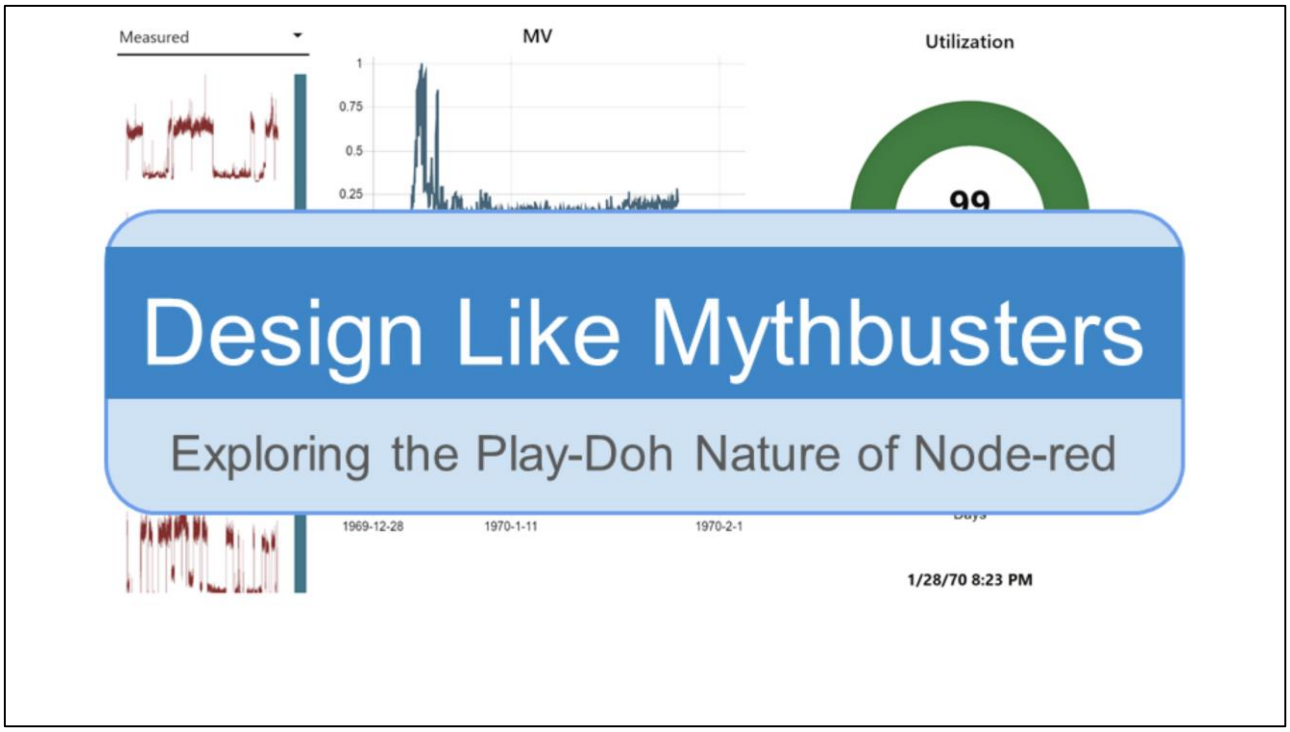

# Design Like Mythbusters

Exploring the Play-Doh Nature of Node-red

Hello everybody! Good to see you!

As developers we get to write software for many different areas of human endeavour. If you've been in the industry for any length of time you’ve probably seen your fair share of insurance companies, government offices, or academia.

While much of the time it seems to be lobbing CRUD or coding yet another front end designed by Gary in Accounting, occasionally we get a gem of a project.

I stumbled into such a gem which was complicated enough to be interesting.

The thing about complexity is that it should be just complicated enough and no more. Sometimes we can find ourselves wanting to dial-down the complexity so that we can get a better understanding of whatever the big complex thing is.

And that’s where these next two guys are going to help….

-mythbusters-

*Mythbusters* was a television show on the Discovery Channel where the hosts Adam Savage and Jamie Hyneman would use the scientific method to determine if various myths, urban legends, and movie scenes had any validity.

The myth would be their **hypothesis** and they would design **experiments** to classify the myth into “confirmed,” “plausible,” or “busted” categories.

Now *I* would have built a machine-learning system that uses Watson and Wikipedia to do the same thing but they actually built physical things and set them in motion.

# of myths: 1,015

- 548 busted
- 251 confirmed
- 216 plausible

-mythbusters-

You probably wouldn’t use their style of “scientific method” if you were at a nuclear research facility but for television it was pretty good. They spent a lot of time explaining what they were expecting to happen, how they were going to measure their experiment, and analyzing it all afterwards.

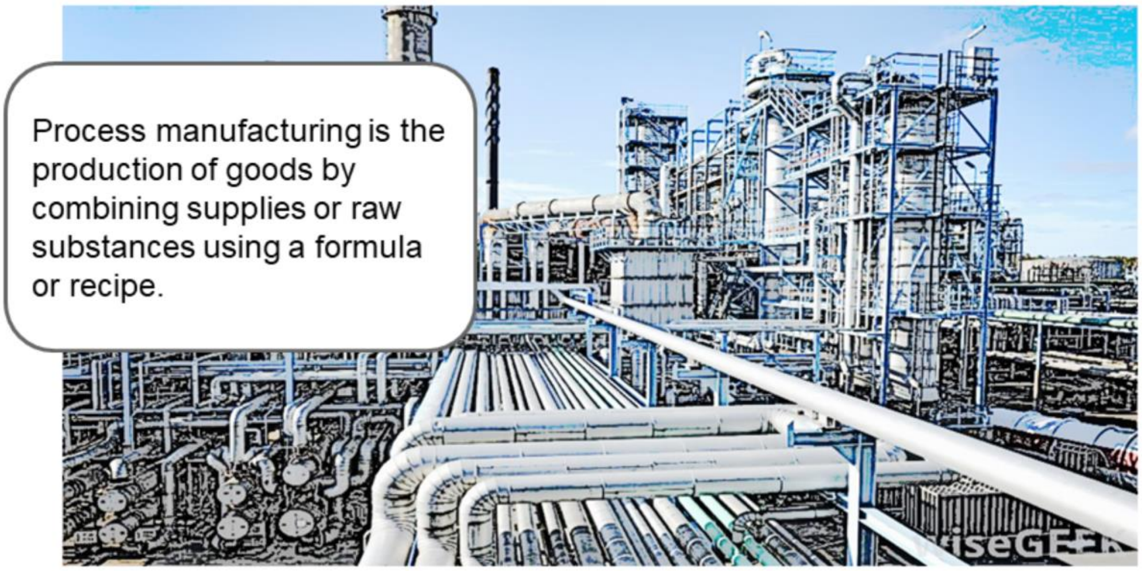
When they had an experiment that was complicated, expensive, or dangerous they would build a model.

- Duct tape stuck down: 83 miles
- # of experiments: 2,950
- # of vehicles destroyed: 295
- # of explosions: 900

-mythbusters-

They couldn’t shoot people so they would build a model of a human torso out of ballistic gel. They had a crash test dummy named “Buster” so that they wouldn’t have to jump out of an airplane using a life raft as a parachute. San Francisco real estate being what it is they built a shack in the desert to blow up with a hot water heater.

They used simple models to understand complicated systems.


Process manufacturing is the production of goods by combining supplies or raw substances using a formula or recipe.

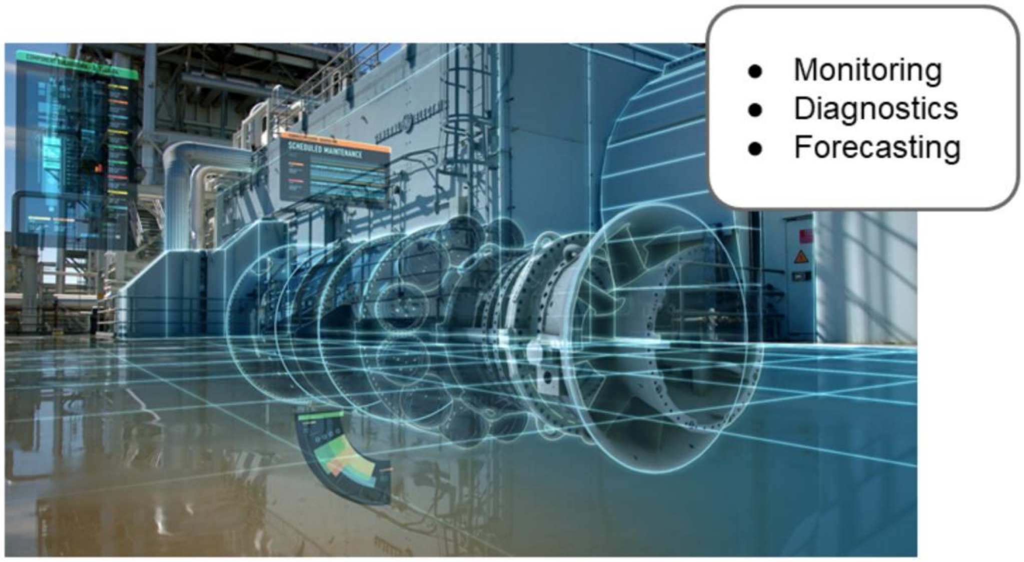
-process industries-

Two years ago at Stir Trek I attended a session on Azure stream analytics. My client at the time was in the process industries space creating monitoring and analysis systems for industrial automation.

*Process industries* is the class of manufacturing that works with transforming raw materials into finished products. Think paper mills or chemical refineries. The software took measurements from the processes and calculated key performance indicators for display to the on-site engineers.

The whole thing seemed like a batch processing quagmire. Don’t misunderstand me, that’s just the way that the industry has grown. Data transmission was expensive and there was a lot of it, the industry had to come up with standard protocols for collecting and transferring data. Once you had standards you got commercial products, all of which may or may not work together.

It looked like with stream processing we could simplify the flow and remove bottlenecks.



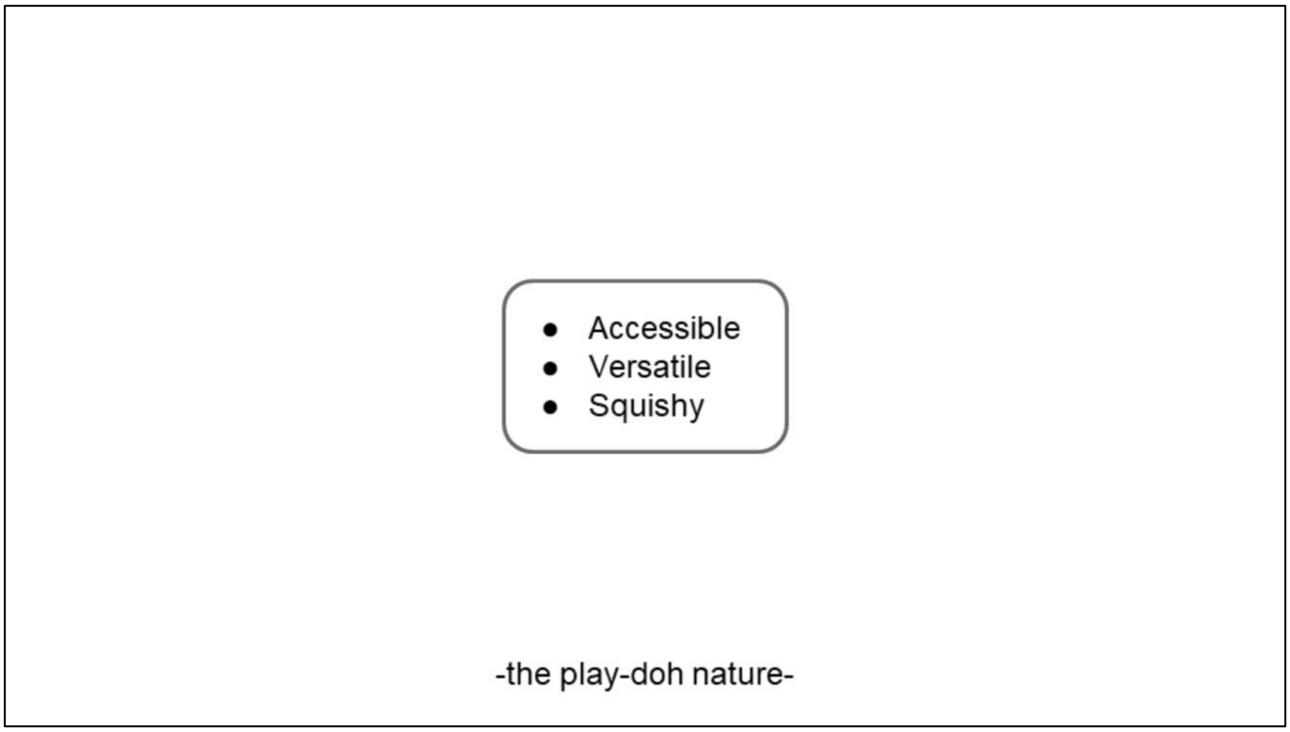
-process industries-

A big idea in industrial automation is that of the *digital twin*. All of the data about a physical piece of machinery is gathered together and presented as a whole. Not only operational characteristics like speed or temperature but structural characteristics like tolerances and dimensions. When Tony Stark waves his hands in the air to manipulate a 3D projection he is manipulating a digital twin.

There are three major pieces of functionality that you want your digital twin to provide:

- **M**onitoring, **D**iagnostics, and **F**orecasting

Monitoring tells you what the machine is doing now, diagnostics tells you if something’s wrong, and forecasting tells you what the machine is going to do in the future.



- Accessible
- Versatile
- Squishy

-the play-doh nature-

When building models you want to have a material that has the Play-Doh nature.

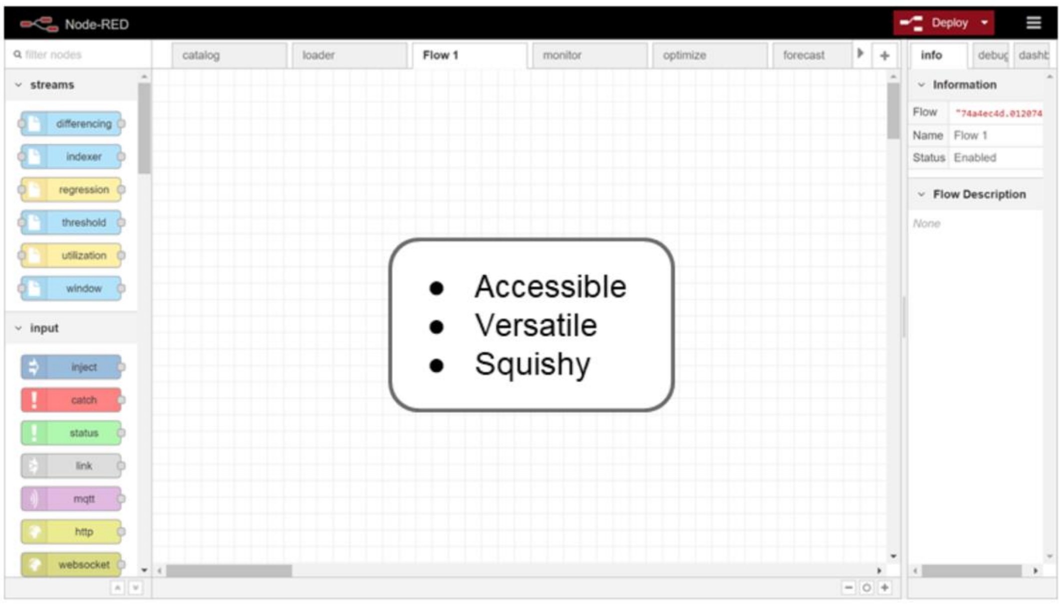
It should be

- **A**ccessible, **V**ersatile, and **S**quishy

**Accessible** means that everyone can relate to it. It should be non-threatening, inviting, friendly.

**Versatile** means that it can be used for many unrelated purposes. A zoologist can use Play-Doh to position small bones for photographing and a white-hat hacker can use it to unlock an IPhone.

**Squishy** means malleable, you can easily form it into what you envision.



- Accessible
- Versatile
- Squishy

-the play-doh nature-

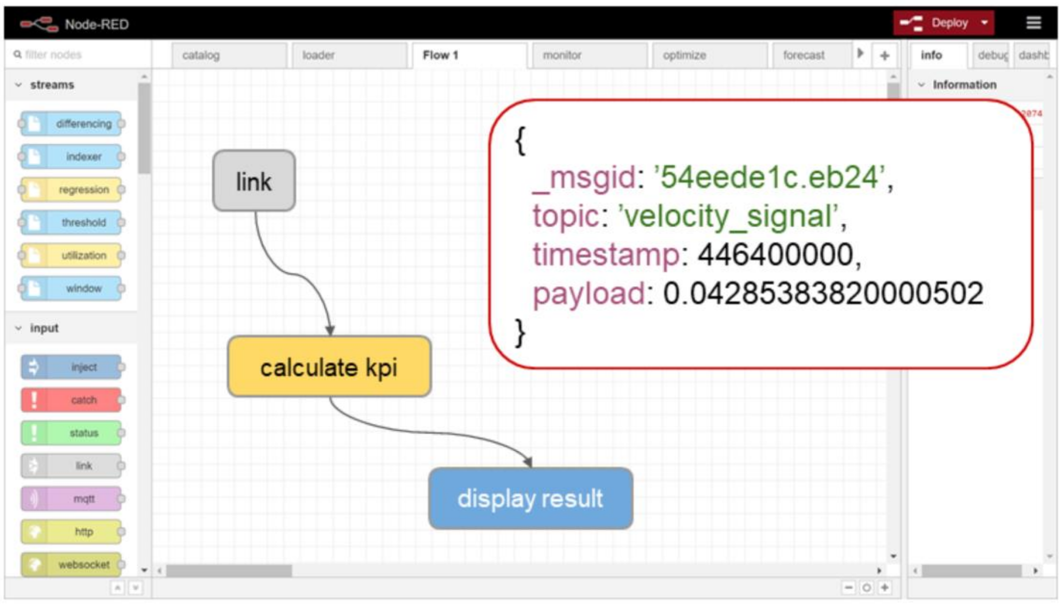
Node-red has the Play-Doh nature.

Node-red is a flow-based programming environment for the internet-of-things.

It is **accessible** because it uses javascript under the hood. I suspect that everybody here could pick up enough javascript in an afternoon.

It is **versatile**. People hook up all manner of things using node-red. I've used it to expose a web API for my rover bot and I'll use it for my next project which is a either a garage door opener or a furnace humidifier controller.

Finally, node-red has the all-important **squishy** property. If you find that you need a specialized node and you can’t find one in the npm registry, you can use javascript to build whatever you need.



```
{
  _msgid: '54eede1c.eb24',
  topic: 'velocity_signal',
  timestamp: 446400000,
  payload: 0.04285383820000502
}
```

-the play-doh nature-

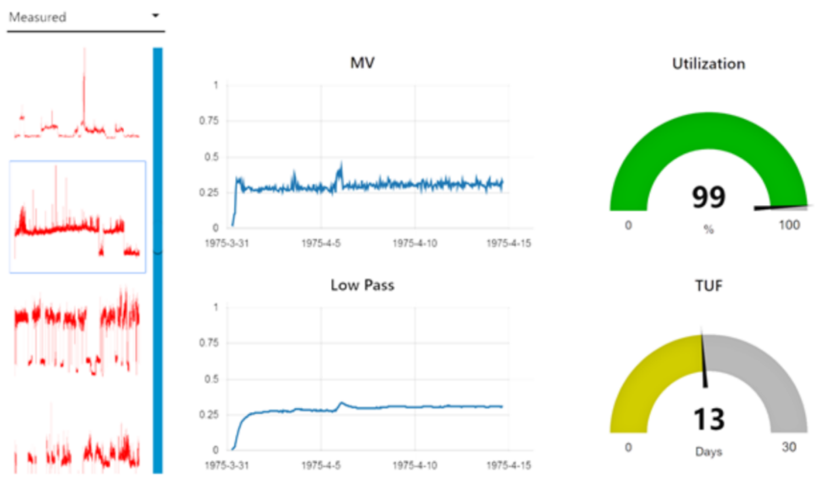
To better understand stream analytics I built a model of a digital twin.

In node-red you drop nodes from the toolbox onto the design surface, set properties of the nodes, and wire the nodes together.

Messages flow from node to node where each node does some processing. Nodes can create new messages. Nodes can read hardware signals. Nodes can raise signals and display information on-screen.

A typical message will have have a **message ID** which is assigned by the runtime, a **topic** property, and a **payload** property. Nodes can modify messages, clone messages, or create brand new messages.

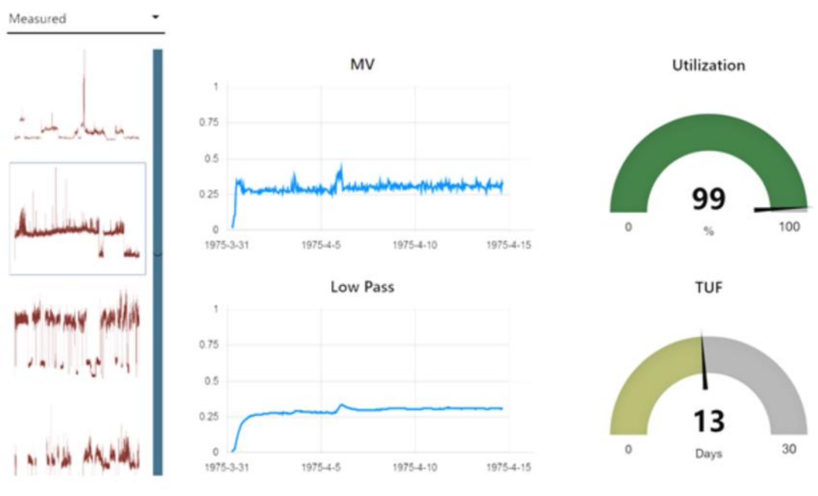
We’ll take a stream of time-stamped values, do some calculations, and display the result.



-digital twin-

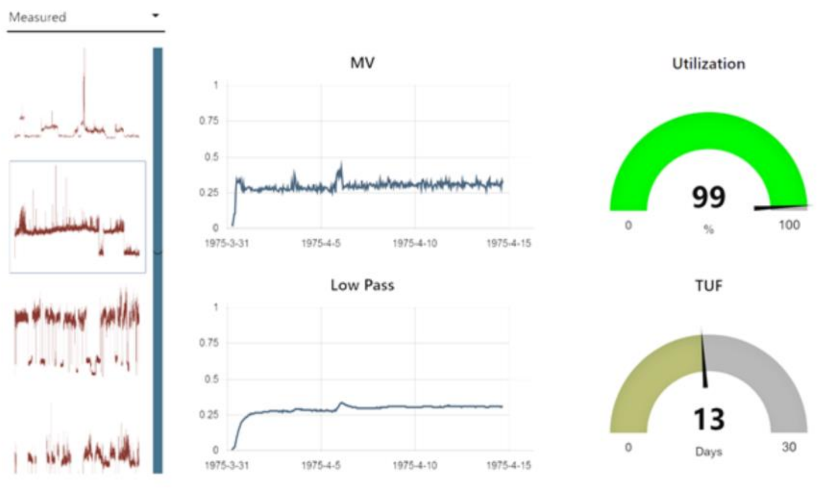
The physical device that I am interested in is a pump like you might see in a chemical plant. These pumps can have any number of sensors on them but since this is a model, I simplified that to one vibration sensor.

Vibration is measured as a velocity along some axis. Think of a washing machine when it’s out of balance. First it goes this way. Then it goes that way. That back-and-forth is vibration and its measure is velocity.



-digital twin-

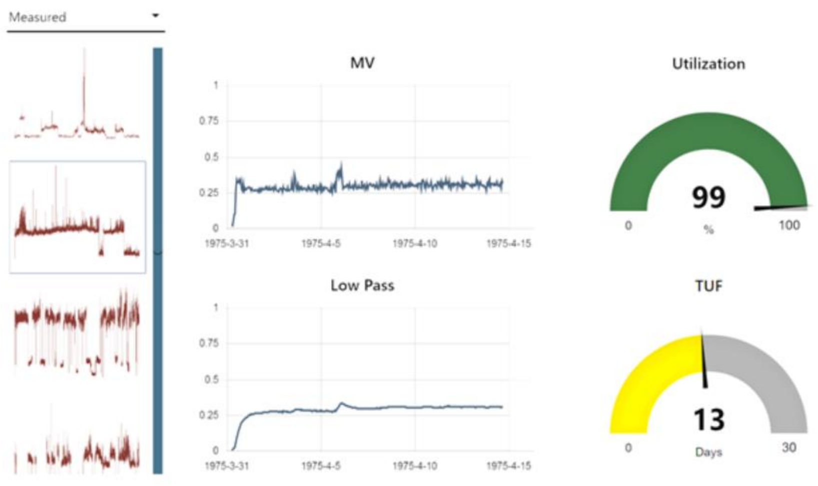
The **monitoring** section of this model  is composed of two charts. The top chart labeled **mv** for *measured value* is the raw vibration signal coming from a pump. The second chart labeled **low pass** is the same signal after it has had the high-frequency components removed by a low-pass filter.



-digital twin-

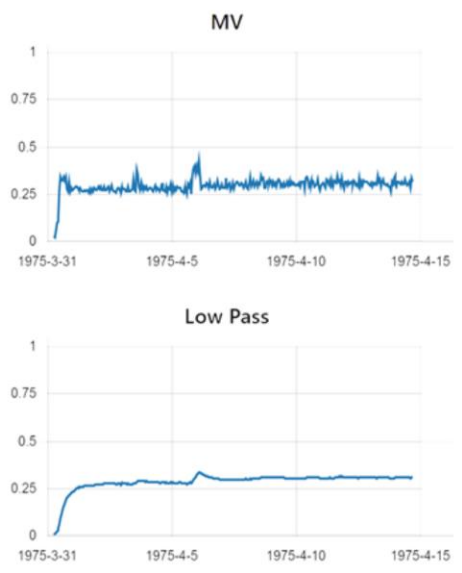
The **diagnostics** section consists of a gauge which displays **utilization** which is the proportion of the time that the pump is in operation. If it’s been on for 8 out of the last 10 days then the utilization is 80%.

Some manufacturers of these pumps provide signals from the pump which tell you the operational status of the pump but this one doesn’t so we’ll have to make do. For our purposes we’ll count the machine as being operational if the vibration signal is over 0.1.



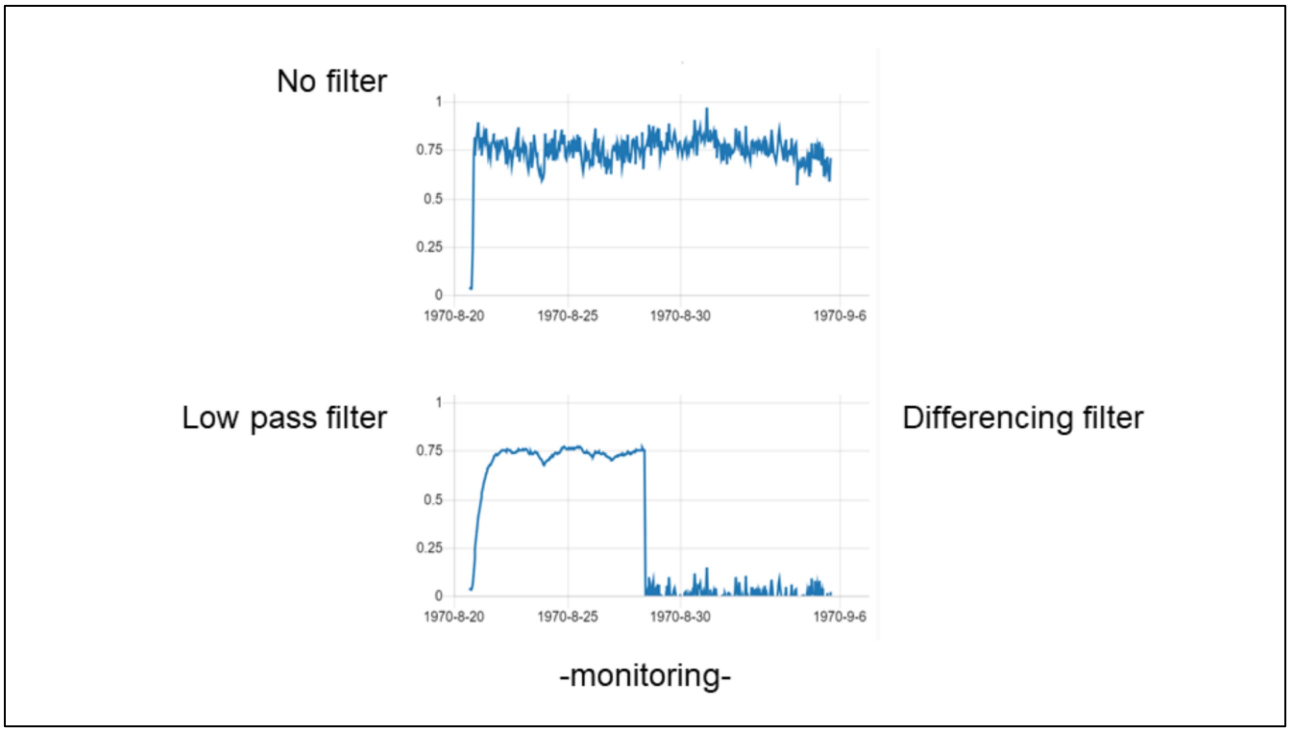
-digital twin-

And this is the **forecasting** section where we'll try to track the **time-until-failure** of the pump. Just like in the diagnostics section we'll pick an arbitrary number and say that if the vibration goes over that level we'll consider the pump to have failed.



-monitoring-

Let’s take a closer look at the **monitoring** section. This is where we would apply different digital-signal-processing algorithms to the stream of signal data and compare the processed stream to the raw stream.



One class of dsp algorithms are the **decomposition functions**.

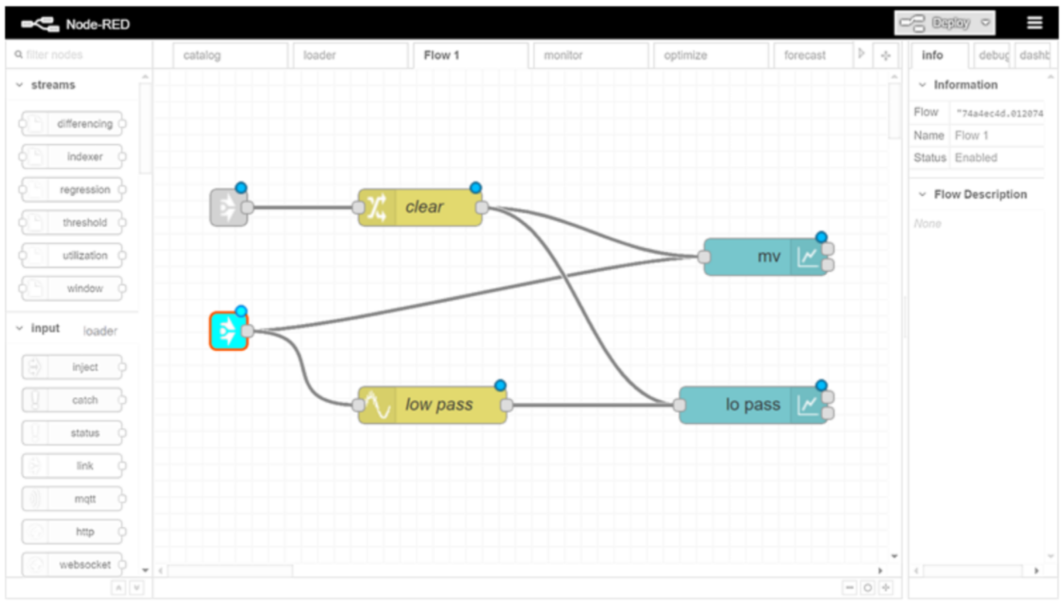
A digital signal can be decomposed into

- **Trend**, **seasonal**, and **residual** components

A low pass filter removes the seasonal and residual components leaving the trend.

A differencing filter removes the trend component and leaves the seasonal and residuals.

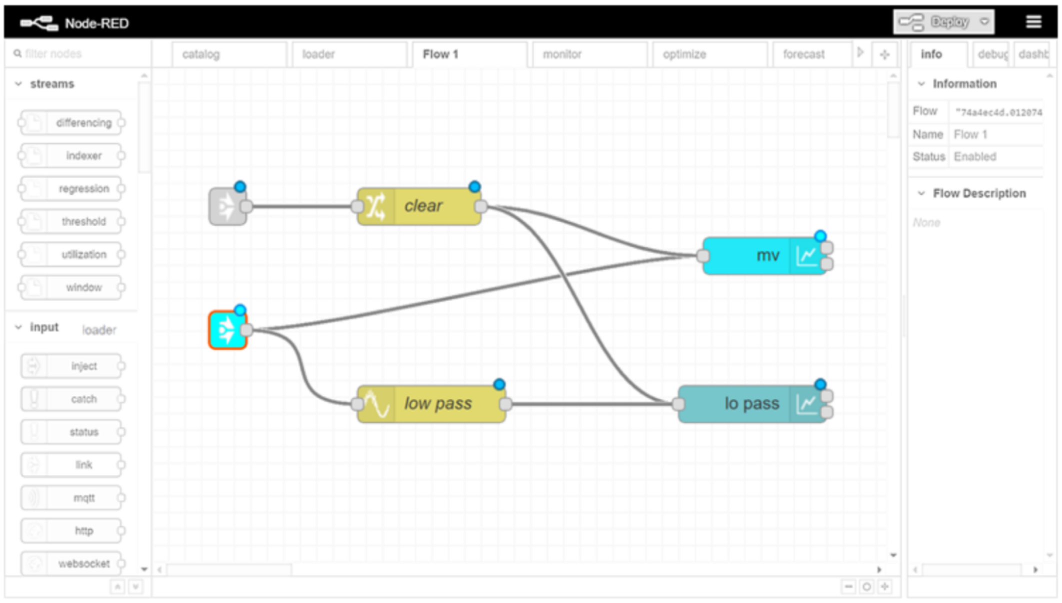
For this project we'll be mainly interested in the trend component.



-monitoring-

In node-red these tabs are called *flows* and are the basic unit of abstraction. Messages travel from node to node over the interconnecting *wires*, each node will do something with each message and pass it along to one or more downstream neighbors.

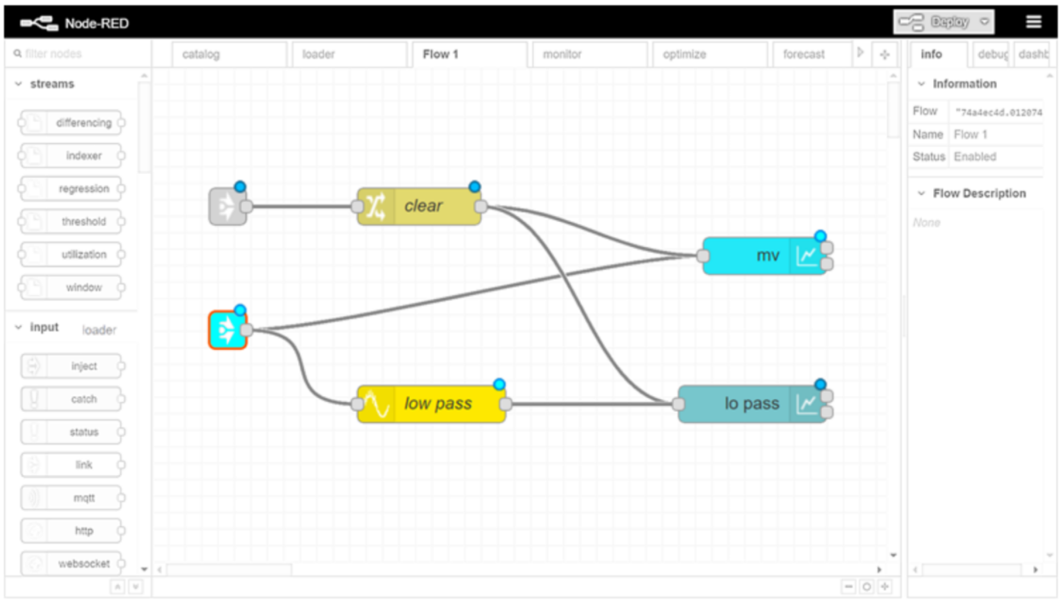
That bright blue blob on the left is a *link* node. Links are the mechanism by which we get messages from one flow to another. This one happens to be an input link, it is accepting messages from another flow.



-monitoring-

The message passes down the wire to this *graph* node which is provided by the node-red-dashboard package. In addition to line charts the node can be configured to display bar charts or pie charts.

This particular configuration is using the messages timestamp property for the x-value and the messages payload property for the y-value.

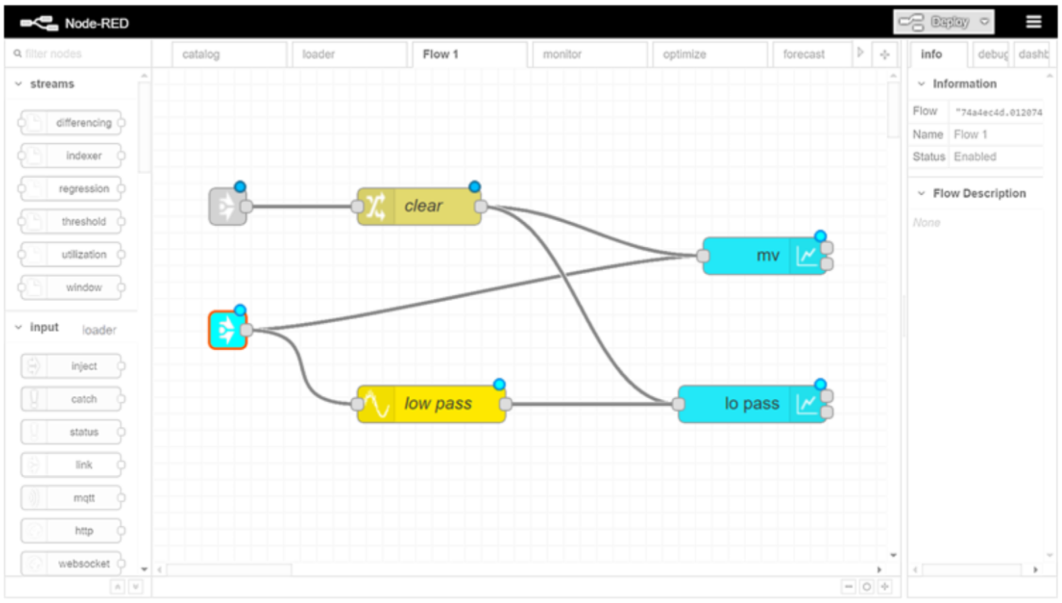


-monitoring-

The next node we'll look at is this one labeled "low pass." Notice that the link node passed the message to the mv node as well as this low pass node.

Node-red has multiple ways of sending messages to more than one downstream node. Messages can be duplicated like we see here or a message can be sent down different paths based on the value of a property on the message. Sometimes you might want to send the original node down one path and a modified node down another path.

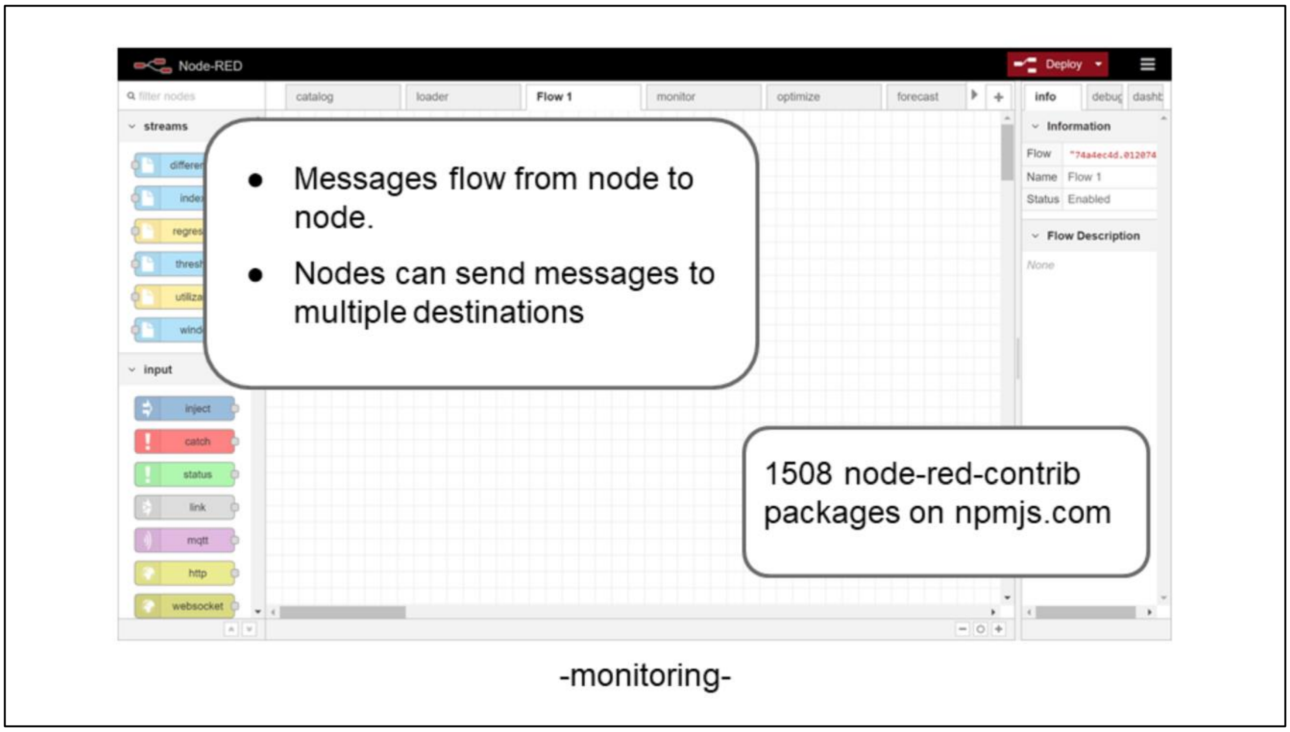
This low pass filter comes from the node-red-node-smooth package and can be configured to apply different smoothing algorithms to the inputs. That is really handy for this model because it allows us to play with the data and we can see the changes on the charts.



-monitoring-

Finally the low pass node sends the message to another chart node. This one is configured the same way as the mv chart and allows us to compare the measured value to the filtered value.

One of the neat things about the chart node … one of the many neat things … is that you can send two data streams to a single chart and the chart will plot both streams.



- Messages flow from node to node.
- Nodes can send messages to multiple destinations

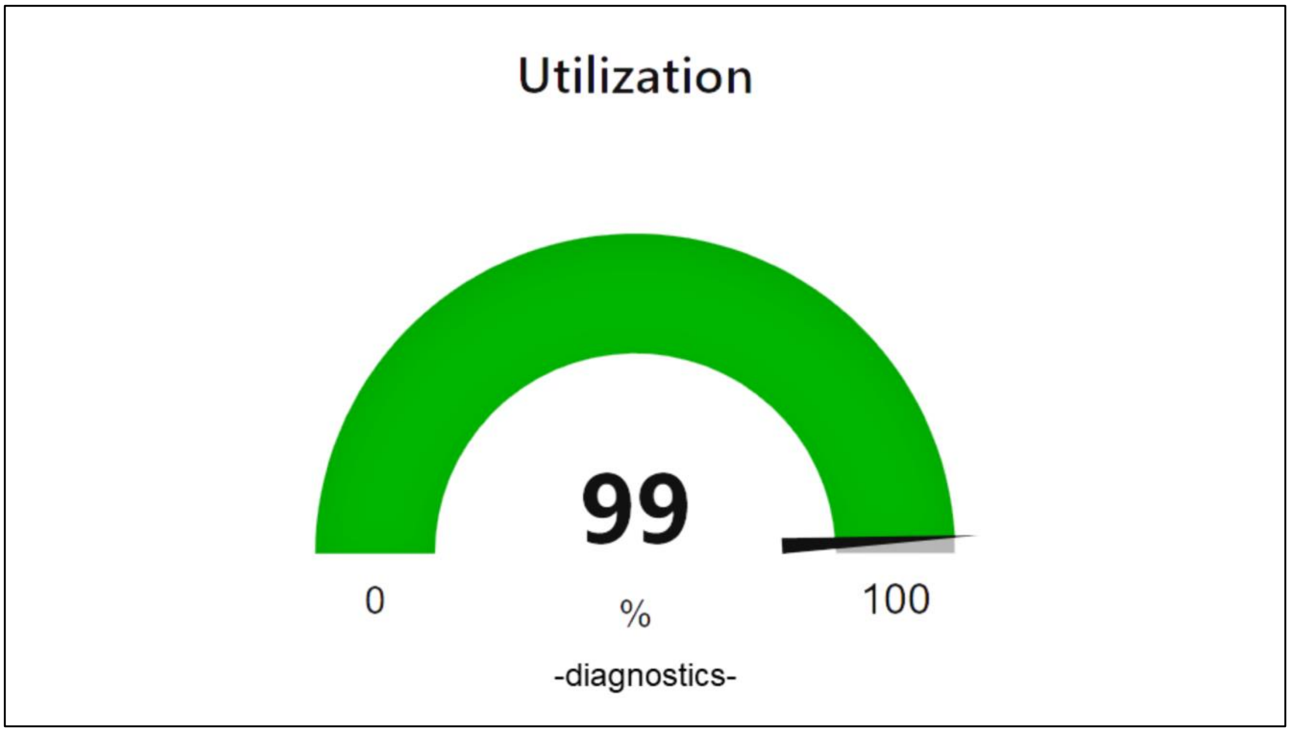
1508 node-red-contrib packages on npmjs.com

-monitoring-

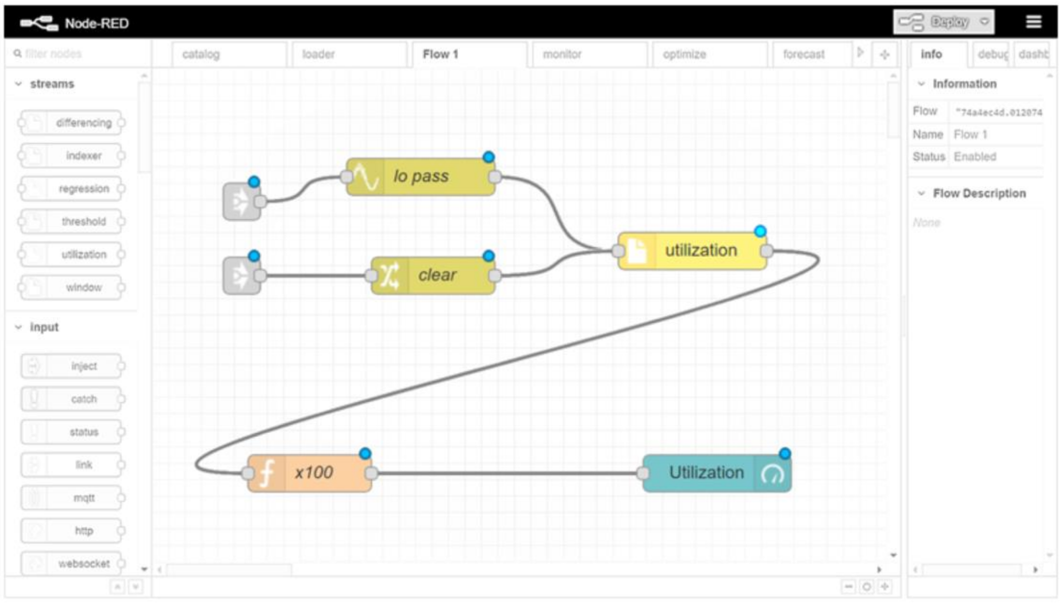
There are a couple of interesting things going on here.

The first is how messages flow from node to node but the more interesting thing is the user community around node-red. There are 1500 packages in the npm registry that have the node-red-contrib-* prefix.

The entire functionality of this part of the model was built with out-of-the-box nodes and a couple of packages from the registry.



In the diagnostics section we measure the utilization of the pump. Remember that we consider the pump to be running if the vibration signal is above 0.1. That 0.1 is totally arbitrary and reflects the fact that my fake data was normalized.



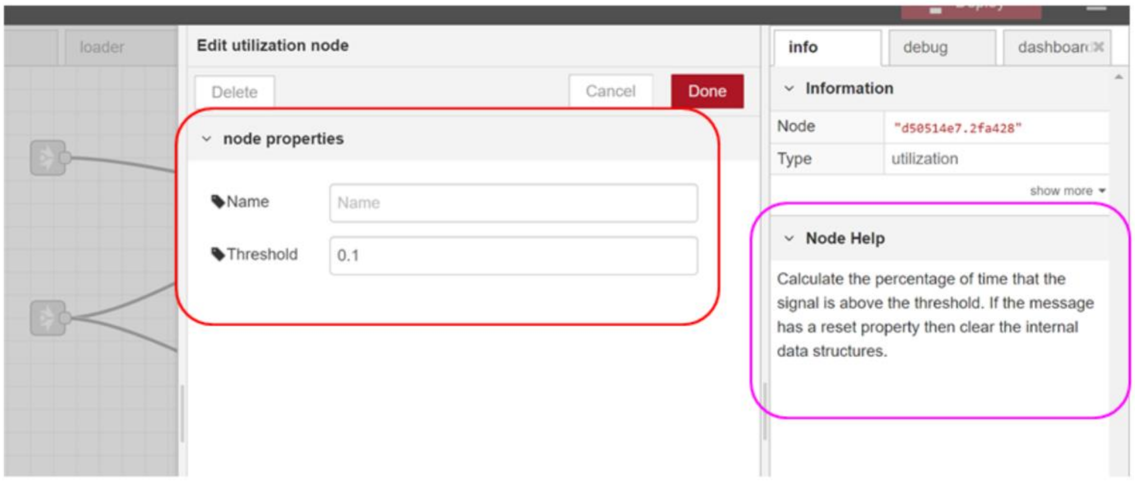
-diagnostics-

Here we see the low pass filter again and another dashboard node which renders the gauge.

The interesting node is that bright yellow one on the right. That is where we calculate the utilization from the stream of velocity samples entering the flow through those link nodes at the upper left.

That yellow utilization node is an example of a *user-defined node*, it is defined by two files - an html file and a javascript file. The html provides the user interface for the node configuration and the javascript  implements the functionality of the node.

We haven’t talked about configuration have we?
Now seems like a good time for a short detour.



node properties

| | |
|---|---|
| Name | Name |
| Threshold | 0.1 |

Information

| | |
|---|---|
| Node | "d50514e7.2fa428" |
| Type | utilization |

Node Help

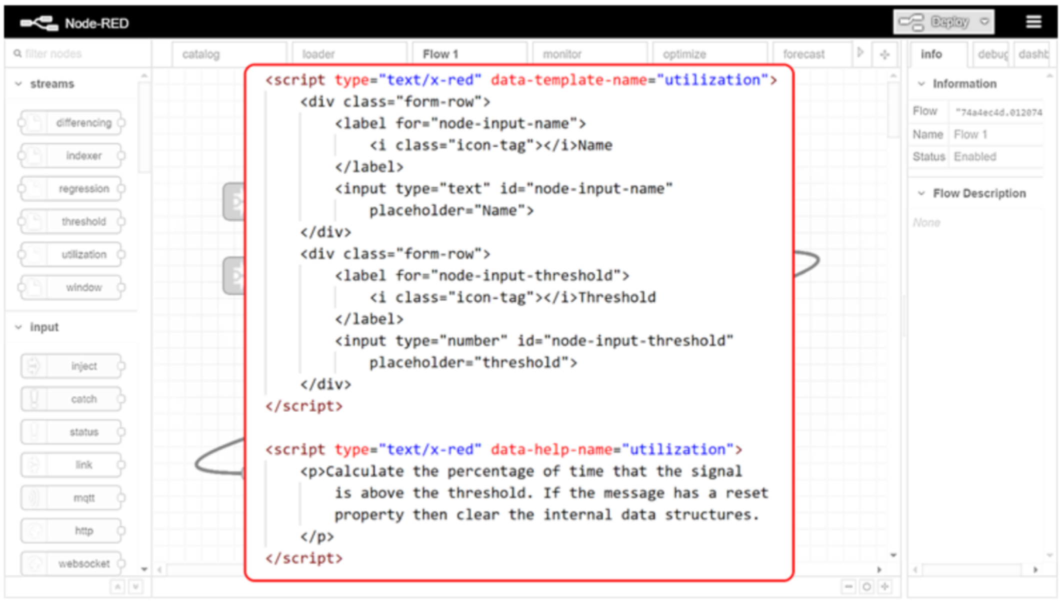
Calculate the percentage of time that the signal is above the threshold. If the message has a reset property then clear the internal data structures.

-diagnostics-

This is the configuration panel for the utilization node. This is where we can change that 0.1 we picked earlier to another value.

In the red box are the edit fields for the properties of the node. Like any proper web app the form has validation, default values, and placeholder text.

The purple box is the node help which tells the user what the node does. This node has an example of a fairly common special behavior - if the message has a property called “reset” then the node will clear any internal data structures and start over.



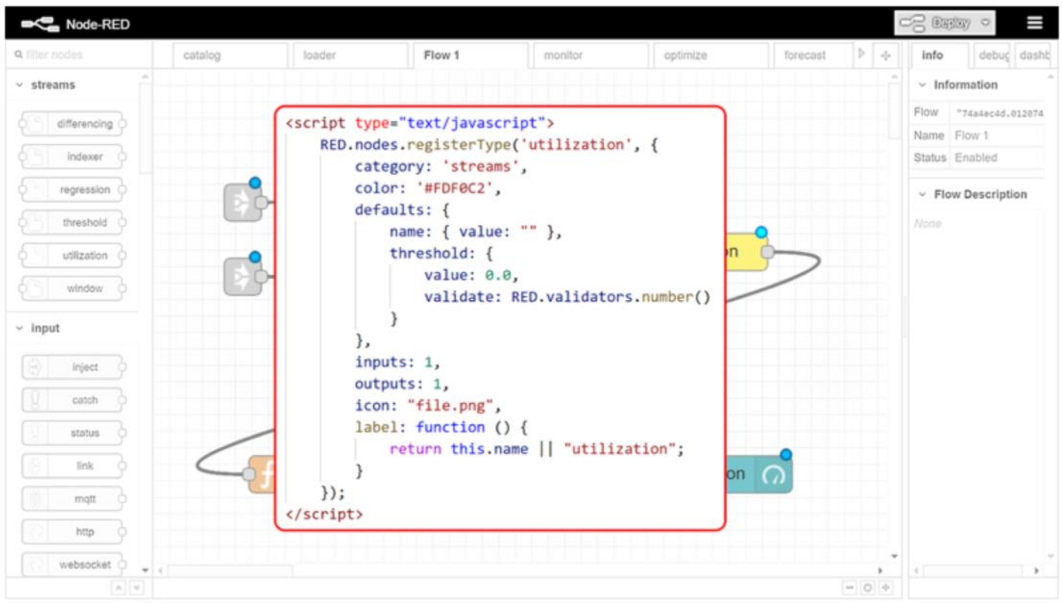
```
<script type="text/x-red" data-template-name="utilization">
    <div class="form-row">
        <label for="node-input-name">
            <i class="icon-tag"></i>Name
        </label>
        <input type="text" id="node-input-name"
            placeholder="Name">
    </div>
    <div class="form-row">
        <label for="node-input-threshold">
            <i class="icon-tag"></i>Threshold
        </label>
        <input type="number" id="node-input-threshold"
            placeholder="threshold">
    </div>
</script>

<script type="text/x-red" data-help-name="utilization">
    <p>Calculate the percentage of time that the signal
        is above the threshold. If the message has a reset
        property then clear the internal data structures.
    </p>
</script>
```

-diagnostics-

User-defined nodes have an html file and a javascript file.

This html fragment defines the form fields and the help text. If you've spent any time at all with html then the divs and inputs won't be very exciting.

What *is* exciting is the **type** attribute on the **script** tag.Props to anyone who needs to define their own MIME type.



```
<script type="text/javascript">
    RED.nodes.registerType('utilization', {
        category: 'streams',
        color: '#FDF0C2',
        defaults: {
            name: { value: "" },
            threshold: {
                value: 0.0,
                validate: RED.validators.number()
            }
        },
        inputs: 1,
        outputs: 1,
        icon: "file.png",
        label: function () {
            return this.name || "utilization";
        }
    });
</script>
```

-diagnostics-

This is the other part of the html file, notice that the script tag just has a boring old javascript MIME type.

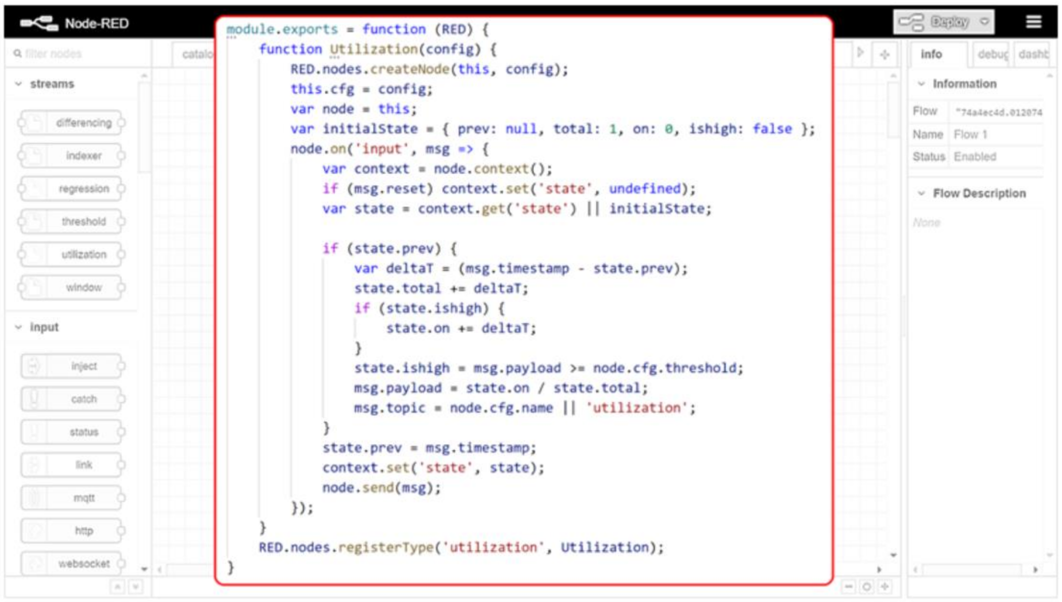
This fragment sets up visuals of the node like color and icon as well as default values for the edit fields.

Node-red provides two validators, one for numbers and the other is for strings and uses regular expressions.

If you have a node that has some very specific editing needs you can define your own validators and hook into the edit life cycle with callbacks.

At this point the configuration editor is defined but what is this node going to actually **do**?

For that we need ...



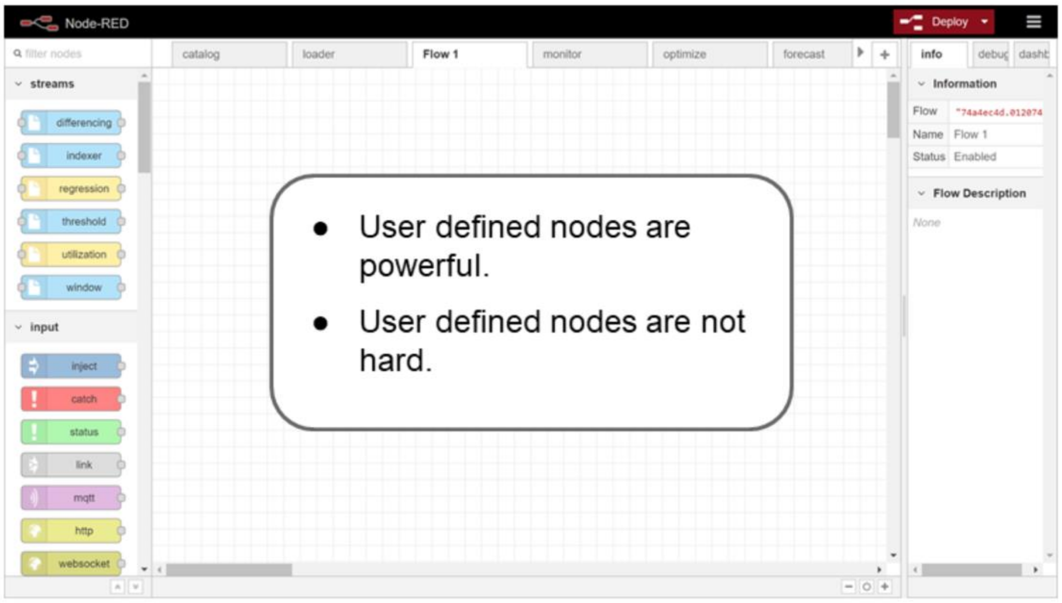
```
module.exports = function (RED) {
    function Utilization(config) {
        RED.nodes.createNode(this, config);
        this.cfg = config;
        var node = this;
        var initialState = { prev: null, total: 1, on: 0, ishigh: false };
        node.on('input', msg => {
            var context = node.context();
            if (msg.reset) context.set('state', undefined);
            var state = context.get('state') || initialState;

            if (state.prev) {
                var deltaT = (msg.timestamp - state.prev);
                state.total += deltaT;
                if (state.ishigh) {
                    state.on += deltaT;
                }
                state.ishigh = msg.payload >= node.cfg.threshold;
                msg.payload = state.on / state.total;
                msg.topic = node.cfg.name || 'utilization';
            }
            state.prev = msg.timestamp;
            context.set('state', state);
            node.send(msg);
        });
    }
    RED.nodes.registerType('utilization', Utilization);
}
```

-diagnostics-

… a little more javascript. I'm beginning to think of javascript as the duct tape of the internet.

The first thing of interest here is the **Utilization** function, this is a factory that creates the node and sets up an event listener for the **input** event. This factory gets registered with the run-time in that last line.

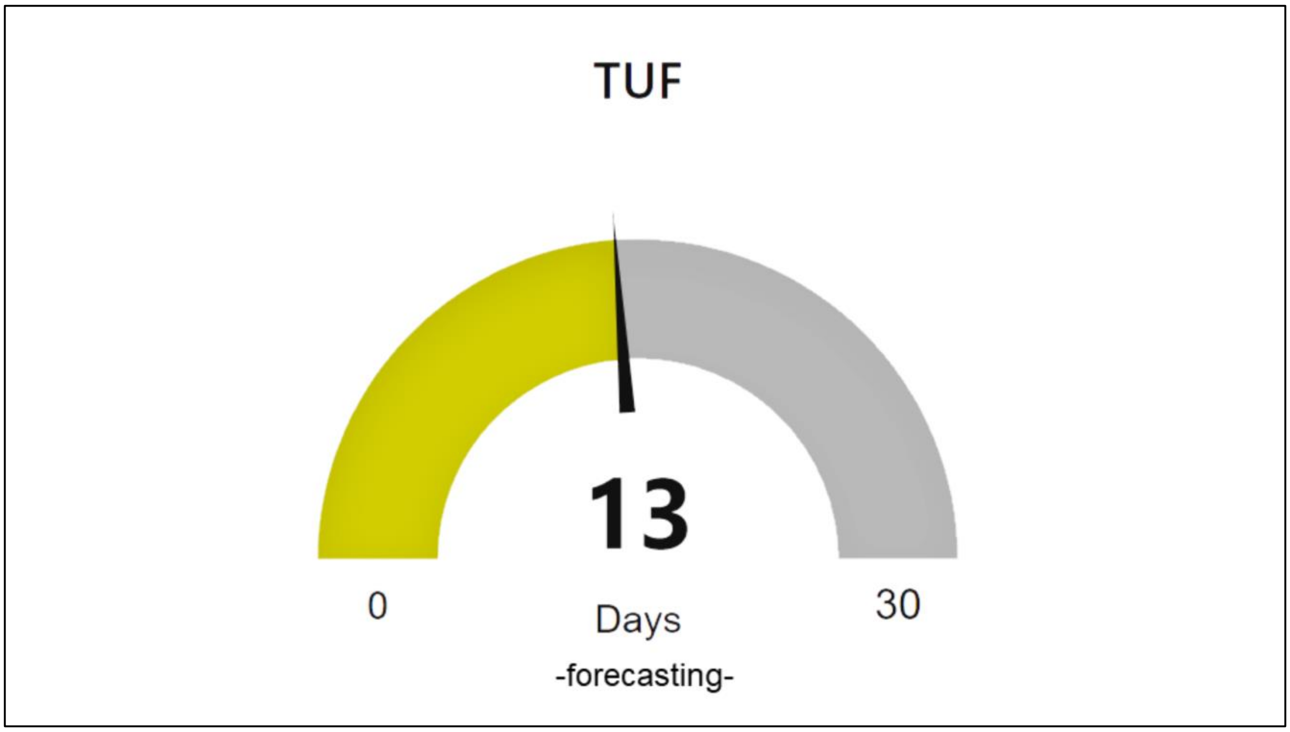
The other thing of interest is the event handler - this is where the node does its work. In this case it keeps track of the sum of the time differences between messages and the sum of the time differences where the value of the message was greater than the configured threshold. With each message that comes in to the node it updates the sums, calculates **on** time divided by **total** time, and sends the message downstream.



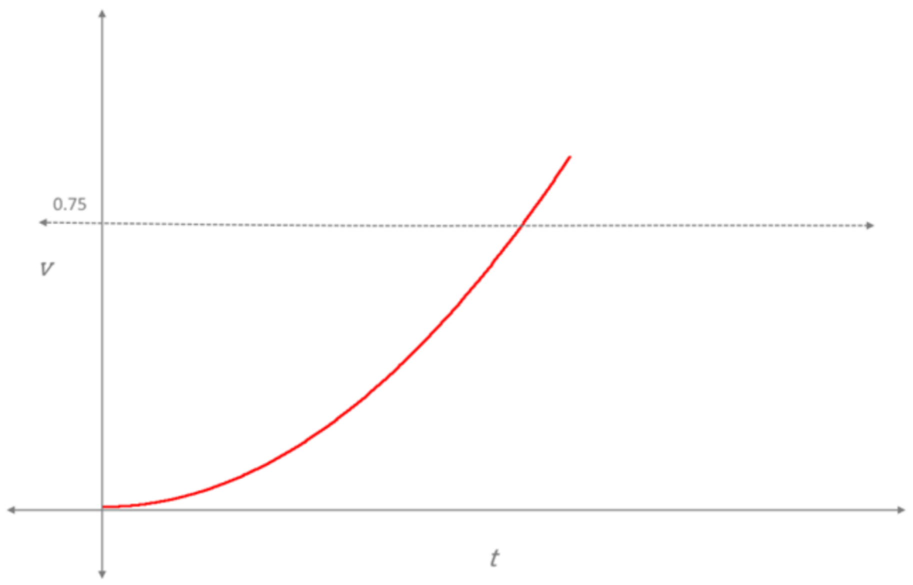
- User defined nodes are powerful.
- User defined nodes are not hard.

-diagnostics-

User-defined nodes are easier than it looked from those last few slides. They are a powerful tool if you are building something that requires a lot of instances of the same node with potentially different configuration values for each instance.

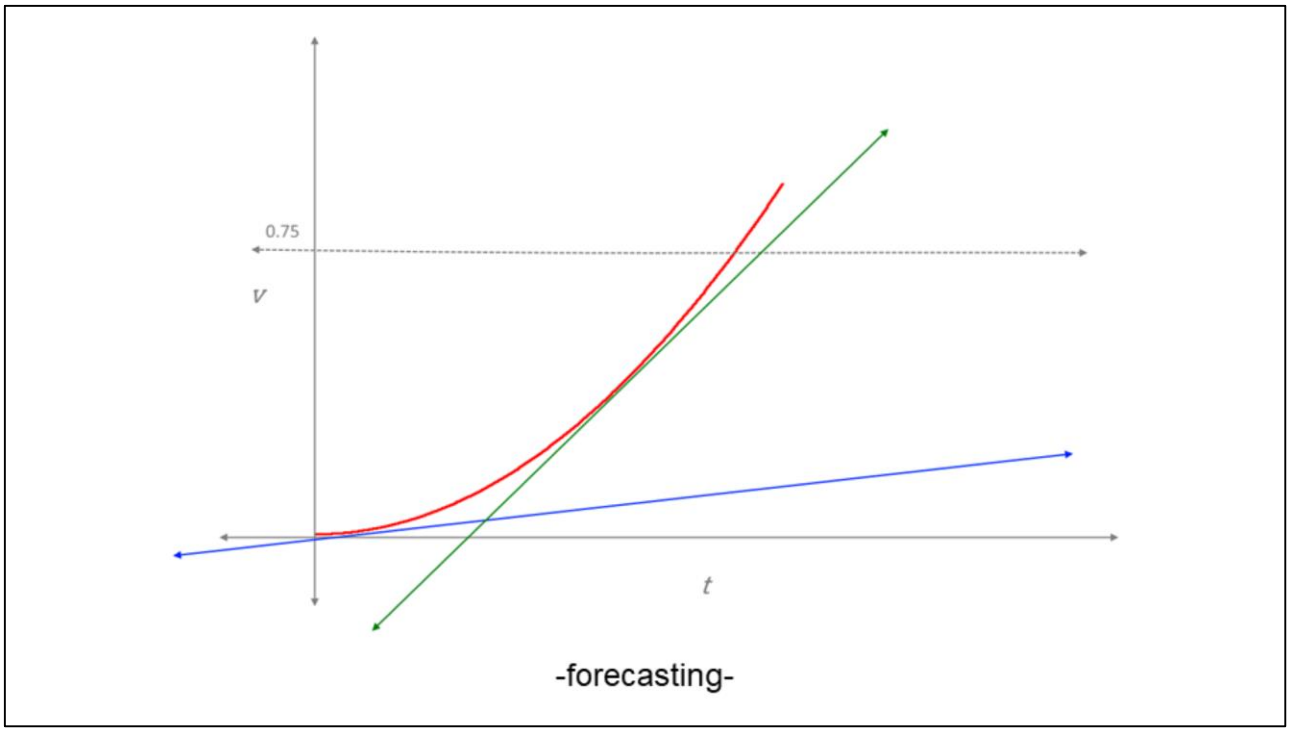


In this section we want to forecast the **time-until-failure** for the pump. Knowing when something is going to break allows you to plan ahead and replace the asset on *your* schedule and minimize downtime, costs,  and lost revenue.



-forecasting-

Just like in the utilization kpi we'll pick an arbitrary threshold to indicate failure. Let's define failure as the point where the vibration signal is greater than 0.75. We want to know when the red trend line crosses over the threshold. The tricky part is that we want to know before it happens, preferably days before it happens.

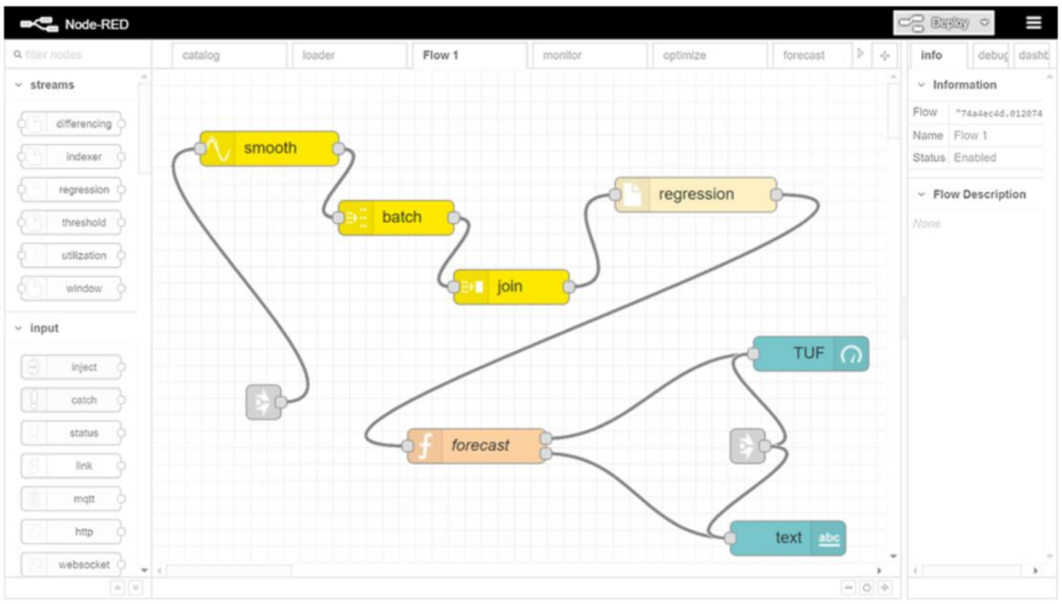


-forecasting-

We’re going to do that by using a regression model, setting *v* to 0.75 and then solving for *t*. That should tell us when the pump will fail. When the vibration isn’t getting any worse the blue line won’t cross the threshold for a quite while.

When the vibration starts to change more dramatically you can see how the green line will cross the threshold a lot sooner.

I know that there is someone in the audience thinking “That looks like a parabola. Just do a 2nd order regression and be done with it!” You are absolutely correct. However, this is fake-fake data meant to help visualize the kpi calculations. We’ll look at some real-fake data later.

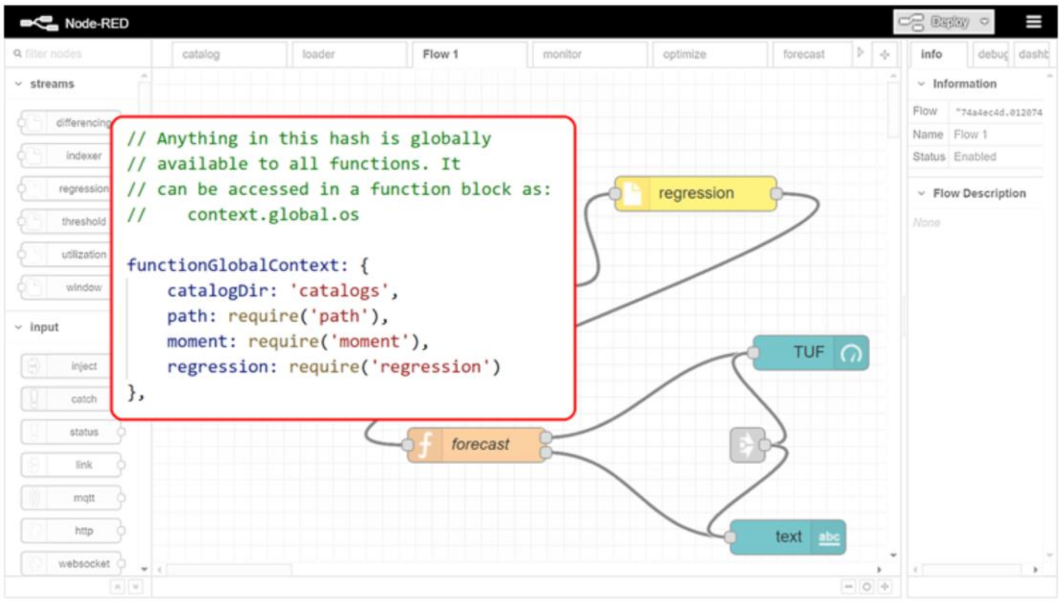


-forecasting-

The three yellow nodes on the left are doing some processing of the data stream to get it ready for the next node. The first one should be familiar, it's the low pass node we saw in earlier flows. It's followed by two nodes that group messages together. Before I understood how the **batch** and **join** nodes worked, I wrote a user-defined node that provides the same functionality. Not only was it easy to do it was ultimately unnecessary.

The batch node adds a property to each message as it passes through the node - that prop indicates the messages position in a specific group of messages. The join node then looks at the property and bundles related messages into an array and sends the array downstream. The batch node allows you to configure the window size and if you need a rolling window you can configure the size of the overlap.

Just like the smooth operator in the monitoring flow and the threshold value in the diagnostics flow, we can manipulate the batch node configuration and observe how those changes affect the outcome.

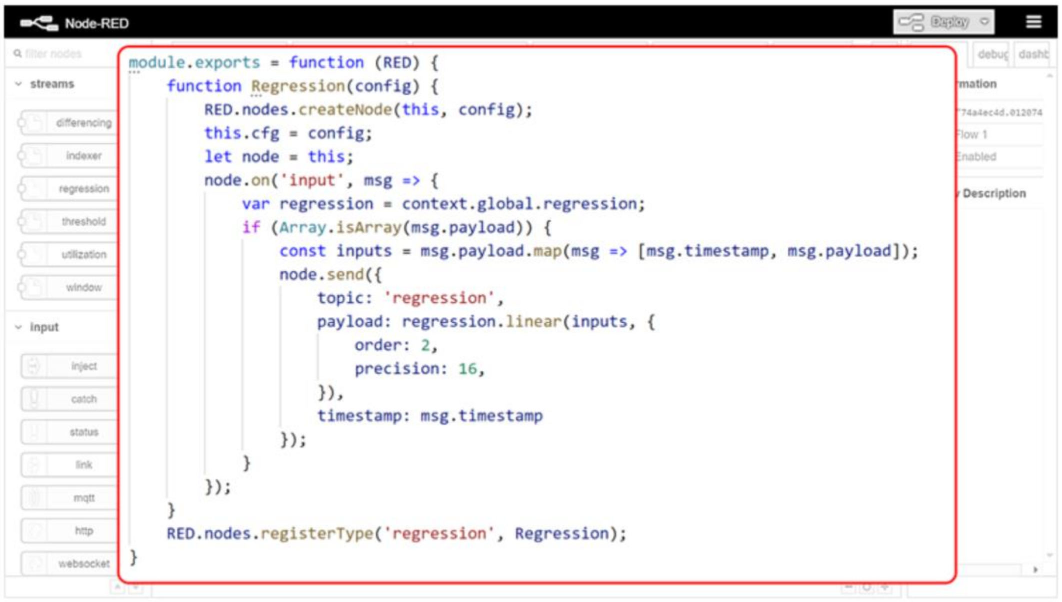


```
// Anything in this hash is globally
// available to all functions. It
// can be accessed in a function block as:
//    context.global.os

functionGlobalContext: {
    catalogDir: 'catalogs',
    path: require('path'),
    moment: require('moment'),
    regression: require('regression')
},
```

-forecasting-

The yellow regression node is a user-defined node that makes use of an npm package. Packages are made available through the global context by *requiring* the package in the node-red settings.js file.

Here you can see that we set four properties on the global context - one string and the results of three calls to require(). These properties are available to all user-defined nodes as well as function nodes.

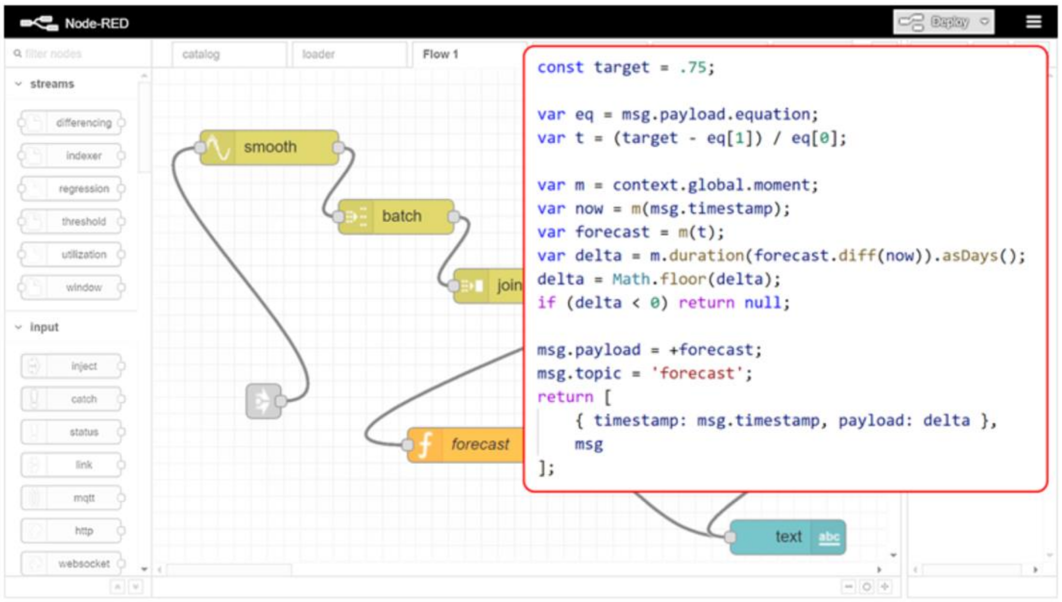


```
module.exports = function (RED) {
    function Regression(config) {
        RED.nodes.createNode(this, config);
        this.cfg = config;
        let node = this;
        node.on('input', msg => {
            var regression = context.global.regression;
            if (Array.isArray(msg.payload)) {
                const inputs = msg.payload.map(msg => [msg.timestamp, msg.payload]);
                node.send({
                    topic: 'regression',
                    payload: regression.linear(inputs, {
                        order: 2,
                        precision: 16,
                    }),
                    timestamp: msg.timestamp
                });
            }
        });
    }
    RED.nodes.registerType('regression', Regression);
}
```

-forecasting-

This is the code for the regression node. The first thing that happens in the event handler is we grab the regression package from the global context. The next bit of interest is the call to map() to coerce the data into the form that the regression package needs. Recall that the messages have timestamp and payload properties on the message object, the regression package wants a list of lists.

Don't get fooled by the **order** property of that options object. It is effectively ignored because we're calling the .linear() function. The regression package also has .exponential(), .logarithmic(), .power(), and .polynomial() regression models.



```
const target = .75;

var eq = msg.payload.equation;
var t = (target - eq[1]) / eq[0];

var m = context.global.moment;
var now = m(msg.timestamp);
var forecast = m(t);
var delta = m.duration(forecast.diff(now)).asDays();
delta = Math.floor(delta);
if (delta < 0) return null;

msg.payload = +forecast;
msg.topic = 'forecast';
return [
    { timestamp: msg.timestamp, payload: delta },
    msg
];
```

-forecasting-

After all of that smoothing and regression-ing we can finally do some forecasting in the **forecast** node.

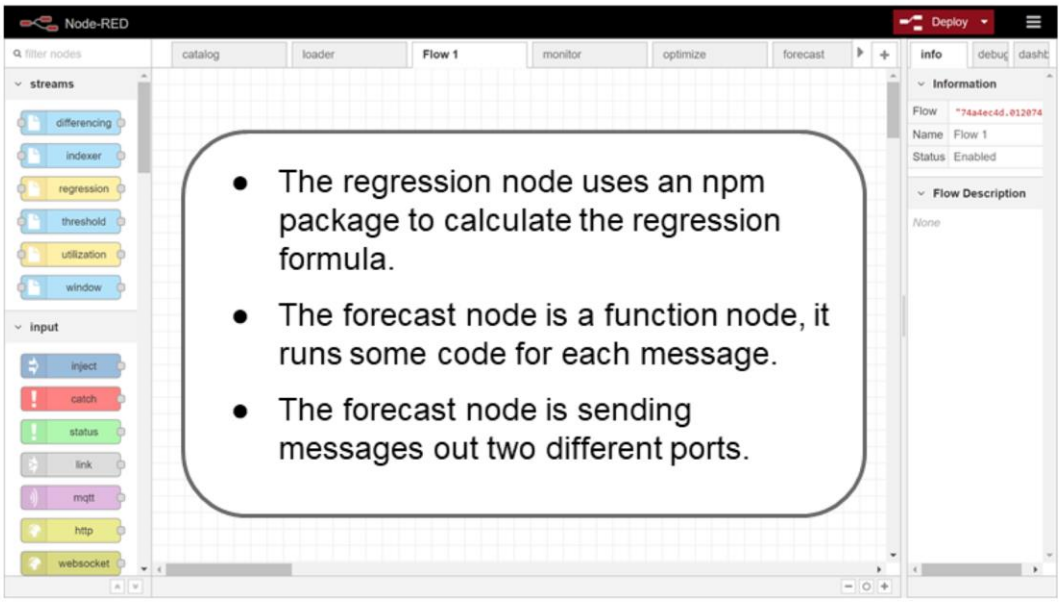
The forecast node is implemented as a node-red **function** node. A function node is like the event handler portion of a user-defined node.

There are really only two things going on in this node that are of any interest. First is that third line where we're using the coefficients from the regression model to figure out at what time the regression crosses the 0.75 threshold.

The next thing of interest is the return statement.

When you set the properties of a function node you can define the number of output ports the node will have. So far, all of the user-defined nodes that we’ve looked at return a single message object but here we see that the function is returning an array of messages. Each message in the array will be sent out of a different port. The first message has been formatted for the gauge widget and the second output is formatted for displaying the projected time-of-failure in a text widget.

It’s like a function that has TWO return values! Pretty neat, huh?



- The regression node uses an npm package to calculate the regression formula.
- The forecast node is a function node, it runs some code for each message.
- The forecast node is sending messages out two different ports.

-forecasting-

There is a lot going on here.

- User-defined nodes - we've already covered them.
- The regression node uses an npm package to calculate the regression model.
- The forecast node is a **function** node, it runs some code for each message.
- The forecast node is sending messages out two different ports.

It seems like we've covered a lot of the capabilities of node-red but we're not done yet!

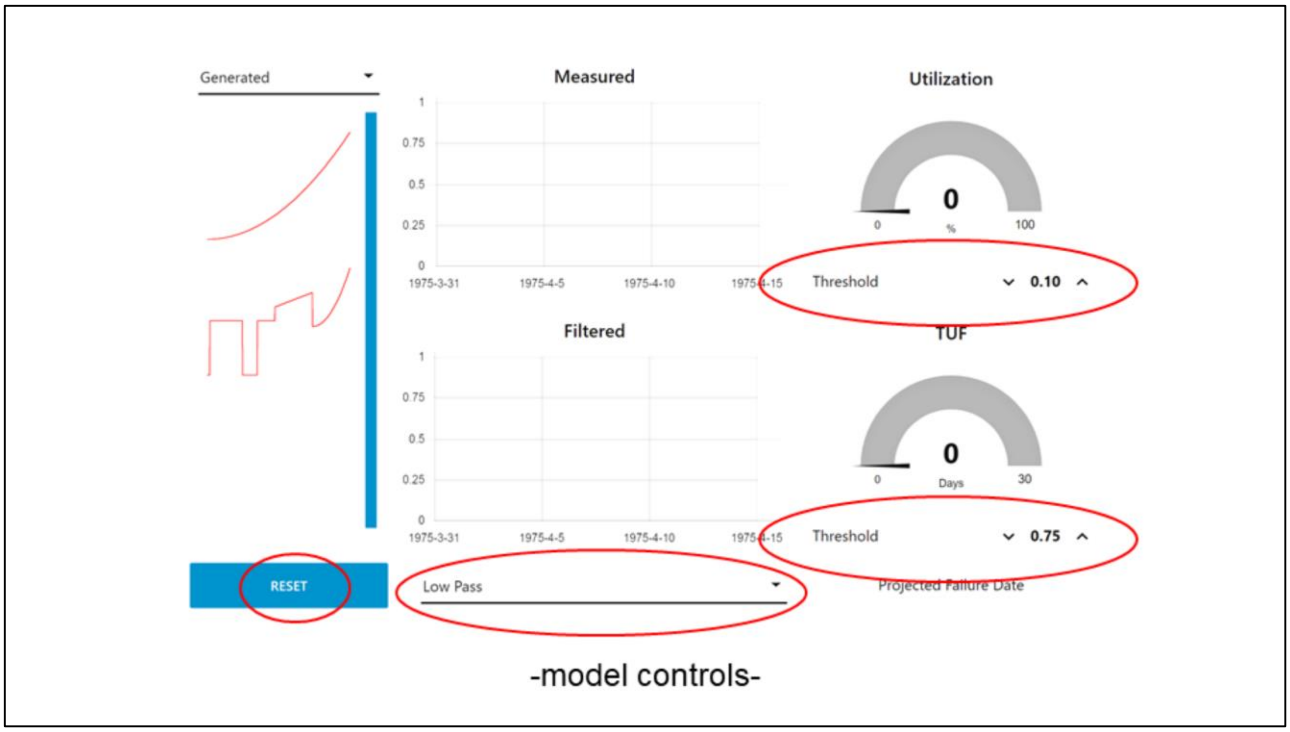
- Mach 3.2 (2,200 mph)
- 85,000 ft ceiling
- 92% titanium airframe
- 53,490 total flight hours

-dynamic vs static-

I've mentioned a few times that one of the reasons to build a model is to have the ability to change the configuration of the model. It's the difference between these models of the **SR-71 Blackbird**. The one on the left flies and the one on the right doesn't.

It's the difference between a static and a dynamic model.

At this point our digital twin model is quasi-dynamic; we can change settings like the utilization threshold through the configuration panel but then we have to deploy the change to the runtime environment.

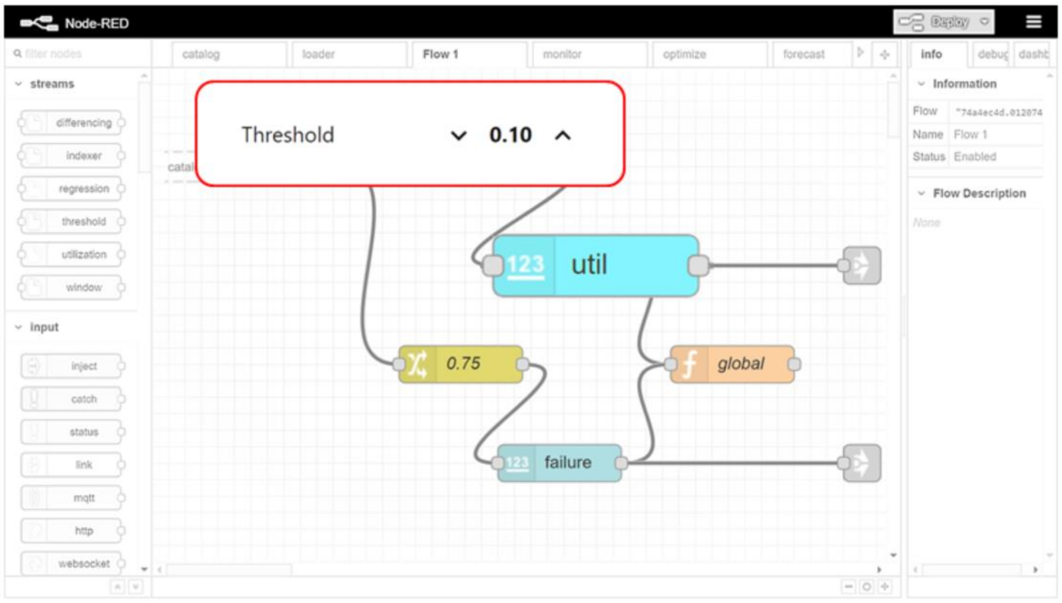


In addition to its data visualization nodes, the dashboard package also has data input nodes.These widgets allow us to change the behavior of the system by injecting messages into the flow.

The numeric inputs on the right change the values of the utilization and failure thresholds

… while he dropdown in the center selects different filters

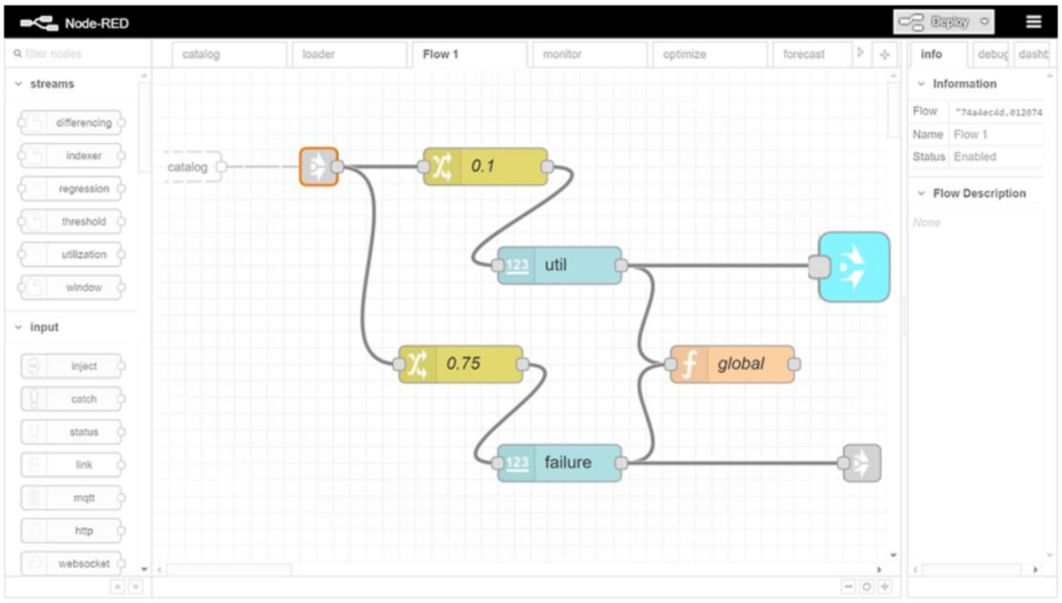
… and the button on the left sets everything back to sane values.



-threshold controls-

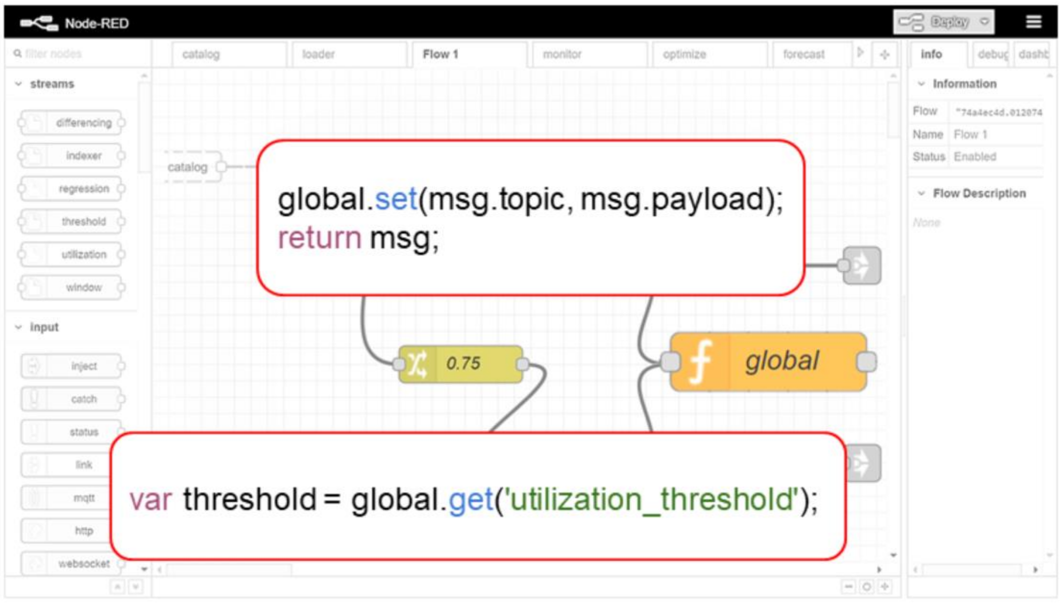
The numeric inputs send a message each time you click on the up or down arrows, the payload of the message is the new value of the control.

That message is sent to a couple of downstream neighbors...



-threshold controls-

… a link node and a function node. The link provides a mechanism for alerting other flows that the value of the threshold has changed.



```
global.set(msg.topic, msg.payload);
return msg;
```

```
var threshold = global.get('utilization_threshold');
```

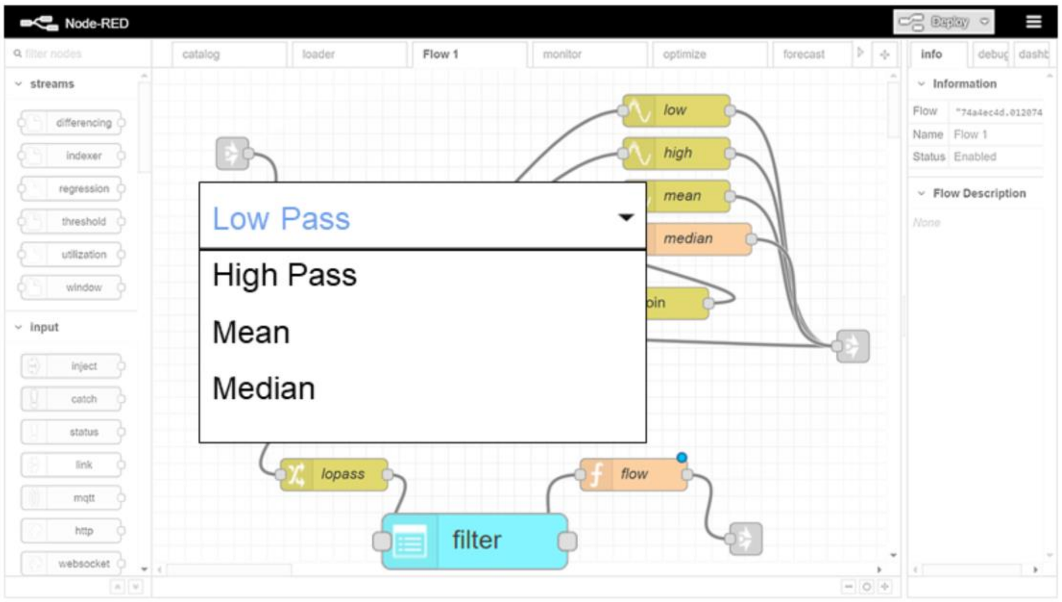
-threshold controls-

The function node takes the new value and stores it in the global context.

Each node has three contexts available to it at the

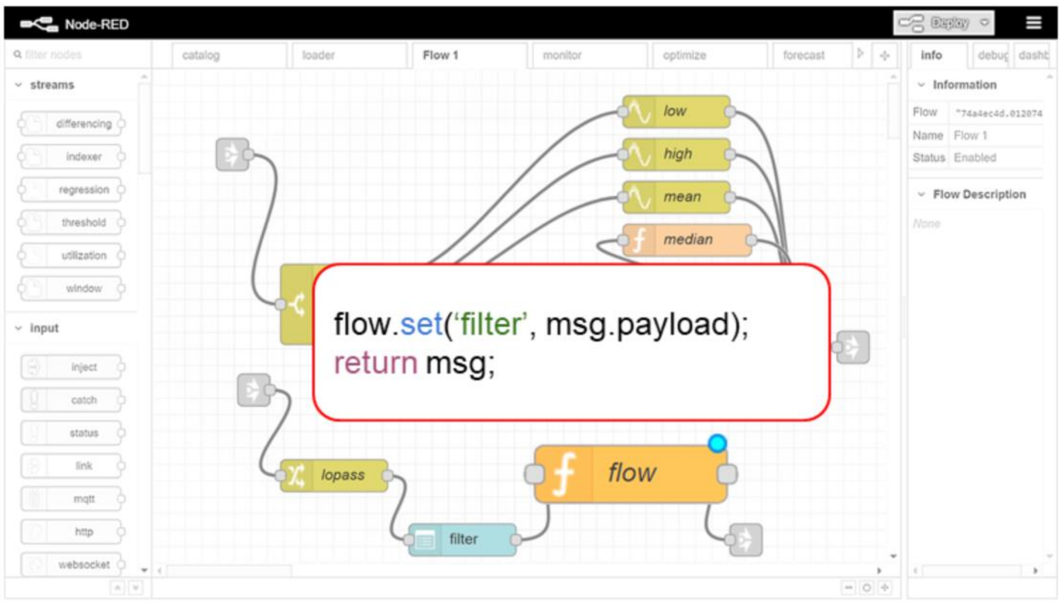
- **Global**, **flow**, and **node** level

The utilization node was refactored to read the threshold from the global context and to use that value in the calculation of the utilization KPI.



-filter control-

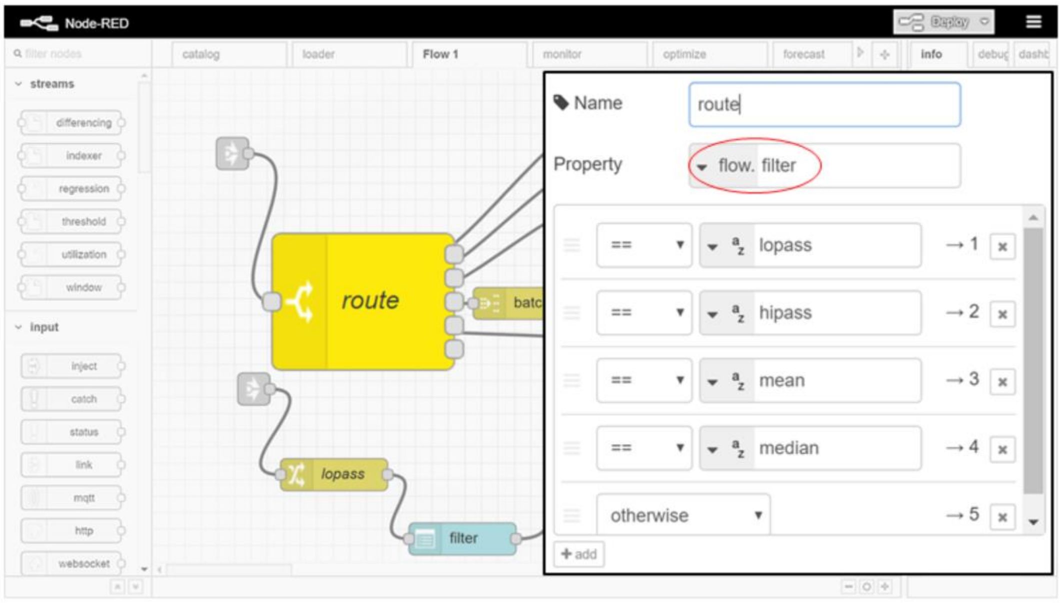
For selecting the filter we have a **dropdown** node. Like the numeric input node, the dropdown node sends a message downstream each time its value changes.



```
flow.set('filter', msg.payload);
return msg;
```

-filter control-

In this filter selector flow, the downstream neighbor of the dropdown is a function node. The selected filter is being saved to the flow context where we'll use it to route data messages to the correct filter node.



Name: route

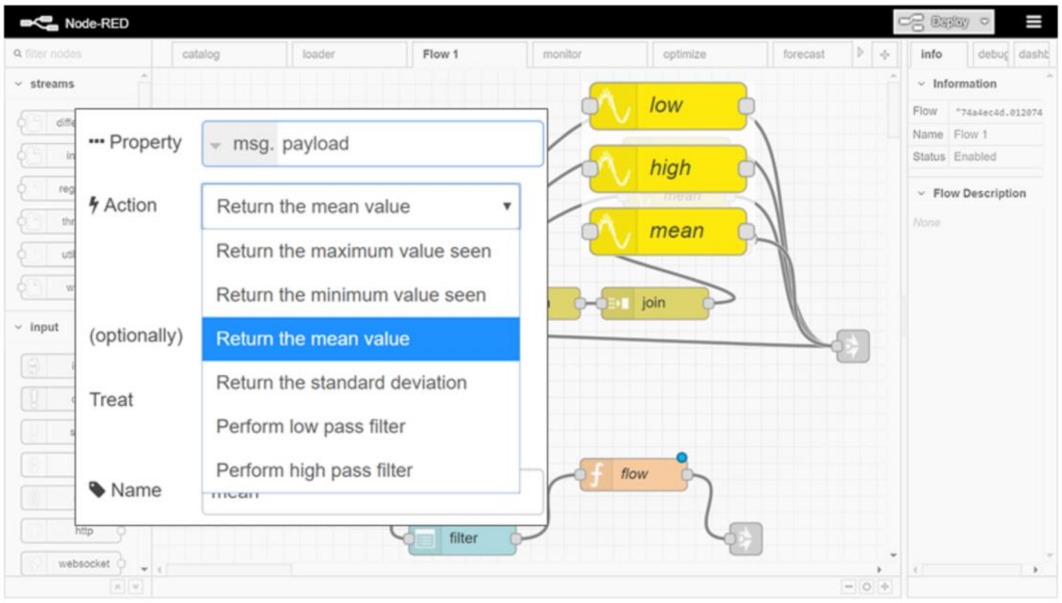
Property: flow. filter

| | | | |
|---|---|---|---|
| == | lopass | → 1 | × |
| == | hipass | → 2 | × |
| == | mean | → 3 | × |
| == | median | → 4 | × |
| otherwise | | → 5 | × |

-filter control-

The smooth node which provides the different filtering functions doesn't have the ability to set the filter type on the fly so we need to use a different mechanism to provide multiple filters.

The switch node labeled **route** is an example of sending a message out different ports based on a property. In this case, we route the message according to the *filter* property of the flow context.



Property: msg. payload

Action: Return the mean value

- Return the maximum value seen
- Return the minimum value seen
- Return the mean value
- Return the standard deviation
- Perform low pass filter
- Perform high pass filter

(optionally)

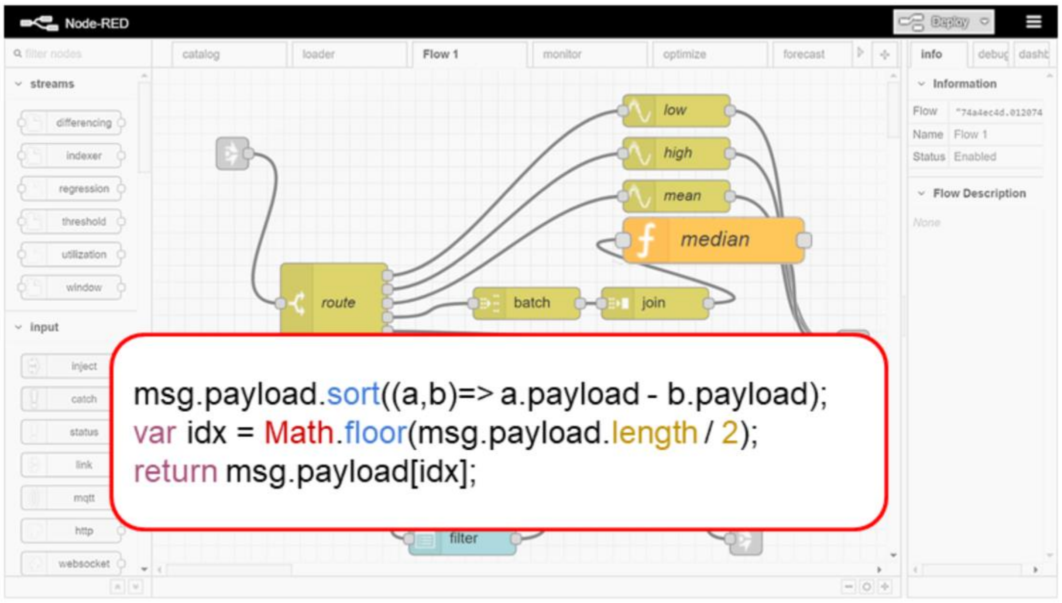
Treat

Name: mean

-filter control-

The smooth node provides three different filters as well as a few statistics about the data stream. Here you can see that we've used three instances of the smooth node, each with different settings.

Data messages come in through the link and get passed to the switch node when routes them to one of the filters. The filtered result is then made available to the other flows through the output link.

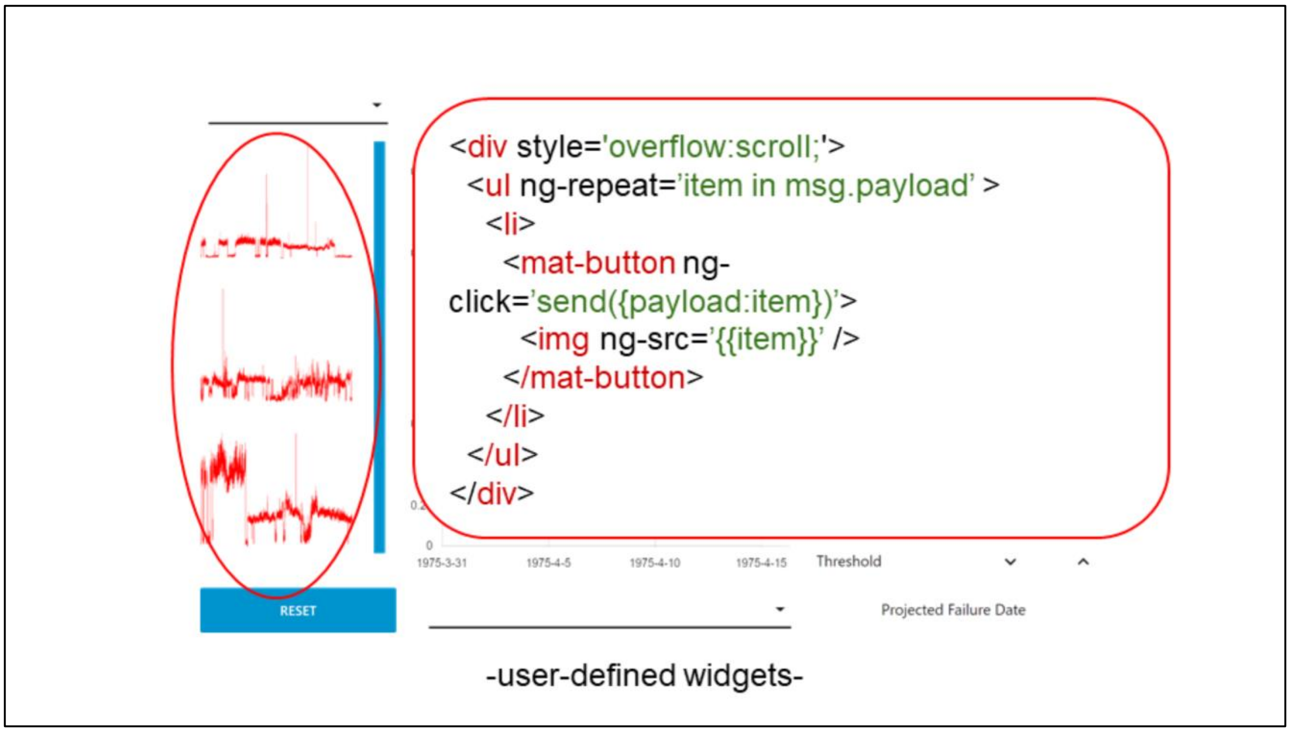


```
msg.payload.sort((a,b)=> a.payload - b.payload);
var idx = Math.floor(msg.payload.length / 2);
return msg.payload[idx];
```

-filter control-

You probably noticed that we had four options for on the dropdown node but only three types of filter in the smooth node.

The fourth filter is defined in a function node. You’ve seen the batch/join sequence before, it is creating an array of messages. The median filter simply takes the array, sorts it, and returns the message in the middle.



```
<div style='overflow:scroll;'>
  <ul ng-repeat='item in msg.payload' >
    <li>
      <mat-button ng-
click='send({payload:item})'>
        <img ng-src='{{item}}' />
      </mat-button>
    </li>
  </ul>
</div>
```

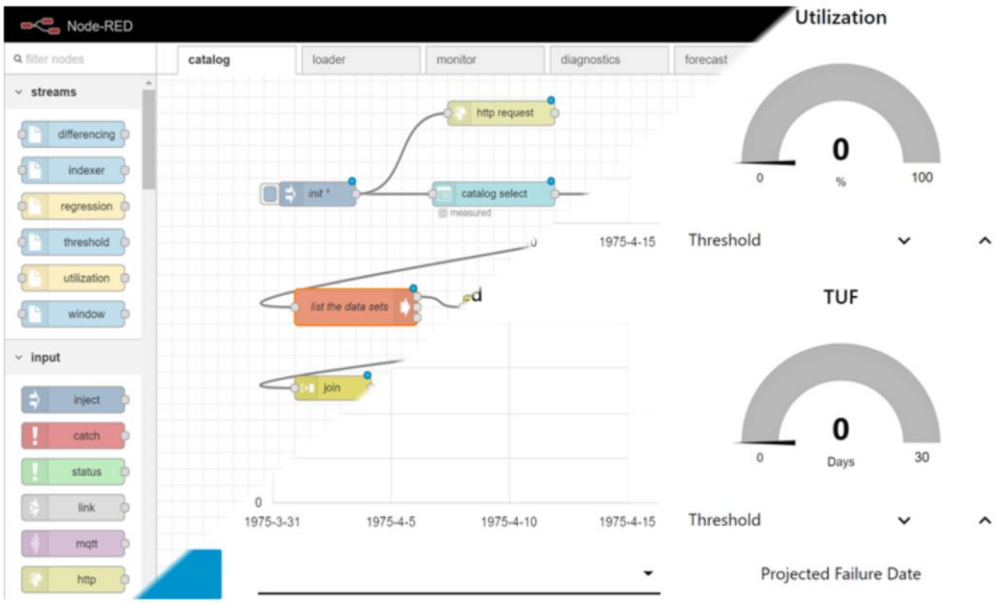
-user-defined widgets-

Since we’re on the subject of user interfaces, what happens if the dashboard package doesn’t have the widget you need? You build one, of course.

I needed a way of displaying a bunch of images which would emit a message when clicked. That scrollable list of buttons on the left.

It turns out that you can use angular and angular-material directives in the template node which is part of the dashboard package.

Now *that* is squishy!



-demo-

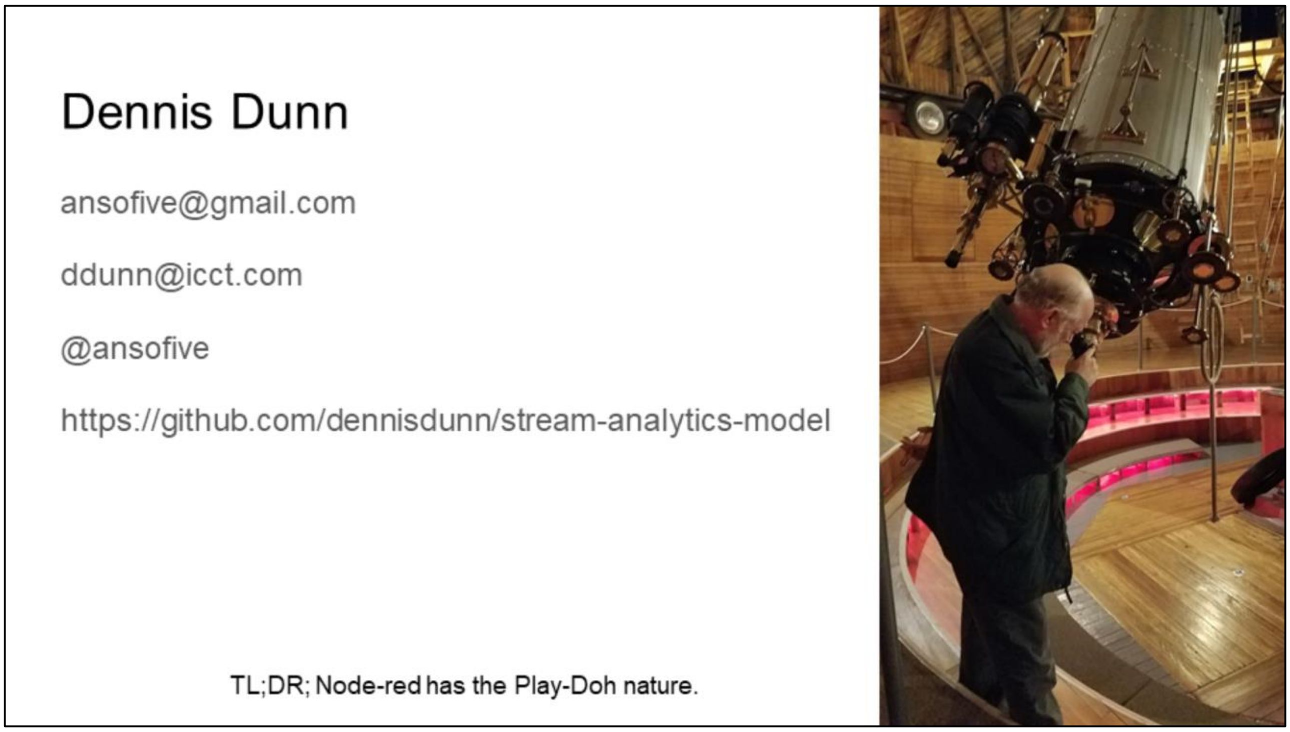
Before we close let’s try a short demo.

Charts and gauges are pretty but what if I’m not staring at the screen? I’d like some sort of notification when the **time-until-failure** is less than … 2.

There is nothing in the current model to support that so I’m going to add it.

This is a raspberry pi zero with a pimoroni blinkt! Hat - it has 8 addressable LEDs of RGB goodness. The LEDs are controlled by node-red running on the pi zero which is exposing api endpoints over HTTP. This is the *versatile* property we were talking about.

Let me re-configure for endo-thermic propulsion …



Dennis Dunn

ansofive@gmail.com

ddunn@icct.com

@ansofive

https://github.com/dennisdunn/stream-analytics-model

TL;DR; Node-red has the Play-Doh nature.

As software engineers we are often tasked with writing software for problem domains that we are unfamiliar with. That's OK, just find a tool with the Play-doh nature and build a model.

My name is Dennis Dunn and I'd like to thank you for coming. Here is some contact info and resources for this talk. I would love to answer any questions I can or we could just try the whole conversation thing.