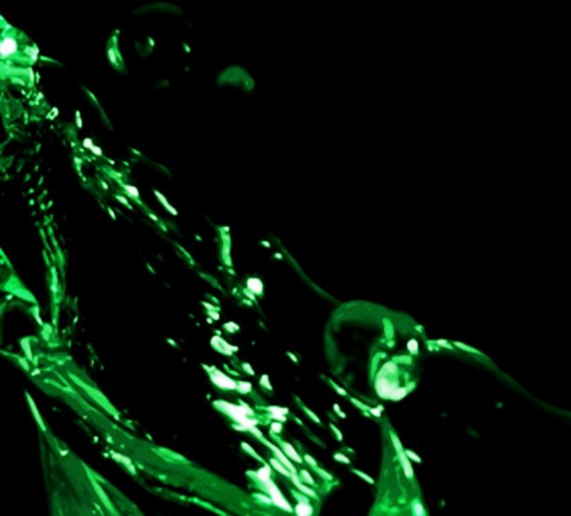
Update Strategies for the Edge

There's a better way.



Kat Cosgrove

IoT Engineer

Developer Advocate

Twitter: @Dixie3Flatline

Email: katc@jfrog.com

jfrog.com/shownotes

Agenda

Part One

Introduction

Problem Domain

What Needs to Change

Proof of Concept

Software Workflow

Firmware Workflow

Other Tools

Part Two

The Live Demo

What's a Donkey Car?

More Natural Controls

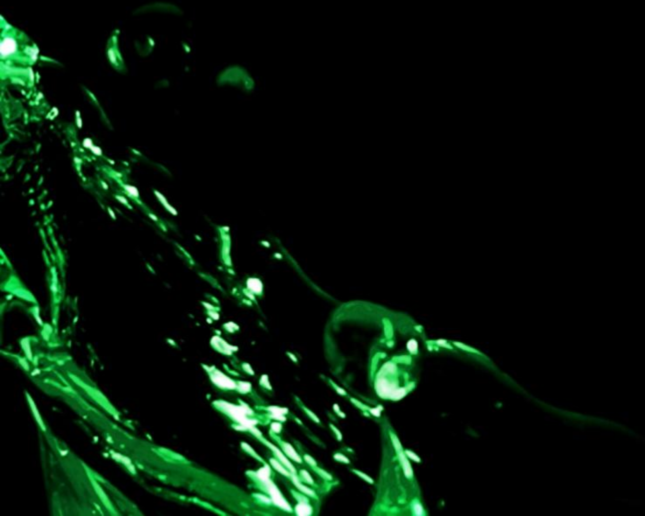
The Driver's Seat

Adding a Green Screen

Automating Training

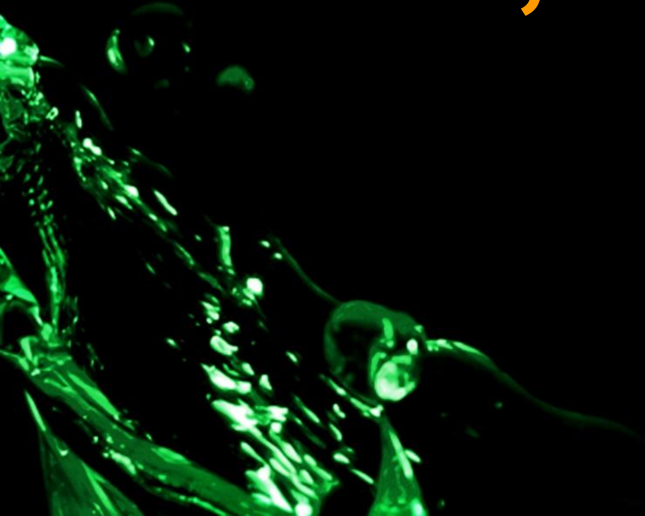
Thank you!

How large is the Edge?



20,400,000,000

That's a lot of devices.



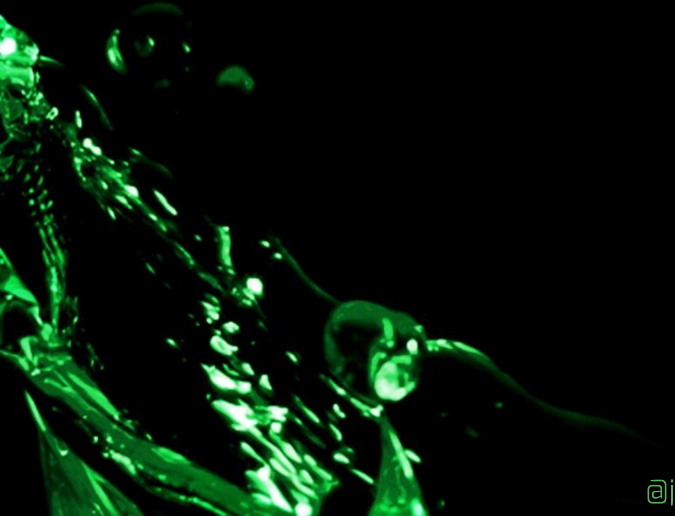
Updates Today

They don't update; device is effectively single-use

OR

It's time-consuming, complicated, or requires physical access

Why change?



It's inconvenient

Edge computing is massive and growing

- Consumer
- Industrial
- Medical

Slow OTA updates are annoying

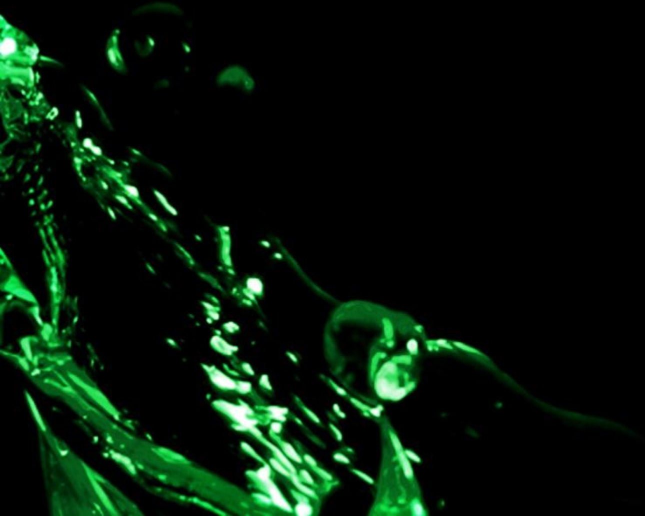
Wired updates are expensive and more annoying

It’s dangerous

Unpatched bugs can be a huge vulnerability

- Expose private data
- Harnessed for a botnet
- Used for cryptocurrency mining
- Safety implications for medical

What’s slowing us down?



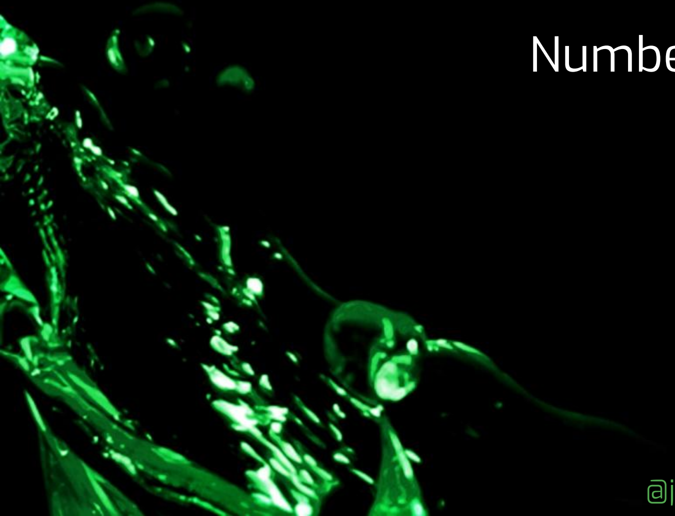
Not building for it.

Many devices are not made to be updated.

- Designed to run one version until the end
- “Update strategy” here is flashing the device
- Bugs are inevitable

Between 1 and 25

Number of bugs per 1000 lines of code



Connectivity Concerns

We can’t rely on the device’s network

- Networks may be unstable
- Bandwidth may be low
- Network probably isn’t secure

Hardware Variations

- It's 20.4 billion devices
- Lots of specialized hardware
- Variations in memory, storage space, architecture

How do we design something that handles so much variety?

Think future-forward.

Updates are your friend. Embrace updates, not security nightmares.

Get better with age.

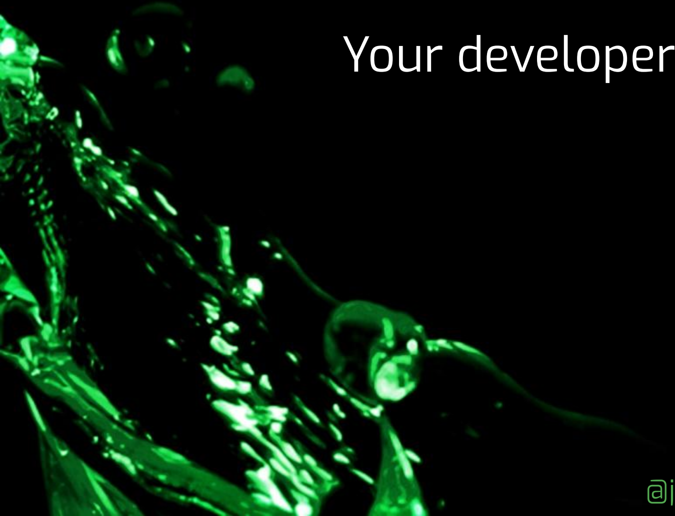
Your product should not be getting worse from the moment it ships.

Build robust.

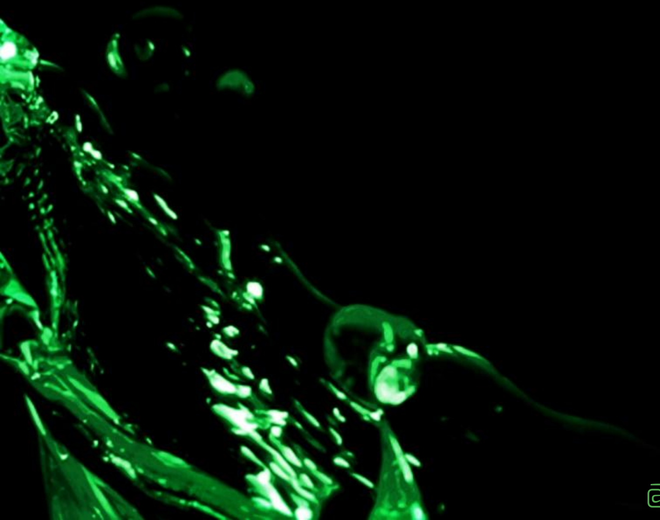
Brittle software means a brittle device, and that doesn’t inspire trust.

Modern DevOps tools.

Your developers will thank you and things will run more smoothly.



The Proof of Concept



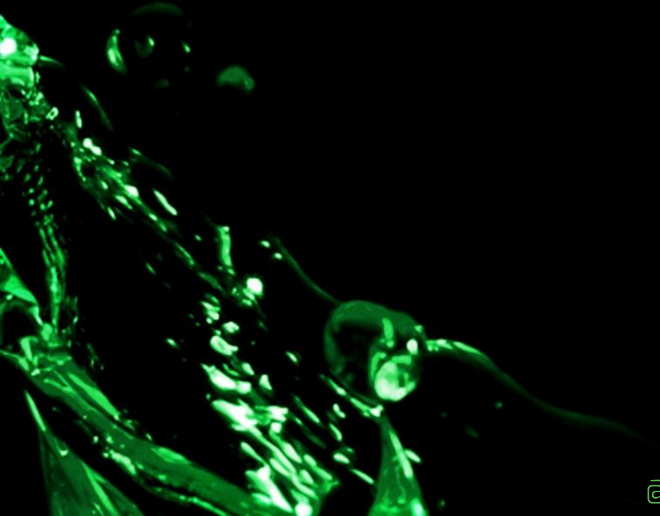
Cars Now

- Majority not designed for OTA updates
- OTA updates are still slow and inconvenient
- Little standardization
- Significant portion of recalls are due to software

Cars as Edge Devices

- Presented a range of solvable pain points in one device
- Tangible example for end users and manufacturers
- Device in question meant speed, reliability, and safety were equally important

Workflows and Tools



Two Distinct Workflows

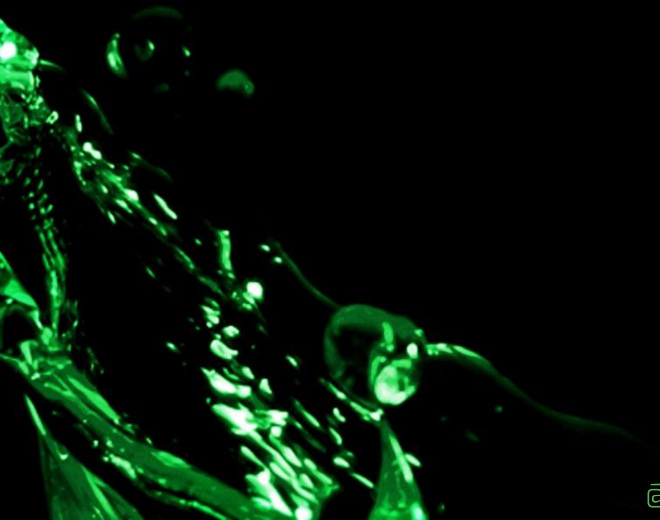
Software Updates

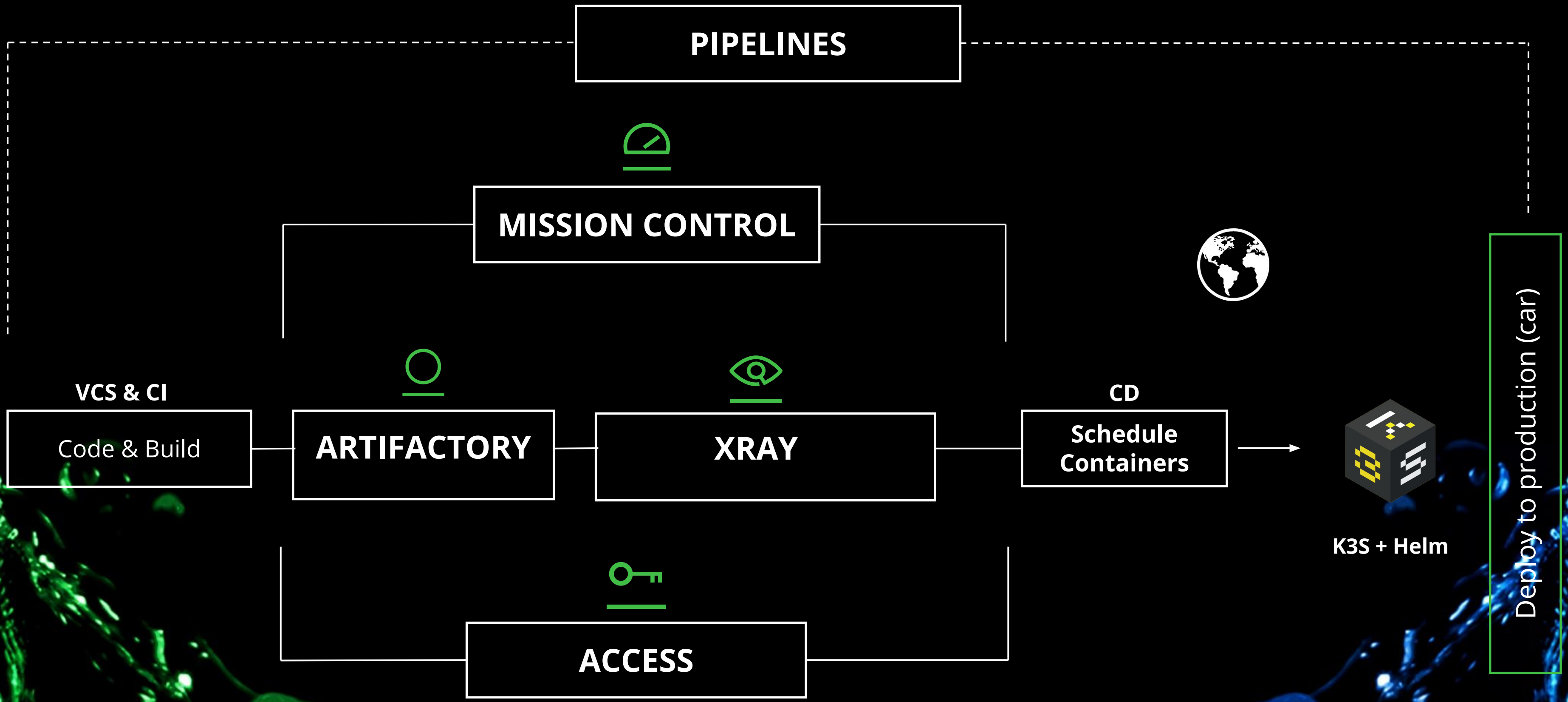
- Without flashing firmware
- No interruption of user
- Takes only seconds
- Relies on K3S and Helm

Firmware Updates

- More difficult update
- Takes only minutes
- Rollback if there is a failure
- Relies on Mender and Yocto

Software Workflow





PIPELINES

MISSION CONTROL

VCS & CI

Code & Build

ARTIFACTORY

XRAY

CD

Schedule Containers

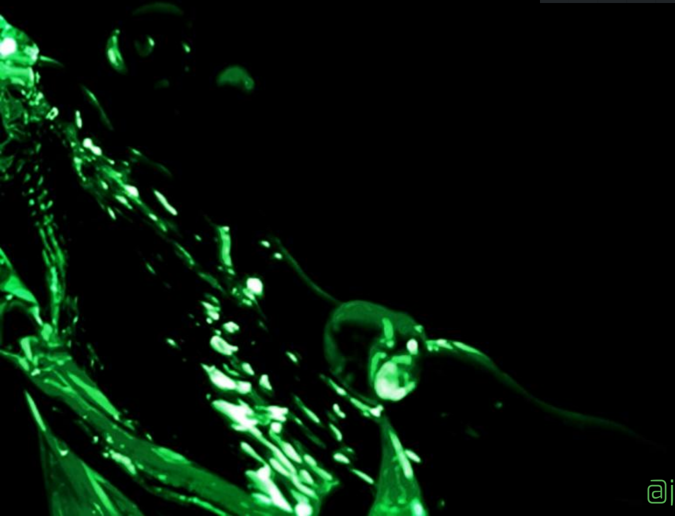
K3S + Helm

Deploy to production (car)

ACCESS

K3S

Kubernetes, but 5 less

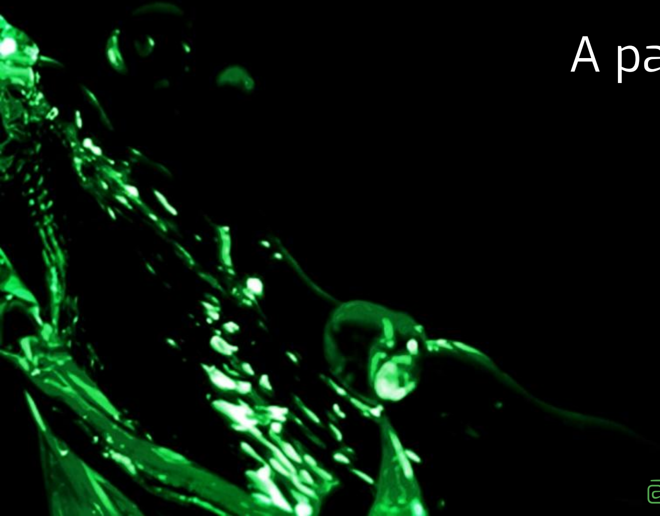


K3S

- Lightweight Kubernetes, designed for Edge devices
- Uses only 512mb of RAM
- 40mb binary
- Very minimal OS requirements

HELM

A package manager for Kubernetes

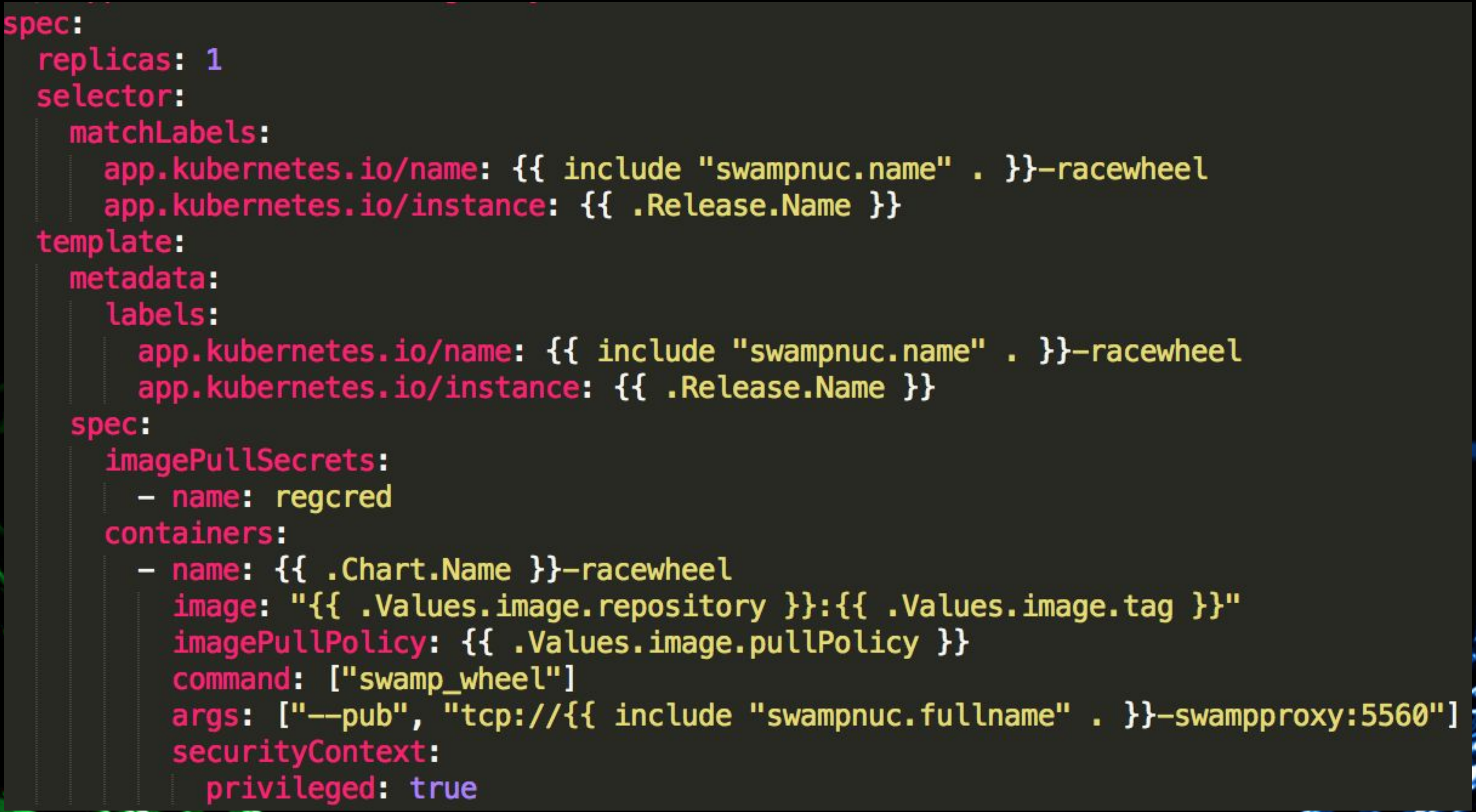


Helm

“Charts” describe complex applications

- Easily repeatable installation
- Final authority on application
- Easy to version
- Supports rollbacks

Helm Charts



```yaml
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: {{ include "swampnuc.name" . }}-racewheel
      app.kubernetes.io/instance: {{ .Release.Name }}
  template:
    metadata:
      labels:
        app.kubernetes.io/name: {{ include "swampnuc.name" . }}-racewheel
        app.kubernetes.io/instance: {{ .Release.Name }}
    spec:
      imagePullSecrets:
        - name: regcred
      containers:
        - name: {{ .Chart.Name }}-racewheel
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
          imagePullPolicy: {{ .Values.image.pullPolicy }}
          command: ["swamp_wheel"]
          args: ["--pub", "tcp://{{ include "swampnuc.fullname" . }}-swampproxy:5560"]
          securityContext:
            privileged: true
```

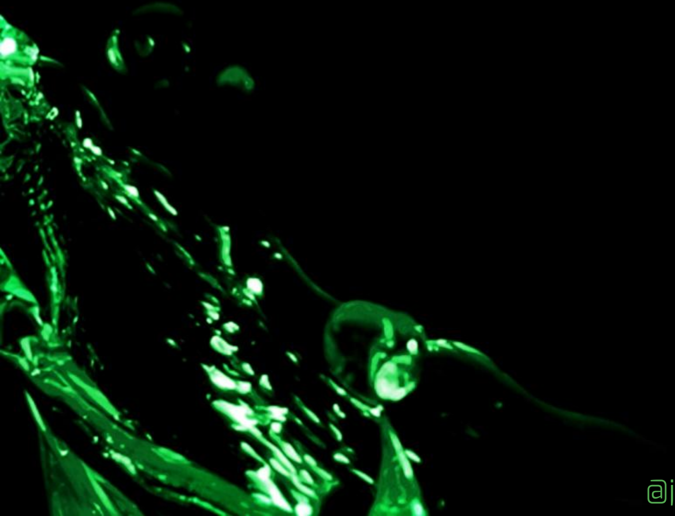
@jfrog |

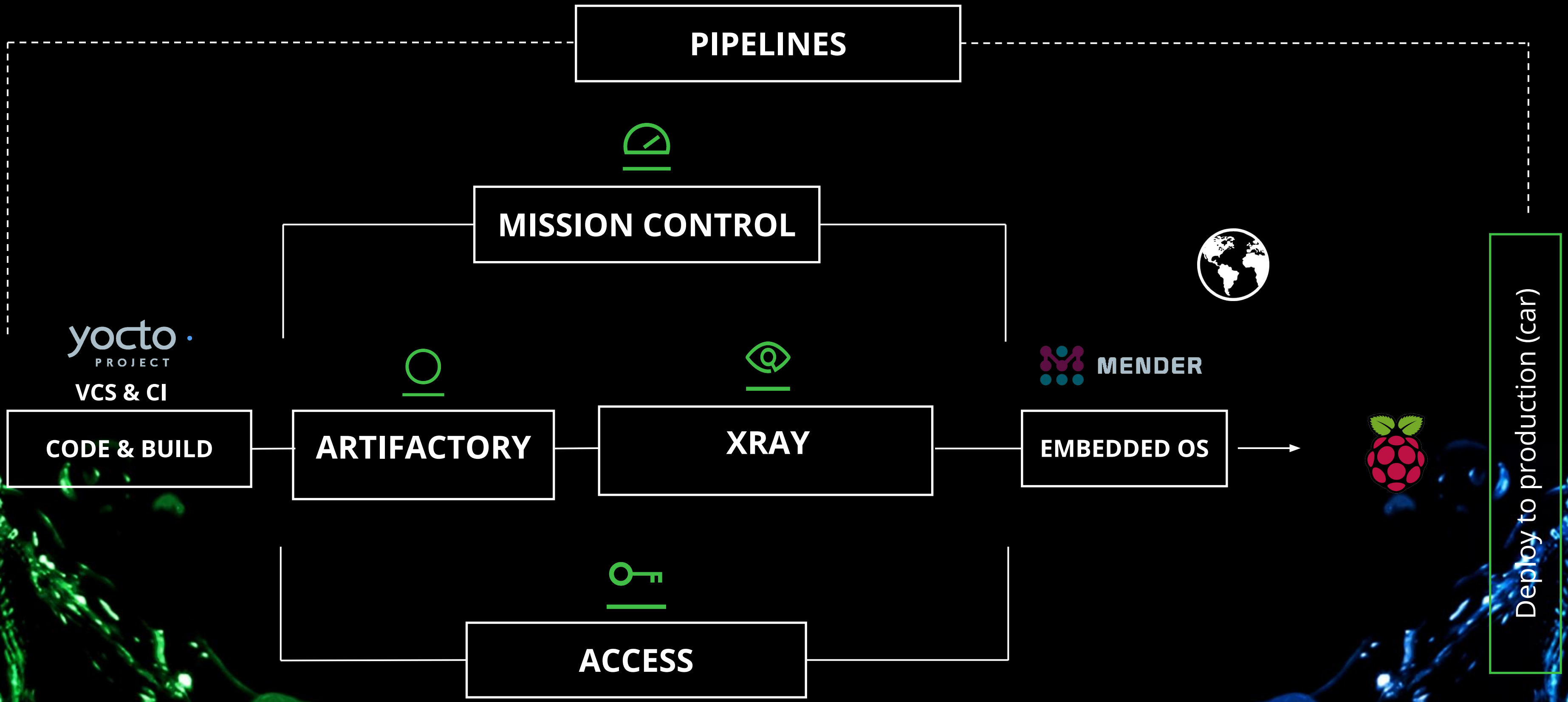
The Result - Software

Application updates are quick and efficient

- Average of 35 seconds from dev to car
- No interruption for the user
- Can happen while device is in use
- Could happen silently, depends on device purpose

Firmware Workflow





PIPELINES

MISSION CONTROL

VCS & CI

CODE & BUILD

ARTIFACTORY

XRAY

EMBEDDED OS

Deploy to production (car)

ACCESS

MENDER

OTA updates for embedded Linux devices

Mender Overview

Ticks several of the boxes we're looking for:

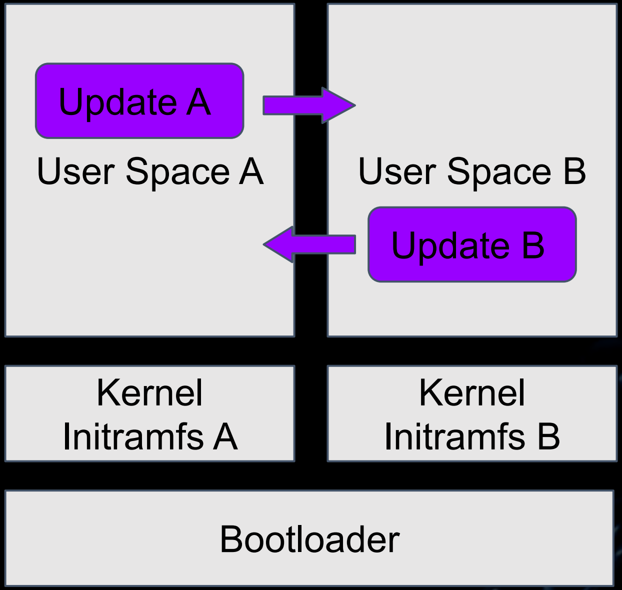
- Updates are signed and verified
- Supports automatic rollbacks
- Several distinct installation strategies
- Dual A/B strategy

Mender - A/B

Two partitions are on the device

- Bootloader aware of “active”
- Update streams to “inactive”
- Automatically revert to previous partition on failure

Now let’s handle the size of our builds.



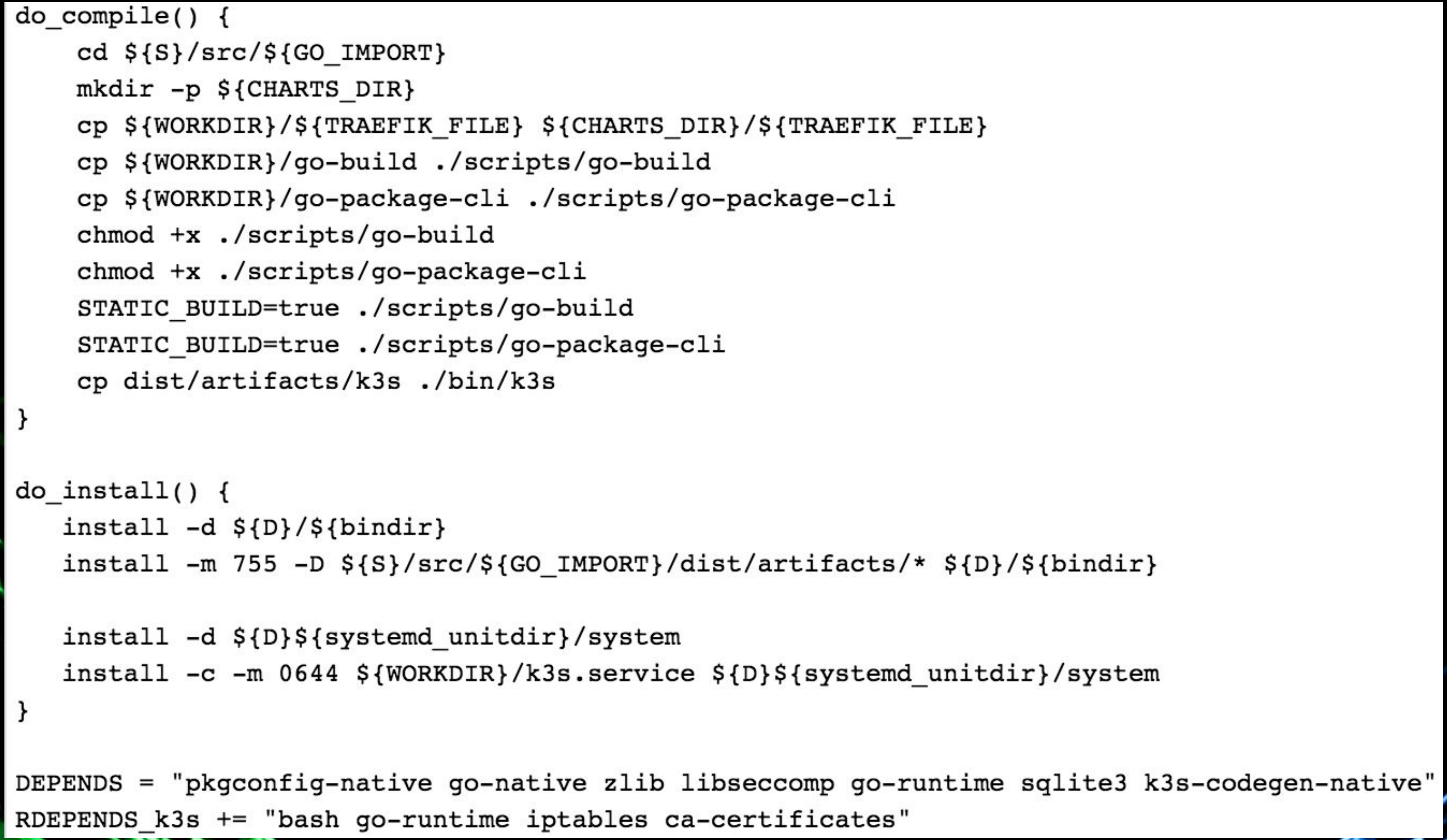
yocto
PROJECT

Custom Linux distributions for any hardware architecture

Yocto Overview

- Eliminates OS bloat
- Drastically reduces resources required
- BitBake recipes and layers define your build
- Layers for common configurations are provided
- Custom layers to isolate applications or behaviors

Yocto Layers



```
do_compile() {
    cd ${S}/src/${GO_IMPORT}
    mkdir -p ${CHARTS_DIR}
    cp ${WORKDIR}/${TRAEFIK_FILE} ${CHARTS_DIR}/${TRAEFIK_FILE}
    cp ${WORKDIR}/go-build ./scripts/go-build
    cp ${WORKDIR}/go-package-cli ./scripts/go-package-cli
    chmod +x ./scripts/go-build
    chmod +x ./scripts/go-package-cli
    STATIC_BUILD=true ./scripts/go-build
    STATIC_BUILD=true ./scripts/go-package-cli
    cp dist/artifacts/k3s ./bin/k3s
}

do_install() {
   install -d ${D}/${bindir}
   install -m 755 -D ${S}/src/${GO_IMPORT}/dist/artifacts/* ${D}/${bindir}

   install -d ${D}${systemd_unitdir}/system
   install -c -m 0644 ${WORKDIR}/k3s.service ${D}${systemd_unitdir}/system
}

DEPENDS = "pkgconfig-native go-native zlib libseccomp go-runtime sqlite3 k3s-codegen-native"
RDEPENDS_k3s += "bash go-runtime iptables ca-certificates"
```

Yocto and Artifactory

- After first build, we can make things much faster
- Yocto cache allows for incremental updates
- Build cache can be stored in Artifactory
- Reduces time required to build by up to 50%

The Result - Firmware

- Cuts the total time after first build to 5-10 minutes
- Build is as small as possible
- Updates are signed and secure
- Automatic rollbacks in case of failure

Success!

Other Tools

OSTree

Git for operating systems

OSTree

- Versions updates of Linux operating systems
- Git-like system with branching
- Tracks file system trees
- Allows for updates and rollbacks
- Exists as a meta-layer for Yocto

LAVA

Testing framework for operating systems on embedded devices

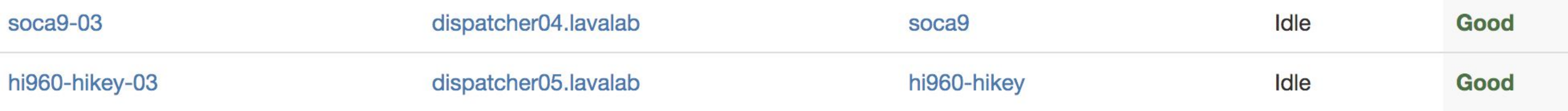
LAVA

- Linaro Automation and Validation Architecture
- CI system for deploying an OS to device for testing
- Can deploy to physical or virtual hardware
- Boot testing, bootloader testing, or system testing
- Results tracked over time

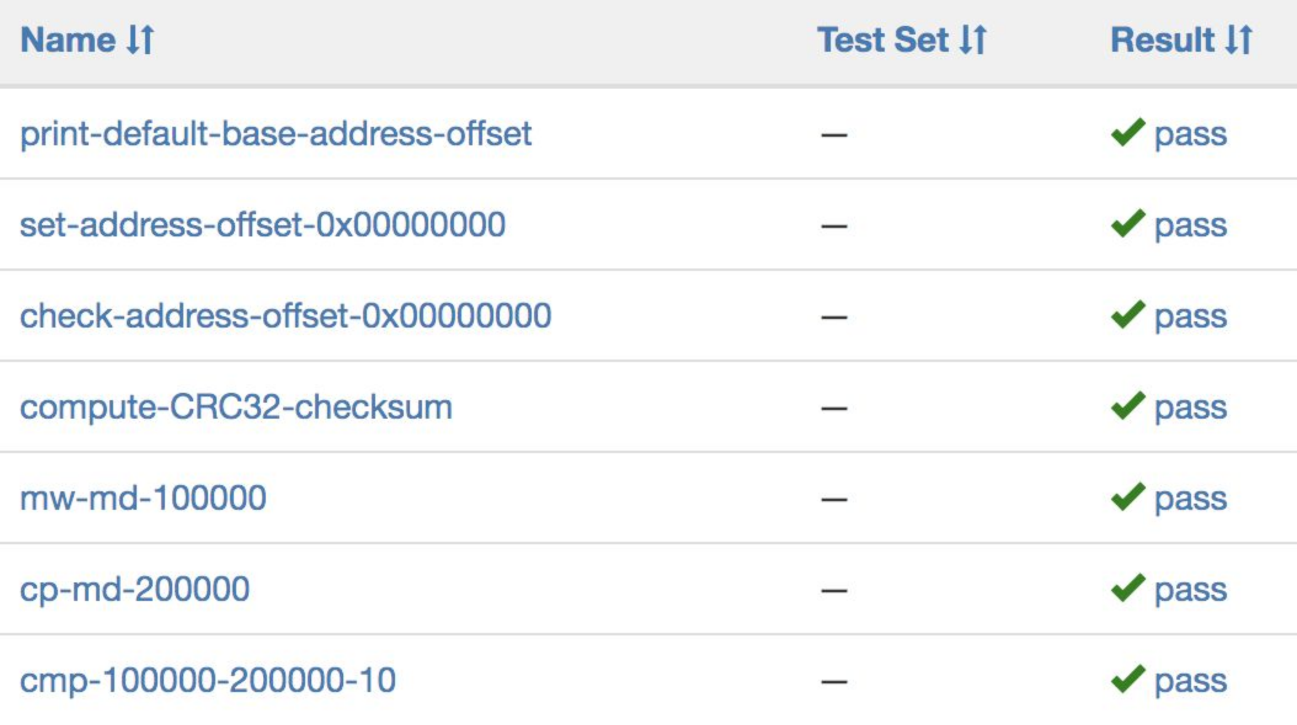
LAVA

- Designed for validation during development
- For example, whether the kernel compiles and boots
- Templates for more than 100 boards built in
- Custom devices types can be added

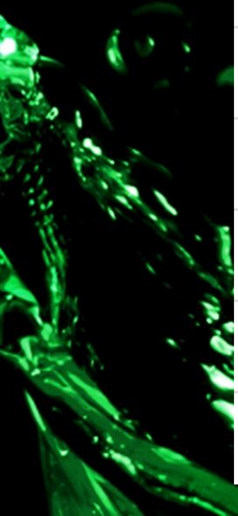
LAVA Tests



soca9-03	dispatcher04.lavalab	soca9	Idle	**Good**
hi960-hikey-03	dispatcher05.lavalab	hi960-hikey	Idle	**Good**



Name ⇅	**Test Set ⇅**	**Result ⇅**
print-default-base-address-offset	—	✓ pass
set-address-offset-0x00000000	—	✓ pass
check-address-offset-0x00000000	—	✓ pass
compute-CRC32-checksum	—	✓ pass
mw-md-100000	—	✓ pass
cp-md-200000	—	✓ pass
cmp-100000-200000-10	—	✓ pass



Wrapping Up

Edge and IoT updates are broken

This is a security problem that must be addressed

Modern DevOps tools are here to help

JFrog ENTERPRISE +

yocto PROJECT

MENDER

HELM

K3S

LAVA

The Fun Part

Overengineering a Toy

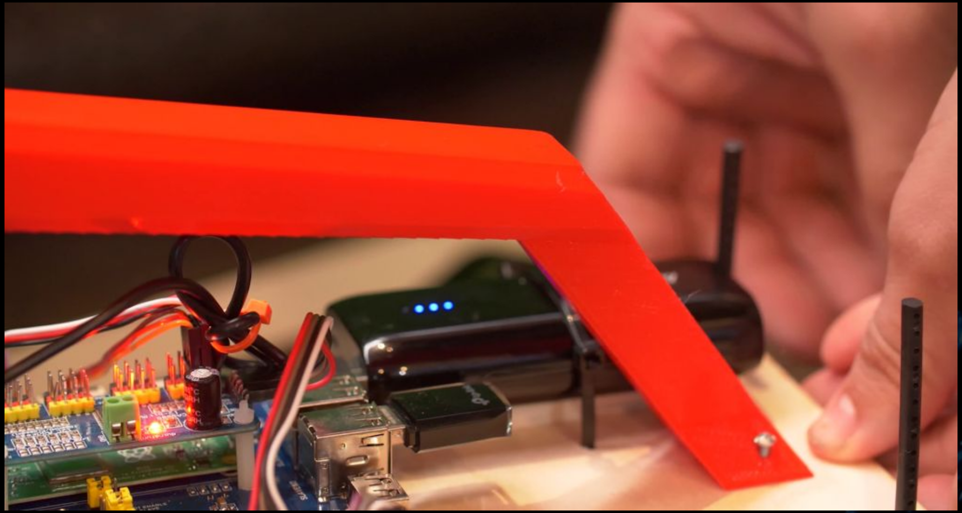
So, what’s this demo?

WANT TO DEV OR DRIVE?
Google Cloud
the next big thing.
circleci

A basic, self-driving miniature car

Donkey Cars

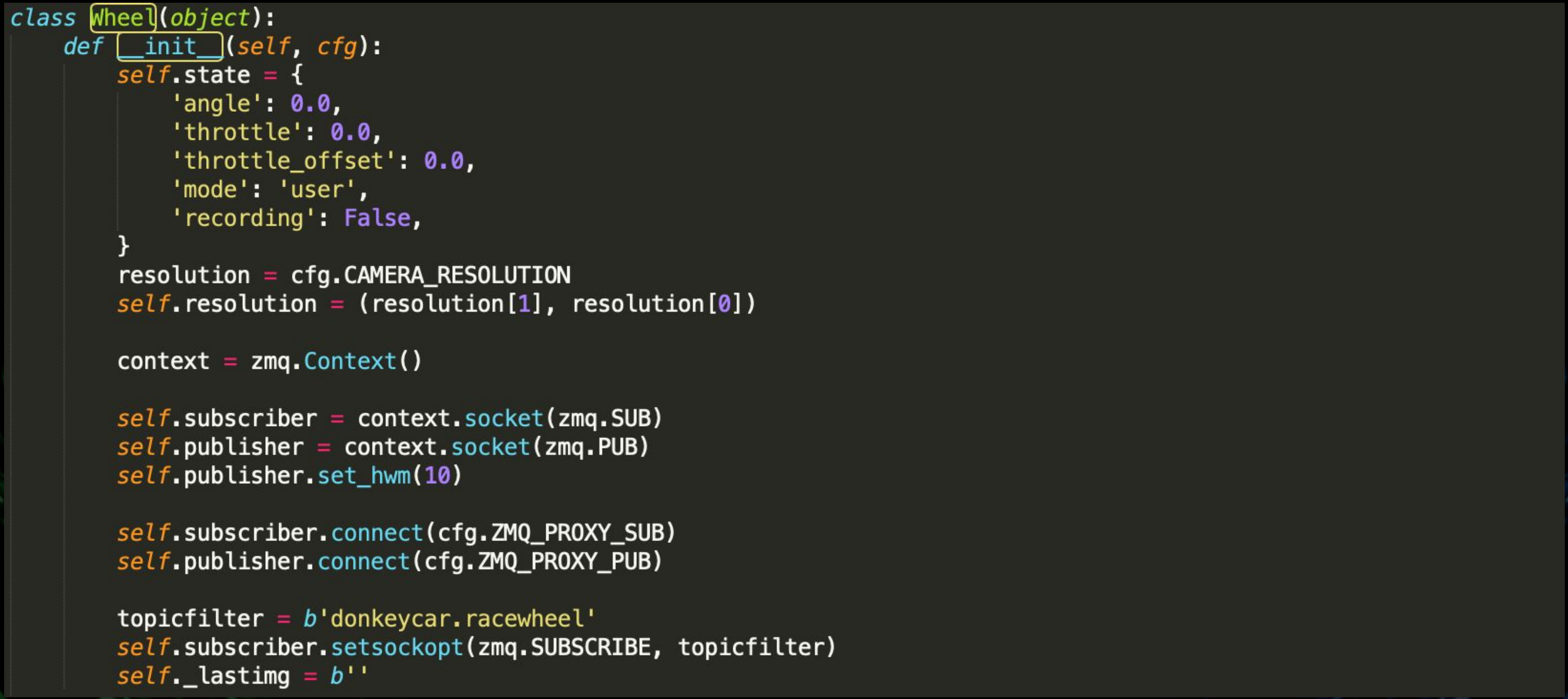
- About $250
- R/C Car
- Raspberry Pi 3B
- Pi Camera
- Race them!



Now make it cooler

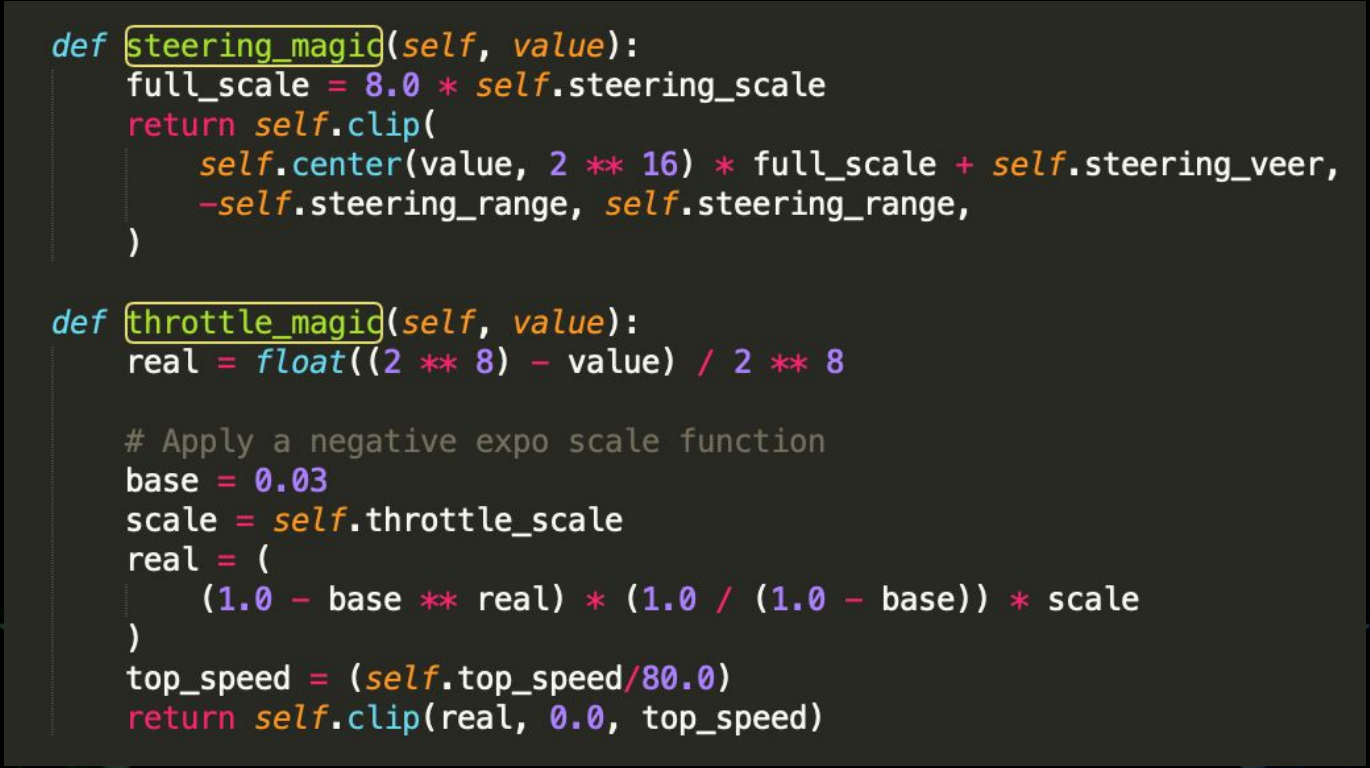
- Control it with a USB race wheel + pedals instead
- Automate training new models
- Move camera feed to Driver's screen
- Add a green screen for some flair

Swapping the Controls



```
class Wheel(object):
    def __init__(self, cfg):
        self.state = {
            'angle': 0.0,
            'throttle': 0.0,
            'throttle_offset': 0.0,
            'mode': 'user',
            'recording': False,
        }
        resolution = cfg.CAMERA_RESOLUTION
        self.resolution = (resolution[1], resolution[0])

        context = zmq.Context()

        self.subscriber = context.socket(zmq.SUB)
        self.publisher = context.socket(zmq.PUB)
        self.publisher.set_hwm(10)

        self.subscriber.connect(cfg.ZMQ_PROXY_SUB)
        self.publisher.connect(cfg.ZMQ_PROXY_PUB)

        topicfilter = b'donkeycar.racewheel'
        self.subscriber.setsockopt(zmq.SUBSCRIBE, topicfilter)
        self._lastimg = b''
```

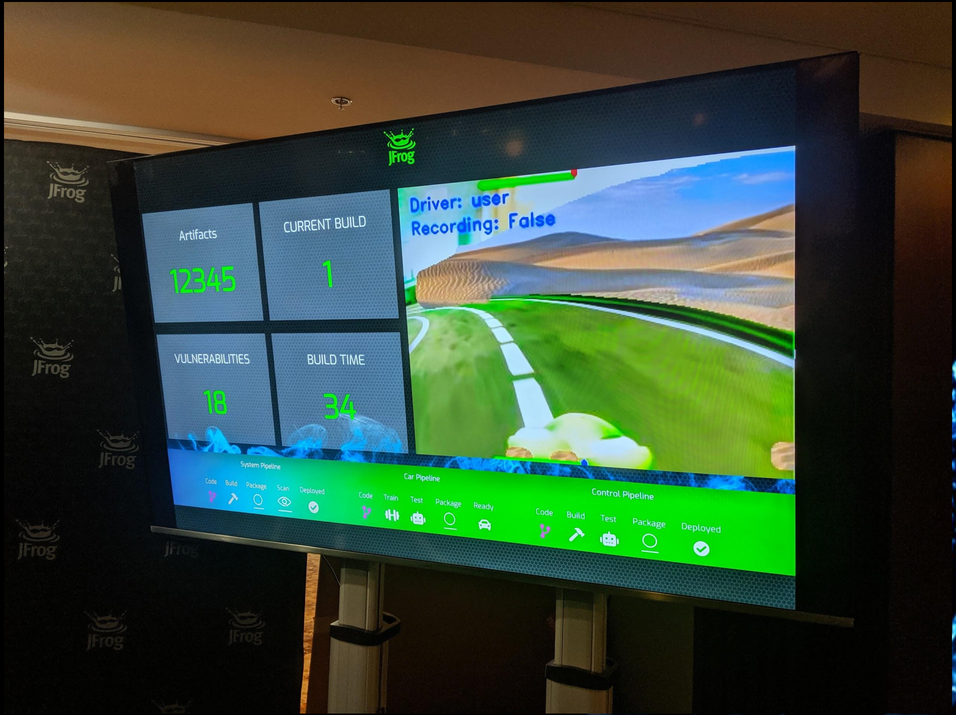
Magic



```python
def steering_magic(self, value):
    full_scale = 8.0 * self.steering_scale
    return self.clip(
        self.center(value, 2 ** 16) * full_scale + self.steering_veer,
        -self.steering_range, self.steering_range,
    )

def throttle_magic(self, value):
    real = float((2 ** 8) - value) / 2 ** 8

    # Apply a negative expo scale function
    base = 0.03
    scale = self.throttle_scale
    real = (
        (1.0 - base ** real) * (1.0 / (1.0 - base)) * scale
    )
    top_speed = (self.top_speed/80.0)
    return self.clip(real, 0.0, top_speed)
```

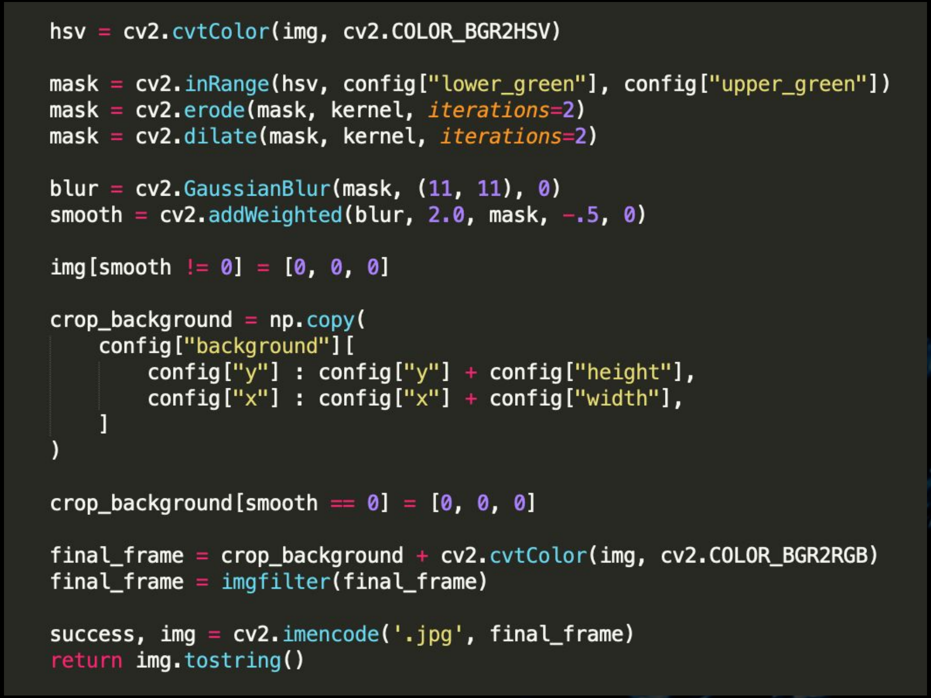
The Driver's Seat

- Managed by Intel NUC
- Sanic Webserver
- VueJS front-end
- ZMQ proxy
  - CI/CD
  - Image feed
  - Racewheel data



Writing a Green Screen

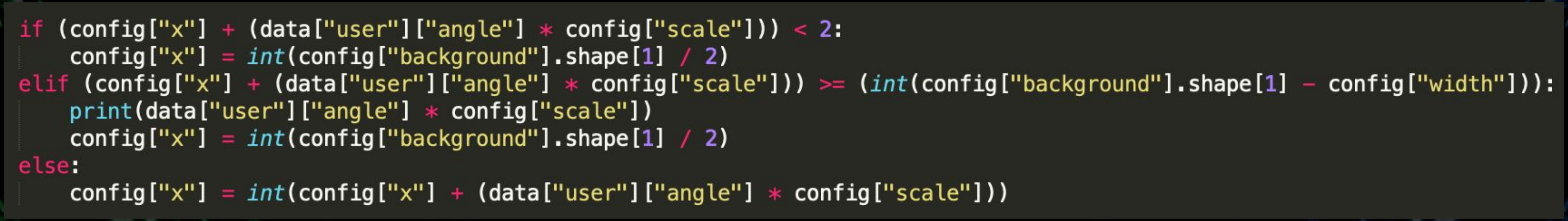
- Read frame with OpenCV
- Convert to HSV
- Set HSV range
- Create mask from range
- Crop background
- Merge them



```
hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

mask = cv2.inRange(hsv, config["lower_green"], config["upper_green"])
mask = cv2.erode(mask, kernel, iterations=2)
mask = cv2.dilate(mask, kernel, iterations=2)

blur = cv2.GaussianBlur(mask, (11, 11), 0)
smooth = cv2.addWeighted(blur, 2.0, mask, -.5, 0)

img[smooth != 0] = [0, 0, 0]

crop_background = np.copy(
    config["background"][
        config["y"] : config["y"] + config["height"],
        config["x"] : config["x"] + config["width"],
    ]
)

crop_background[smooth == 0] = [0, 0, 0]

final_frame = crop_background + cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
final_frame = imgfilter(final_frame)

success, img = cv2.imencode('.jpg', final_frame)
return img.tostring()
```

Green Screen, but better

- Actually really disorienting with a static background
- Angle of steering used to calculate how far to move crop position vs previous frame
- Scale variable to change perceived speed of panning



```
if (config["x"] + (data["user"]["angle"] * config["scale"])) < 2:
    config["x"] = int(config["background"].shape[1] / 2)
elif (config["x"] + (data["user"]["angle"] * config["scale"])) >= (int(config["background"].shape[1] - config["width"])):
    print(data["user"]["angle"] * config["scale"])
    config["x"] = int(config["background"].shape[1] / 2)
else:
    config["x"] = int(config["x"] + (data["user"]["angle"] * config["scale"]))
```

Automating the Training

- ZMQ proxy already has images + steering data
- TubWriter utility on NUC processes the data into usable format
- Data passed up to TensorFlow on GCP for training
- Around 10 minutes to train new model and make it available for the driver
- Still slow, but faster and way easier than manual

Thank you!

@Dixie3Flatline jfrog.com/shownotes