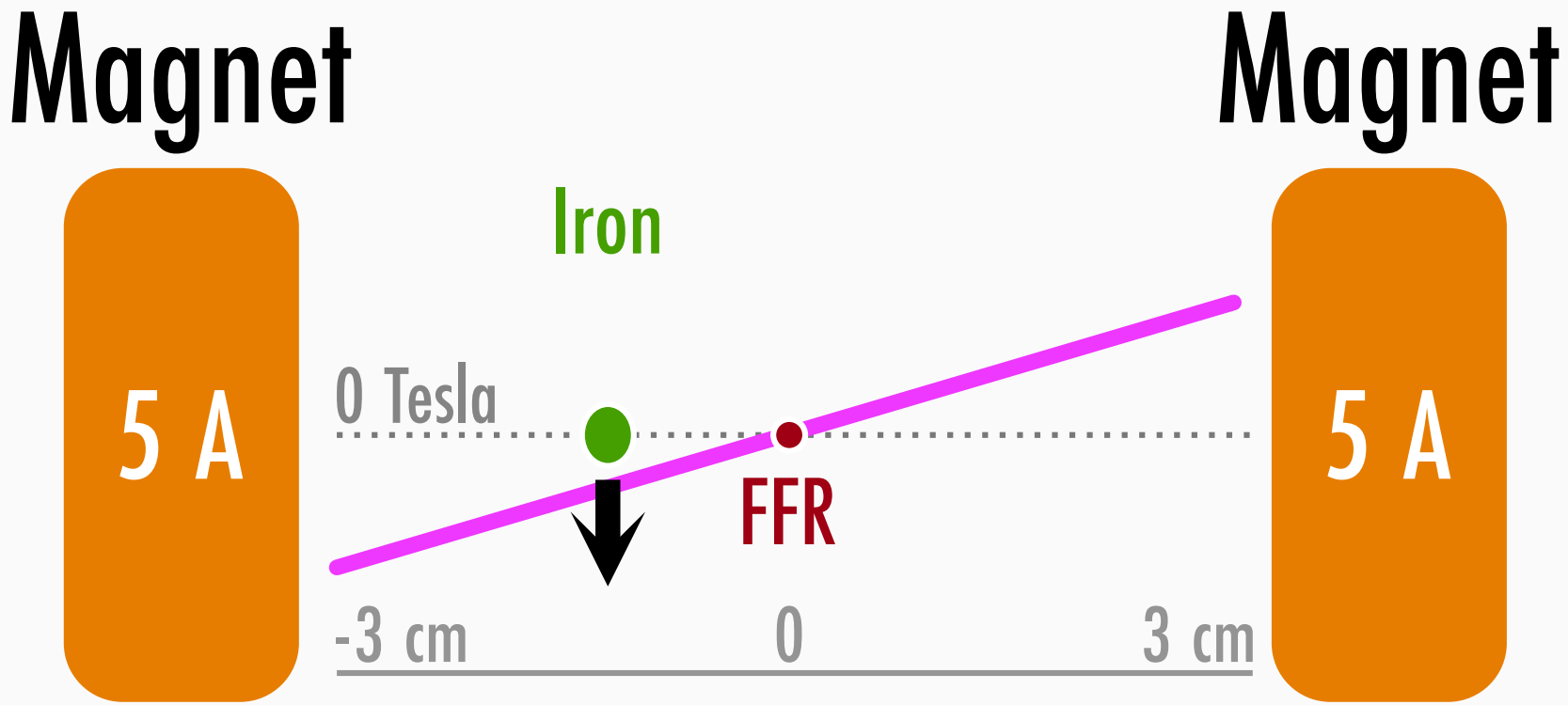
MPI is all about moving around a gradient magnetic field

In MPI, we move around a magnetic field to detect where an iron sample is.



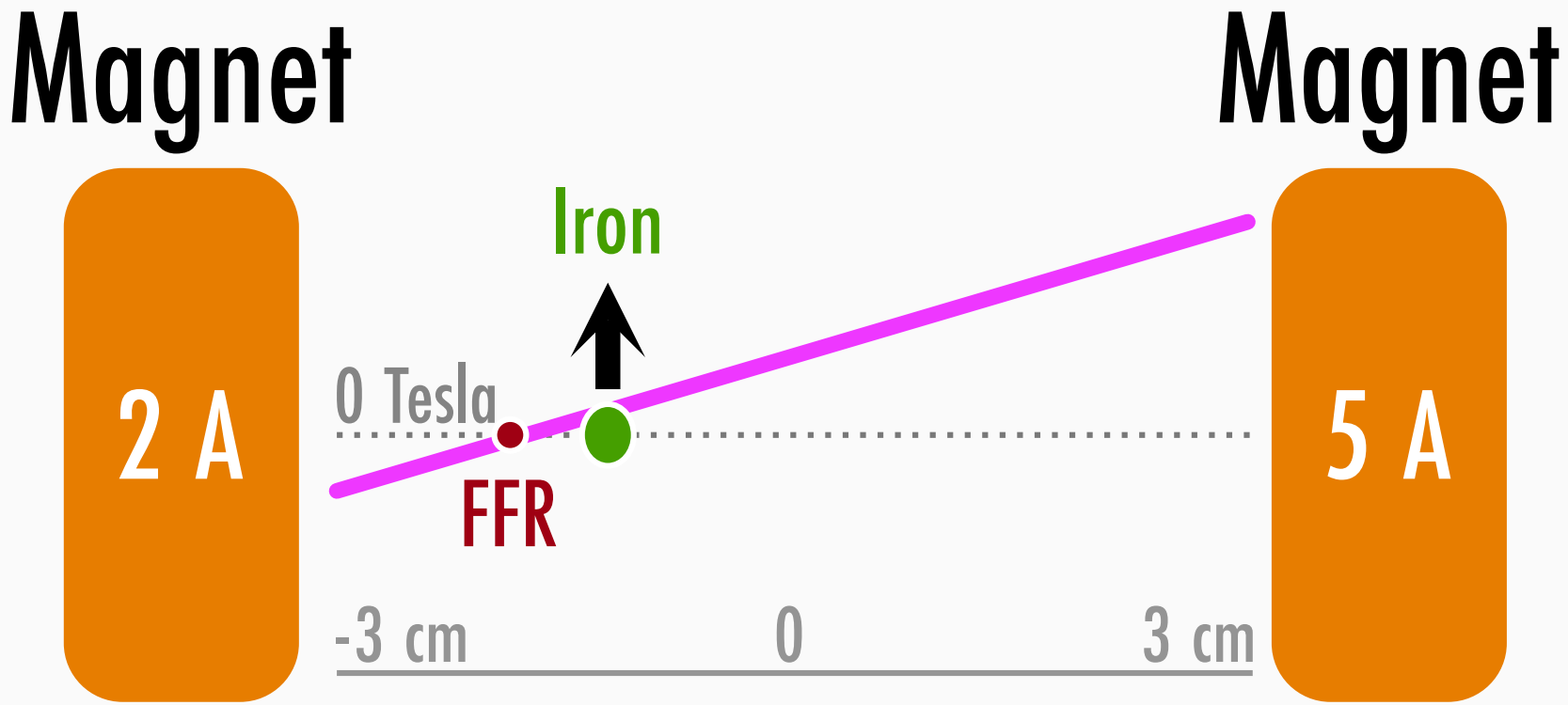
MPI is all about moving around a gradient magnetic field

In MPI, we move around a magnetic field to detect where an iron sample is.



MPI is all about moving around a gradient magnetic field

In MPI, we move around a magnetic field to detect where an iron sample is.



Components used in MPI systems

MPI systems require quite a few components to be handled by the software to acquire an image.

- Auxillary devices (DAQs, custom electronics, motors, safety systems).

Components used in MPI systems

MPI systems require quite a few components to be handled by the software to acquire an image.

- Auxillary devices (DAQs, custom electronics, motors, safety systems).
- GPUs.

Components used in MPI systems

MPI systems require quite a few components to be handled by the software to acquire an image.

- Auxillary devices (DAQs, custom electronics, motors, safety systems).
- GPUs.
- Software safety systems/daemons.

Components used in MPI systems

MPI systems require quite a few components to be handled by the software to acquire an image.

- Auxillary devices (DAQs, custom electronics, motors, safety systems).
- GPUs.
- Software safety systems/daemons.
- Service level access to debug information/tools.

Components used in MPI systems

MPI systems require quite a few components to be handled by the software to acquire an image.

- Auxillary devices (DAQs, custom electronics, motors, safety systems).
- GPUs.
- Software safety systems/daemons.
- Service level access to debug information/tools.

Components used in MPI systems

MPI systems require quite a few components to be handled by the software to acquire an image.

- Auxillary devices (DAQs, custom electronics, motors, safety systems).
- GPUs.
- Software safety systems/daemons.
- Service level access to debug information/tools.

Pinning all this down can be hard!

System Diagram of what is required

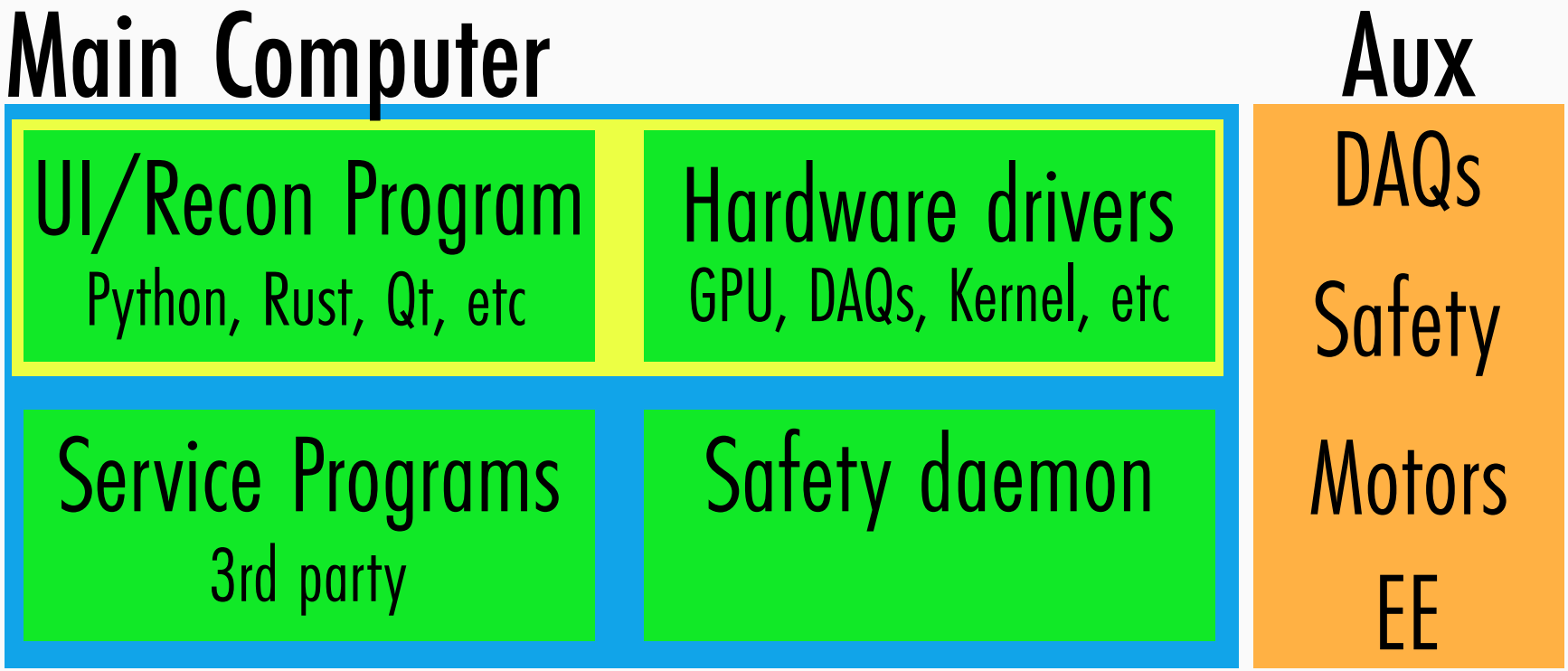


Figure 2: System layout. Items in blue box are located on NixOS machine; items in orange box are connected by networking

And all the connections!

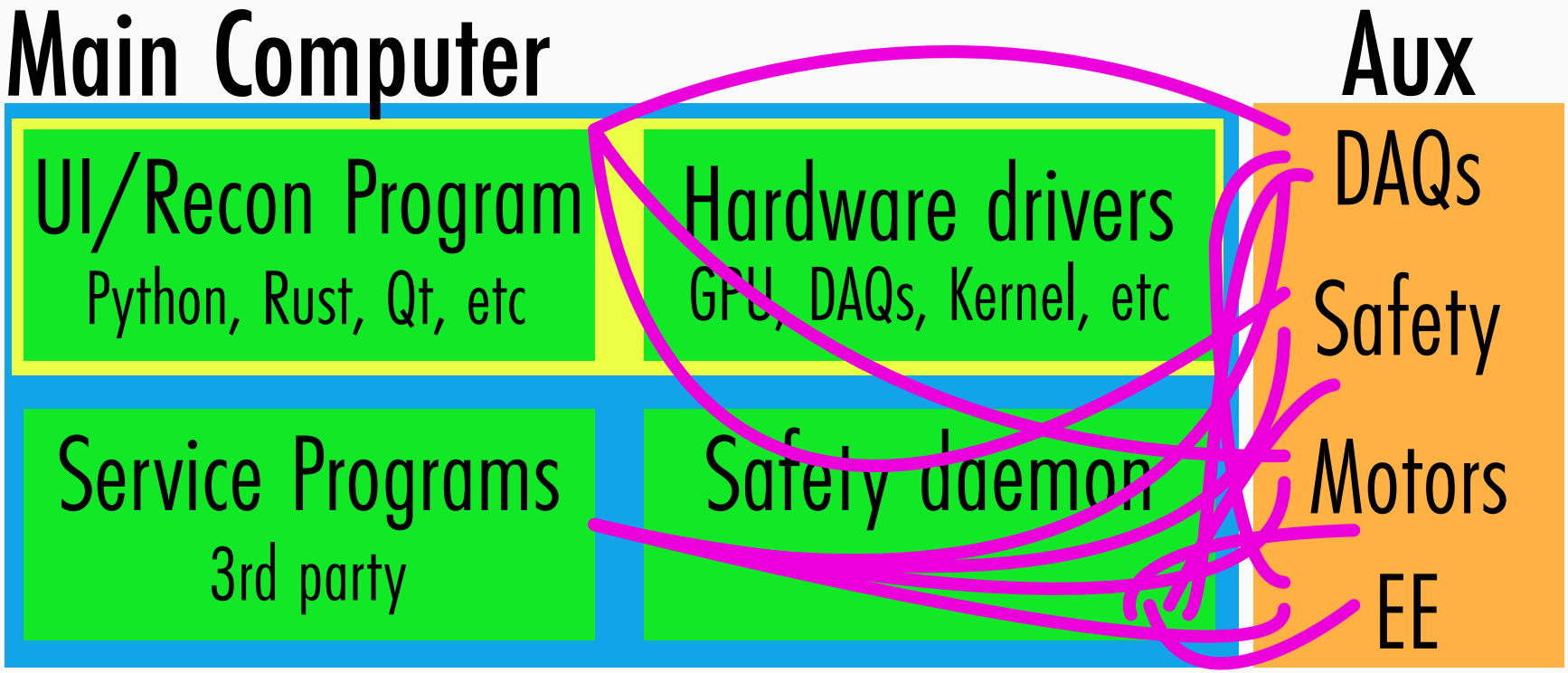


Figure 3: System layout. Items in blue box are located on NixOS machine; items in orange box are connected by networking. Pink is a dependency

Prior challenges faced

Previously the MPI systems have been run on Windows systems. This presents some challenges.

- Creating the same setup twice is hard.

Prior challenges faced

Previously the MPI systems have been run on Windows systems. This presents some challenges.

- Creating the same setup twice is hard.
- Creating the same setup *over time* is very hard.

Prior challenges faced

Previously the MPI systems have been run on Windows systems. This presents some challenges.

- Creating the same setup twice is hard.
- Creating the same setup *over time* is very hard.
- Windows likes to update itself.

Prior challenges faced

Previously the MPI systems have been run on Windows systems. This presents some challenges.

- Creating the same setup twice is hard.
- Creating the same setup *over time* is very hard.
- Windows likes to update itself.
- Upgrading/servicing devices in the field detailed knowledge of the changes being applied.

Our Requirements for NixOS

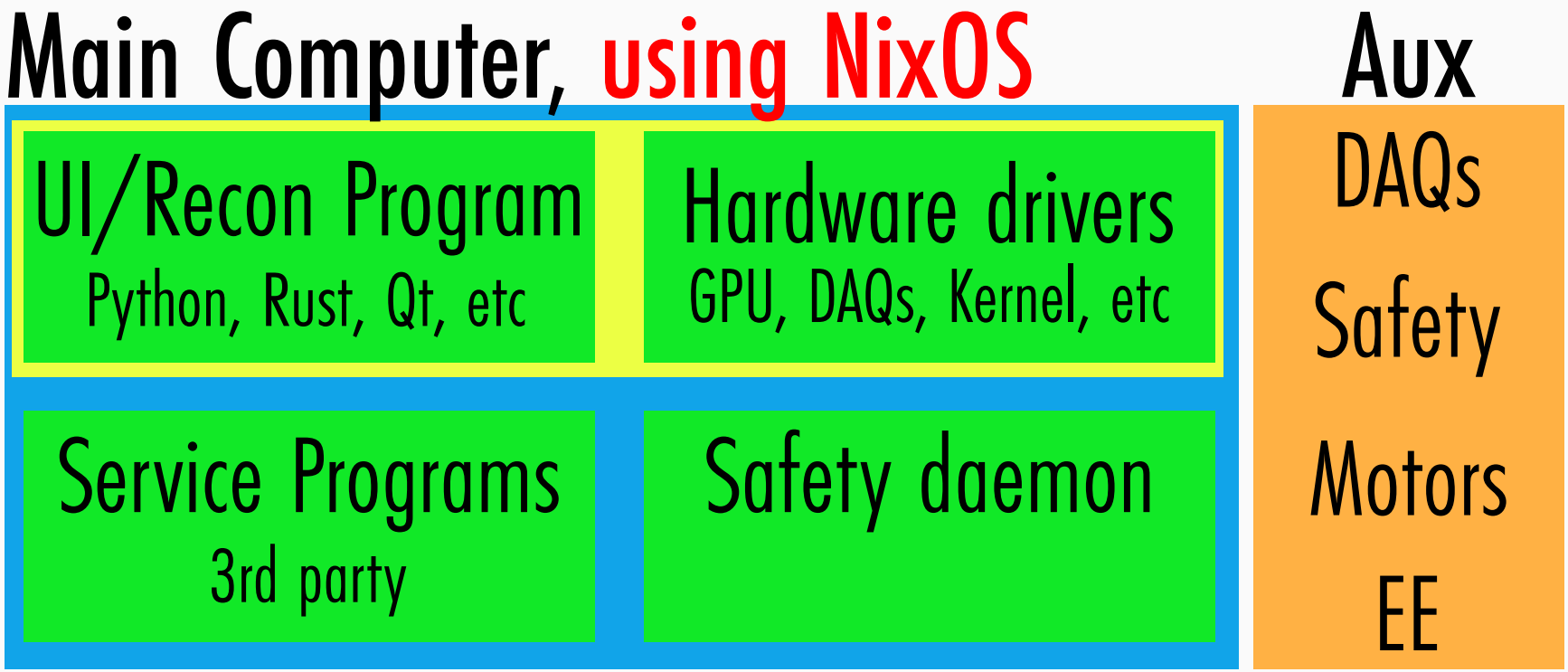
To solve our pain points, we defined the following requirements.

- An easy deployment strategy that any developer/field technician can run.
- No knowledge of the system version changes required.
- Well defined system state, both for our software and the OS.

NixOS to the rescue!

We used NixOS to pin down a bunch of our software stack

We decided to convert the main machine to NixOS. Let's go through each block in turn! What can we *learn* from each subsystem?



Packaging the main environment: Python Advantages

Python has some interesting packaging challenges. System libraries are not often specified by dependency system.

Packaging the main environment: Python Advantages

Python has some interesting packaging challenges. System libraries are not often specified by dependency system.

```
stdenv.mkDerivation {
  ...
  buildInputs = pkgs.python37.withPackages (p: with p; [
    numpy
    scipy
    pyfftw
  ]);
  propagatedBuildInputs = [ pkgs.fftw ];
}
```

Packaging the main environment: Python Challenges

If nixpkgs does not have your desired python package, it can be included easily using an overlay. Here we

```
python37.override {
  packageOverrides = (self: super:
    coloredlogs = self.buildPythonPackage {...};
  )
}
```

Packaging the main environment: Python Challenges

If nixpkgs does not have your desired python package, it can be included easily using an overlay. Here we

```
python37.override {
  packageOverrides = (self: super:
    coloredlogs = self.buildPythonPackage {...};
  )
}
```

But!

- You may need to wrap a bunch of other python dependencies.

Packaging the main environment: Python Challenges

If nixpkgs does not have your desired python package, it can be included easily using an overlay. Here we

```
python37.override {
  packageOverrides = (self: super:
    coloredlogs = self.buildPythonPackage {...};
  )
}
```

But!

- You may need to wrap a bunch of other python dependencies.
- You may need to do some manual dependency resolution (instead of `pip`).

Packaging the main environment: Python Challenges

If nixpkgs does not have your desired python package, it can be included easily using an overlay. Here we

```
python37.override {
  packageOverrides = (self: super:
    coloredlogs = self.buildPythonPackage {...};
  )
}
```

But!

- You may need to wrap a bunch of other python dependencies.
- You may need to do some manual dependency resolution (instead of `pip`).

Packaging the main environment: Python Challenges

If nixpkgs does not have your desired python package, it can be included easily using an overlay. Here we

```
python37.override {
  packageOverrides = (self: super:
    coloredlogs = self.buildPythonPackage {...};
  )
}
```

But!

- You may need to wrap a bunch of other python dependencies.
- You may need to do some manual dependency resolution (instead of `pip`).

Lesson: python support is good but you may spend quite a bit of time defining dependencies.

Packaging the main environment: Rust

Rust is pretty simple to package, just use `buildRustCrate` or `buildRustPackage`. These functions allow you to conveniently package anything that has a `Cargo.lock` file.

Packaging the main environment: Rust

Rust is pretty simple to package, just use `buildRustCrate` or `buildRustPackage`. These functions allow you to conveniently package anything that has a `Cargo.lock` file.

On the less convenient side:

Since we use `cargo` in one big derivation, any changes requires a complete rebuild. Can be solved with `cargo2nix`.

Packaging the main environment: Rust

Rust is pretty simple to package, just use `buildRustCrate` or `buildRustPackage`. These functions allow you to conveniently package anything that has a `Cargo.lock` file.

On the less convenient side:

Since we use `cargo` in one big derivation, any changes requires a complete rebuild. Can be solved with `cargo2nix`.

Lesson: the basics for Rust work but you want to use some tooling (like `cargo2nix`, bazel) to reduce build times.

How to package proprietary drivers

How to package proprietary drivers

Don't. It can be quite difficult.

How to package proprietary drivers

Proprietary drivers may rely on a certain file structure (FHS). For example, the driver may assume `/usr/local/lib` exists.

How to package proprietary drivers

Proprietary drivers may rely on a certain file structure (FHS). For example, the driver may assume `/usr/local/lib` exists.

To get around this, you can bootstrap a chroot environment using the building blocks of `buildFHSUserEnv`.

```
let
  pkgs = import <nixpkgs> {};
  chrootenv = pkgs.callPackage (pkgs.path +
    "/pkgs/build-support/build-fhs-userenv/chrootenv/")
    {};
in
  # can now use the ${chrootenv}/bin/chrootenv binary.
```

How to package proprietary drivers

Proprietary drivers may rely on a certain file structure (FHS). For example, the driver may assume `/usr/local/lib` exists.

To get around this, you can bootstrap a chroot environment using the building blocks of `buildFHSUserEnv`.

```
let
  pkgs = import <nixpkgs> {};
  chrootenv = pkgs.callPackage (pkgs.path +
    "/pkgs/build-support/build-fhs-userenv/chrootenv/")
    {};
in
  # can now use the ${chrootenv}/bin/chrootenv binary.
```

Lesson: FHS assumptions are common in software; you may need to roll your own solutions/do a lot of patching.

How to package UI user programs

We ship with certain programs for image analysis (3D Slicer).

Often these assume standard FHS places to load libraries.

How to package UI user programs

We ship with certain programs for image analysis (3D Slicer).
Often these assume standard FHS places to load libraries.

The trick for these is to

- Use `buildFHSUserEnv` to make a fake FHS environment.

How to package UI user programs

We ship with certain programs for image analysis (3D Slicer).
Often these assume standard FHS places to load libraries.

The trick for these is to

- Use `buildFHSUserEnv` to make a fake FHS environment.
- Use programs like `ldd` and `strace` to find program dependencies.

How to package UI user programs

We ship with certain programs for image analysis (3D Slicer).
Often these assume standard FHS places to load libraries.

The trick for these is to

- Use `buildFHSUserEnv` to make a fake FHS environment.
- Use programs like `ldd` and `strace` to find program dependencies.
  - Programs do not always specify all their dependencies correctly in their READMEs/package management systems! :-(

How to package UI user programs

We ship with certain programs for image analysis (3D Slicer).
Often these assume standard FHS places to load libraries.

The trick for these is to

- Use `buildFHSUserEnv` to make a fake FHS environment.
- Use programs like `ldd` and `strace` to find program dependencies.
  - Programs do not always specify all their dependencies correctly in their READMEs/package management systems! :-(
- Look at other GUI programs in nixpkgs for guidance, but you'll need to look at the nix expressions directly.

How to package UI user programs

We ship with certain programs for image analysis (3D Slicer).
Often these assume standard FHS places to load libraries.

The trick for these is to

- Use `buildFHSUserEnv` to make a fake FHS environment.
- Use programs like `ldd` and `strace` to find program dependencies.
  - Programs do not always specify all their dependencies correctly in their READMEs/package management systems! :-(
- Look at other GUI programs in nixpkgs for guidance, but you'll need to look at the nix expressions directly.

How to package UI user programs

We ship with certain programs for image analysis (3D Slicer).
Often these assume standard FHS places to load libraries.

The trick for these is to

- Use `buildFHSUserEnv` to make a fake FHS environment.
- Use programs like `ldd` and `strace` to find program dependencies.
  - Programs do not always specify all their dependencies correctly in their READMEs/package management systems! :-(
- Look at other GUI programs in nixpkgs for guidance, but you'll need to look at the nix expressions directly.

Lesson: patterns in nixpkgs are a primary way of solving a problem.

How to package UI user programs

We ship with certain programs for image analysis (3D Slicer).
Often these assume standard FHS places to load libraries.

The trick for these is to

- Use `buildFHSUserEnv` to make a fake FHS environment.
- Use programs like `ldd` and `strace` to find program dependencies.
  - Programs do not always specify all their dependencies correctly in their READMEs/package management systems! :-(
- Look at other GUI programs in nixpkgs for guidance, but you'll need to look at the nix expressions directly.

Lesson: patterns in nixpkgs are a primary way of solving a problem.

Lesson: you can apply patches before they are in the main code.

How to package service/hardware programs

These often come as debian/RPM packages. You'll can patch binaries to use `/nix/store` paths for library dependencies.

How to package service/hardware programs

These often come as debian/RPM packages. You'll can patch binaries to use `/nix/store` paths for library dependencies.

```
stdenv.mkDerivation rec {
  src = ./something.deb; nativeBuildInputs = [ dpkg ];
  unpackPhase = "dpkg -x $src .";

  preFixup = let
    libPath = lib.makeLibraryPath [ stdenv.cc.cc.lib ];
  in ''for f in $out/lib/lib* ; do
        patchelf --set-rpath "${libPath}:$out/lib" $f
      done'';}
```

How to package service/hardware programs

These often come as debian/RPM packages. You'll can patch binaries to use `/nix/store` paths for library dependencies.

```
stdenv.mkDerivation rec {
  src = ./something.deb; nativeBuildInputs = [ dpkg ];
  unpackPhase = "dpkg -x $src .";

  preFixup = let
    libPath = lib.makeLibraryPath [ stdenv.cc.cc.lib ];
  in ''for f in $out/lib/lib* ; do
         patchelf --set-rpath "${libPath}:$out/lib" $f
       done'';}
```

Lesson: Handling packages from other package managers is not too bad; dependency management is manual.

How to package service/hardware programs: the hard times

Programs for hardware are often only for Windows. In these cases, NixOS has great virtualbox support. You can enable it with the following flag in your `configuration.nix` file (or equivalent).

```
virtualisation.virtualbox.host.enable = true;
```

How to package service/hardware programs: the hard times

Programs for hardware are often only for Windows. In these cases, NixOS has great virtualbox support. You can enable it with the following flag in your `configuration.nix` file (or equivalent).

```
virtualisation.virtualbox.host.enable = true;
```

Some packages have such extensive requirements that a decent patch/first step is getting the program running under docker.

```
virtualisation.docker.enable = true;
```

How to package service/hardware programs: the hard times

Programs for hardware are often only for Windows. In these cases, NixOS has great virtualbox support. You can enable it with the following flag in your `configuration.nix` file (or equivalent).

```
virtualisation.virtualbox.host.enable = true;
```

Some packages have such extensive requirements that a decent patch/first step is getting the program running under docker.

```
virtualisation.docker.enable = true;
```

Lesson: You won't always succeed in the attempts to package things in a timely manner. But there are escape hatches.

How to pin the whole system

Pin using the method Gabriel mentions.

http://www.haskellforall.com/2018/08/nixos-in-production.html

How to pin the whole system

Pin using the method Gabriel mentions.

http://www.haskellforall.com/2018/08/nixos-in-production.html

```
let
  nixpkgs-src = builtins.fetchTarball { ... };
in import "${nixpkgs-src}/nixos" {
  system = "x86_64-linux";
  configuration = import ./configuration.nix;
}
```

How to pin the whole system

Pin using the method Gabriel mentions.
http://www.haskellforall.com/2018/08/nixos-in-production.html

```
let
  nixpkgs-src = builtins.fetchTarball { ... };
in import "${nixpkgs-src}/nixos" {
  system = "x86_64-linux";
  configuration = import ./configuration.nix;
}
```

Lesson: Pinning the whole system make deployment wasy and help identify missing dependencies.

Pro tip: Use `nix-diff` to determine the difference between two systems!

How to package services

Packaging up service is pretty simple. Just import a file like this into your `configuration.nix`

```
let

  cfg = config.services.vnc;

in {

  options.services.vnc.enable = mkEnableOption "vnc";
  options.services.vnc.port = mkOption { ... };

  config = mkIf (cfg.enable) {systemd.services.vnc = {...};
```

How to package services

Packaging up service is pretty simple. Just import a file like this into your `configuration.nix`

```
let

  cfg = config.services.vnc;

in {

  options.services.vnc.enable = mkEnableOption "vnc";
  options.services.vnc.port = mkOption { ... };

  config = mkIf (cfg.enable) {systemd.services.vnc = {...};
```

Lesson: After defining a package, it is relatively painless to define a daemon.

Deployment strategy

Deployment is pretty simple after these changes.

1. Define a system configuration with everything needed.
2. Build that system using `nix-build`
3. Copy to a remote system using `nix-copy-closure`
4. Run `switch-to-configuration` to change the system over to the new state.

Deployment strategy

Deployment is pretty simple after these changes.

1. Define a system configuration with everything needed.
2. Build that system using `nix-build`
3. Copy to a remote system using `nix-copy-closure`
4. Run `switch-to-configuration` to change the system over to the new state.

This method also gives us the flexibility to copy the closure onto a USB device and then copy that onto a remote system.

Deployment strategy

Deployment is pretty simple after these changes.

1. Define a system configuration with everything needed.
2. Build that system using `nix-build`
3. Copy to a remote system using `nix-copy-closure`
4. Run `switch-to-configuration` to change the system over to the new state.

This method also gives us the flexibility to copy the closure onto a USB device and then copy that onto a remote system.

Lesson: Anyone can do deployment with a few commands that can be scripted.

Lessons Learned

Lessons learned summary

We found out a few things the hard way.

- It is easy to spend a lot of time on packaging something when it is not defined in nixpkgs. But you get a reproducible build for your work!

Lessons learned summary

We found out a few things the hard way.

- It is easy to spend a lot of time on packaging something when it is not defined in nixpkgs. But you get a reproducible build for your work!
- Software is not always forthcoming with all of its assumptions on the system state.

Lessons learned summary

We found out a few things the hard way.

- It is easy to spend a lot of time on packaging something when it is not defined in nixpkgs. But you get a reproducible build for your work!
- Software is not always forthcoming with all of its assumptions on the system state.
- Some code (especially proprietary) is too hard to wrap. You can use docker/virtualbox as escape hatches to work on your main goal.

Lessons learned summary

We found out a few things the hard way.

- It is easy to spend a lot of time on packaging something when it is not defined in nixpkgs. But you get a reproducible build for your work!
- Software is not always forthcoming with all of its assumptions on the system state.
- Some code (especially proprietary) is too hard to wrap. You can use docker/virtualbox as escape hatches to work on your main goal.
- You will end up a great understanding of your dependencies; those dependencies will be documented in the process of becoming nix expressions.

Did we reach our goals through this process?

Did we reach the goals we had before?

- An easy deployment strategy that any developer/field technician can run. ✓

Did we reach our goals through this process?

Did we reach the goals we had before?

- An easy deployment strategy that any developer/field technician can run. ✓
- No knowledge of the system version changes required. ✓

Did we reach our goals through this process?

Did we reach the goals we had before?

- An easy deployment strategy that any developer/field technician can run. ✓
- No knowledge of the system version changes required. ✓
- Well defined system state. ✓

What do our developers think of Nix/NixOS?

How well has the Nix transition worked for our developers?

- They love the reproducibility.

What do our developers think of Nix/NixOS?

How well has the Nix transition worked for our developers?

- They love the reproducibility.
- They are able to get set up with the software quickly.

What do our developers think of Nix/NixOS?

How well has the Nix transition worked for our developers?

- They love the reproducibility.
- They are able to get set up with the software quickly.
- We will need training to get people to make their own nix expressions.

Questions?