Gauge Watcher: An IoT monitoring system for legacy analog gauges

W251 project, Summer 2018
John Blakkan
Cheng Cheng
Andrew Mamroth
John Tabbone

Abstract:

We present Gauge Watcher, a system for monitoring, reporting, and analyzing readings on legacy analog gauges. This IoT solution uses low cost Raspberry Pi microcomputers to optically read gauges, then reduces the image data to a gauge value using a cloud-trained Convolutional Neural Network (CNN.) The data are then distributed to monitoring/reporting clients via the MQTT publish/subscribe protocols and brokers, and presented to users for visual inspection via a JavaScript animated web interface.

Background and need

Many legacy monitoring and controls systems were built in an era of manual monitoring. Analog gauges were monitored by human operators or had, at best, simple electrical contactors to detect over-limit or under-limit conditions. These gauges are sometimes installed individually in isolated locations, or may be centralized into panels in control rooms. These remain widespread in many industrial, commercial, utility, and military facilities.

Rising labor costs, and an increased awareness of the consequence of human error in monitoring such gauges motivates a desire to automate this task. With automation, for example, gauges in remote locations could be read continuously, rather than on a meter-reader's schedule. In the case of panels of multiple gauges, human monitoring could be augmented with an automated system. This could prevent human error, as well as allow more sophisticated, data-driven analysis of patterns and faults.

Consider, for example, a complex correlation of readings on multiple meters which might go unnoticed by human operators. A trained machine learning (ML) system could identify rarely-seen meter readings which are predictive of failures in the system being monitored. Other examples might include redundant monitoring of critical or life-support systems: Onsite human staff could be augmented by secondary (and even tertiary) remote, redundant monitoring.

One alternative to our proposal would be to replace or augment existing analog meters with new IP-enabled sensors. This is an attractive alternative for new installations, and, in certain applications, for retrofit of existing meters. One well-known example is the deployment of smart meters for household utility services. Commercial and municipal electric companies have been able to justify this based on labor cost saved in meter reading. There have been other advantages- Smart water meters have been used to identify residential plumbing leaks (e.g. a small, continuous usage every minute in a 24 hr period is likely a leak, rather than normal usage.)

Not all legacy uses are well-suited for a “new gauge” retrofit, however. In some cases new gauges may not be available. Or, due to physical access concerns, it may be very costly to replace them. Finally, for some applications (e.g. avionics, nuclear, public utilities, telco facilities,) equipment – particularly monitoring equipment – may be subject to significant regulation or re-qualification procedures. It might be possible to replace a gauge with an electronic version, but regulatory compliance may be cost-prohibitive. While it may be infeasible to remove and replace the old gauge, it would be permissible to add an additional monitoring capability (so long as the original gauge is untouched, still operable, and still visible to local operators.)

Simple closed-circuit TV monitoring could be proposed for this task. However, this only enables remote monitoring and recording, and could not generate specific alert messages. It also uses high communication bandwidth for the actual information transmitted (e.g. gauge needle position.)

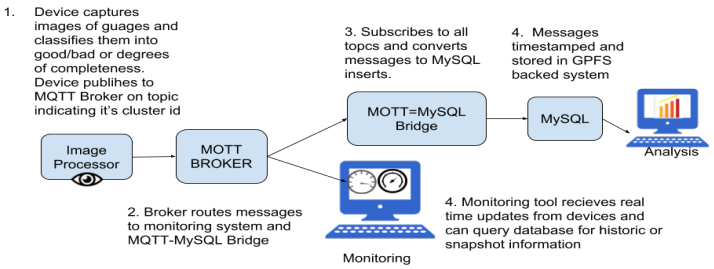
Architecture

The overall architecture of Gauge Watcher is shown in the figure below. Each element will be describe more fully in subsequent sections.

Training data for the Raspberry-Pi resident CNN was created by driving analog meters to known values, photographing them, and storing the value-labeled images in cloud storage (Softlayer gpfs.) A CNN was trained in the cloud, and the weighted model transferred to the multiple Raspberry Pi monitoring systems. Note that our intial implementation is a gauge reading classifier (e.g. resolve each gauge to one of n ranges, for n a discrete value), rather than a gauge reading regression (e.g. attempt to resolve to a precise reading.)

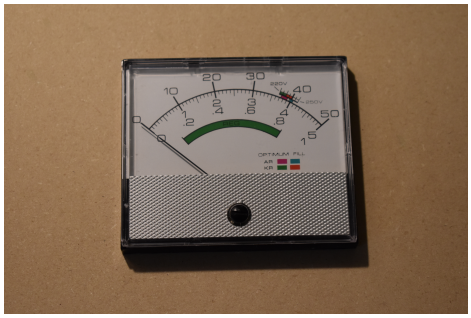
While running, the Raspberry pi performs continuous image processing. It samples a video stream approximately once per second (this sampling speed could be altered), uses openCV to perform preprocessing (e.g. converts from color to grayscale to thresholded line images), then uses the pre-trained CNN model to classify the image and publish the result to a cloud-based MQTT broker.

JavaScript-enabled webpages also subscribe to the MQTT broker, and update a user display. The webpage performs a minimal amount of alert processing in addition to displaying the gauge state; It adds a color code indicating if the meter is in the high end of its range.



Hardware

A variety of gauges with different face plates were procured from a local (Santa Clara, CA) surplus dealer.

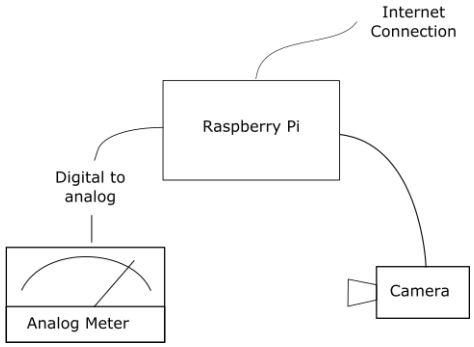


Our initial plan was to use one type of microcomputer (An Arduino Uno) to drive the meters for training and demonstration, and another type (Raspberry Pi B 3+) for the actual gauge reading and CNN processing.

However, we ultimately determined – after controlling gauges with both types of microcomputer - that the Raspberry Pi was suitable for both tasks, and stopped further development on Arduino.

Initially we were planning to use the microcontrollers' PWM (Pulse Width Modulation) output pins to drive the meters. However, for both the Arduino and the Raspberry Pi, PWM drive had significant accuracy issues in the low range of the meters. It consistently read too high, beyond what we could linearize with software. We attempted augmenting the PWM output pin with a MOSFET driver circuit to get better accuracy on the lower ranges of gauge values, but accuracy was still poor. Ultimately, we used the Raspberry Pi's I2C interface to drive an external Digital to Analog Converter (DAC) through a series resistor to precisely control the meters. With a small amount of additional tuning of the meter-drive linearization, we achieved the needed accuracy to produce training data and run the demonstration.

RPi Hardware Configuration



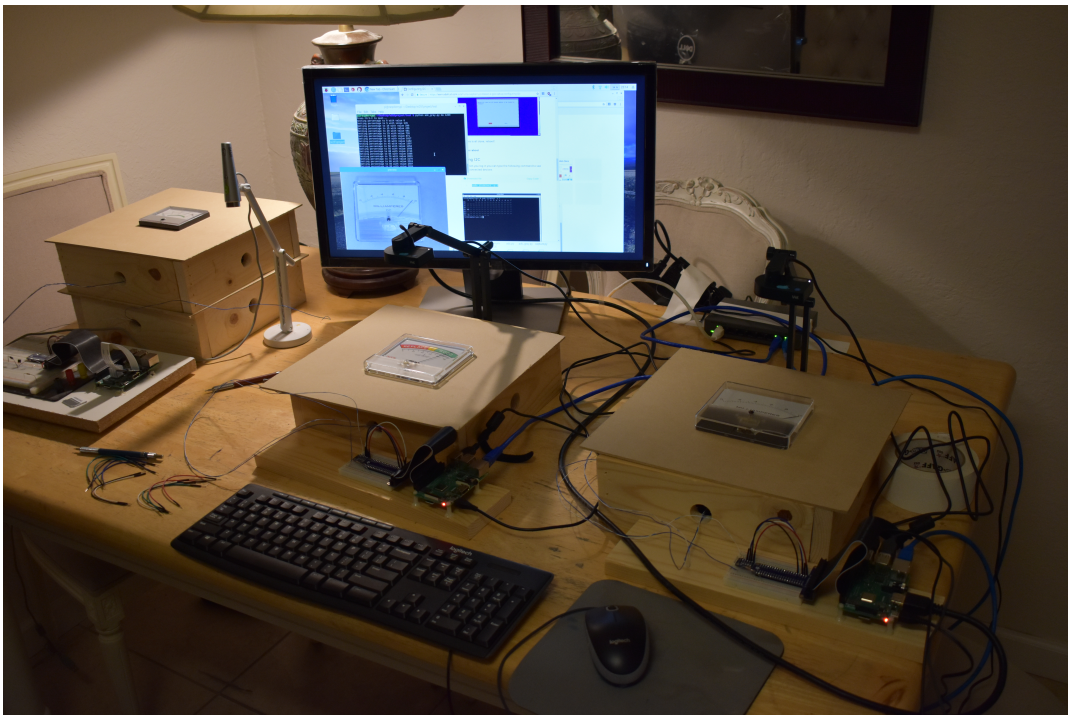
We still had various practical issues; e.g. one of our surplus meters was determined to “stick” in place in its high range.

We mounted the meters in fiberboard panels and built wooden frames for the panels to provide durability and access. We built a total of four meter panels and two Raspberry Pis (assembled on prototype boards with the DACs). We used 8M Pixel USB cameras rather than the Raspberry Pi’s native camera simply for convenience and mechanical durability. A production system would use the Raspberry Pi camera for its lower cost. In addition, we had a third “development” Raspberry Pi (older version, lower clock rate, less capable camera) which we did not use for training data capture or in operation. The development system is on the left in the photo below. Note that in operation only the two rightmost Raspberry Pi’s are used (with the black cameras in the back), and two of the gauges.

Training data collection

The photo below shows the generation of the training data. We initially ran the system at a fairly slow pace (4 seconds per training image, with two systems running in parallel), so it took many hours to produce 1200 images for each of the four gauges. (Fortunately much of this time was unattended and could be run overnight.) We eventually determined that meter-settling time on the gauges would permit images to be captured, processed, and saved at about four times this rate. We later generated additional groups of 3000 images per gauge.

The photograph below shows that the meter on the far right of the image is at near high-scale, and is being displayed (in gray scale) in the lower left of the monitor. The two stacked meter boxes in the left are awaiting use, and the “development” Raspberry Pi (with its larger breadboard and different camera) is on the far left.



Model and Training (Cloud)

Keras/Tensorflow training was completed on a multicore cloud-based server (IBM softlayer.) With the model and weights transferred to the Raspberry Pis for classification.
(ref: https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html)

Multiple changes were made for the sake of simplicity and to reduce training time as the images are very simple and significant training was not required. The first two convolutional layers were reduced to a filter size of 1, similar to those used in squeezenet, which did not have significant impact on accuracy but significantly reduced training time. Also, because the validation images and the training images were VERY similar, dropout was reduced to 0.3 as generalizing was less of an issue (once again speeding up training time). Also most importantly, the images were reduced in size by half which had to most impact on reducing training time. Because the training images were black and white edge images, it didn't affect model accuracy to reduce the resolution of the images and had an enormous impact on training time. The model hit nearly 100% accuracy after roughly 2-3 epochs on 1000 training images per class and 100 validation images per class.

Once training was complete the model along with the weights were saved as separate files so they could be loaded in the “watcher.py” program Resident on the Raspberry Pi for image classification.

RPi resident software

The Raspberry Pi is running the current Raspberrian Distribution. (An X-Ubuntu distribution was initially tried on the development system; it works, but needs special modifications to run on the Raspberry Pi 3B+, so we standardized on Raspberrian. We use openCV, Keras with Tensorflow, and the paho-mqtt library.

There are two main python routines: adc_gray.py and watcher.py. adc_gray.py creates training data by driving the meter (via the DAC), taking an image, and saving the JPG image with the meter value encoded in the file name. For some of our data, we used SSHFS to actually locally mount the cloud storage file system and write it directly, while for others we simply saved the data local to the Rpi's 32GB SSD card and wrote it to the cloud in bulk.

watcher.py initializes openCV, the CNN, and the MQTT broker connection. It then loops, randomly setting meter values, gathering images, preprocessing them, classifying them, and publishing results to MQTT. An important option in watcher.py is the ability to bypass the classifier. In this mode, watcher.py sets the meter to a random value, but reports that value directly to MQTT (bypassing the image processing and CNN classification.) Including this diagnostic bypass option was critical to the process of developing and debugging the MQTT and User Visualization/JavaScript portions of the project. watcher.py has the ability to report back a "VOID" value, as well as a classified value, to provide an error indication to users which is separate from a "Gauge reads zero" value.

Performance

Training accuracy was found to be very close to 100% against our "hold-out" set of images.

When we trained the model, we randomly selected 10% images out from the training dataset and use them as validation data. Through the process of training, the accuracy was always around 100%. After we got the pre-trained model, we also ran the model and weights on other random images (the static images and camera images), it turned out to reach 100% accuracy as well.

However, we find that the actual performance of the physically implemented system is lower (see the final section for a discussion of this.)

MQTT broker

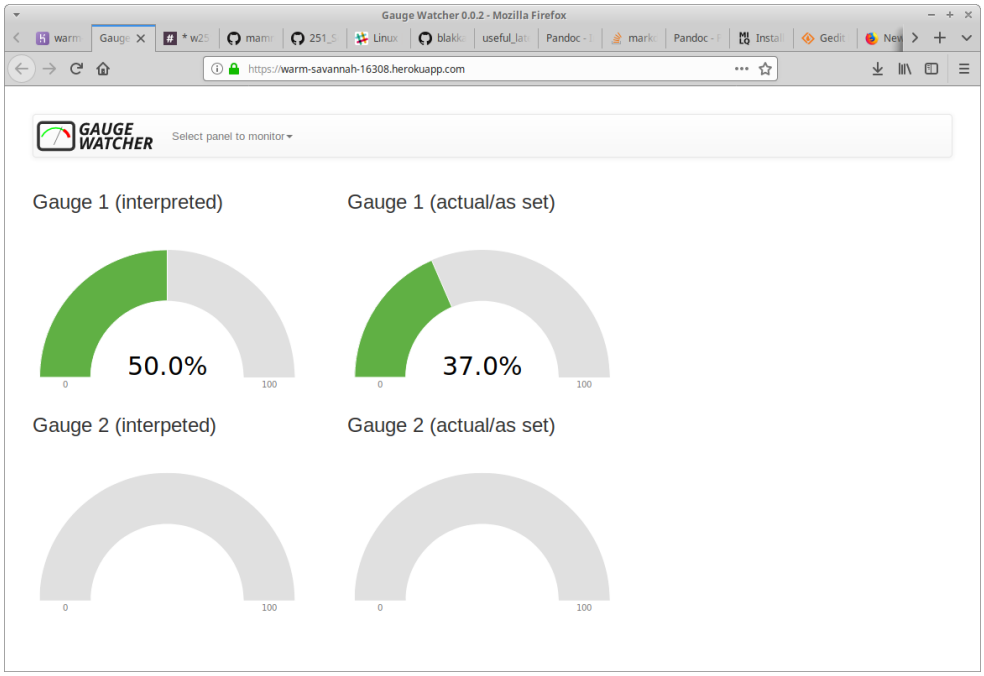
During the course of project development, we used two different MQTT brokers. Early development was done using a commercial MQTT broker (a services offered by ADD ADD in partnership with heroku.com). Later, we transitioned to our own MQTT broker running on a Softlayer Virtual Machine.

User Visualization

Our visualization is a webpage. Javascript subscribes to the MQTT broker and, upon message receipt, updates the appropriate abstract gauge display. The abstract gauge displays are designed for uniform appearance (they do not attempt a photograph-like reproduction of the different gauge types). The displays are produced as .svg element using the d3 library (in conjunction with the c3

chart library).

We host the page on a ruby-on-rails site we developed and deploy on the heroku.com Platform-as-a-Service (PaaS) site. Initially we anticipated a need for backend processing of the MQTT subscription, but we found that the JavaScript MQTT interface was sufficient. Although the application is effectively serverless, we retained the ruby-on-rails backend for convenient deployment and ability to serve a large number of requests for the site.



Operationalization

Planning and development to operationalize the computer vision client has been considered. Creating a system suitable for a production environment expands the scope beyond the resources of the project. However the team did consider key questions necessary to move this project into a production environment.:

- How can we support multiple image recognition clients?
- What happens to the classification and images after the device creates them?
- How can data scientists analyze historical data?
- How can operators review current information before writing a ticket?

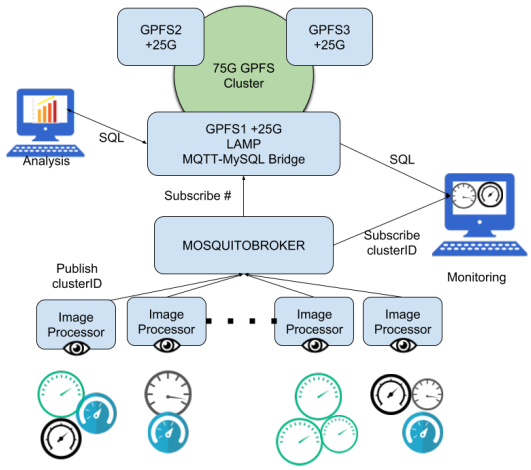
Architectural Components

Storage: 3 Redhat servers contributing 25G each to a GPFS based shared file system. MySQL database hosted on gpfs1 sbcked by clustered storage.

IPC: MQTT message broker hosted on Ubuntu server provides intra-process communication

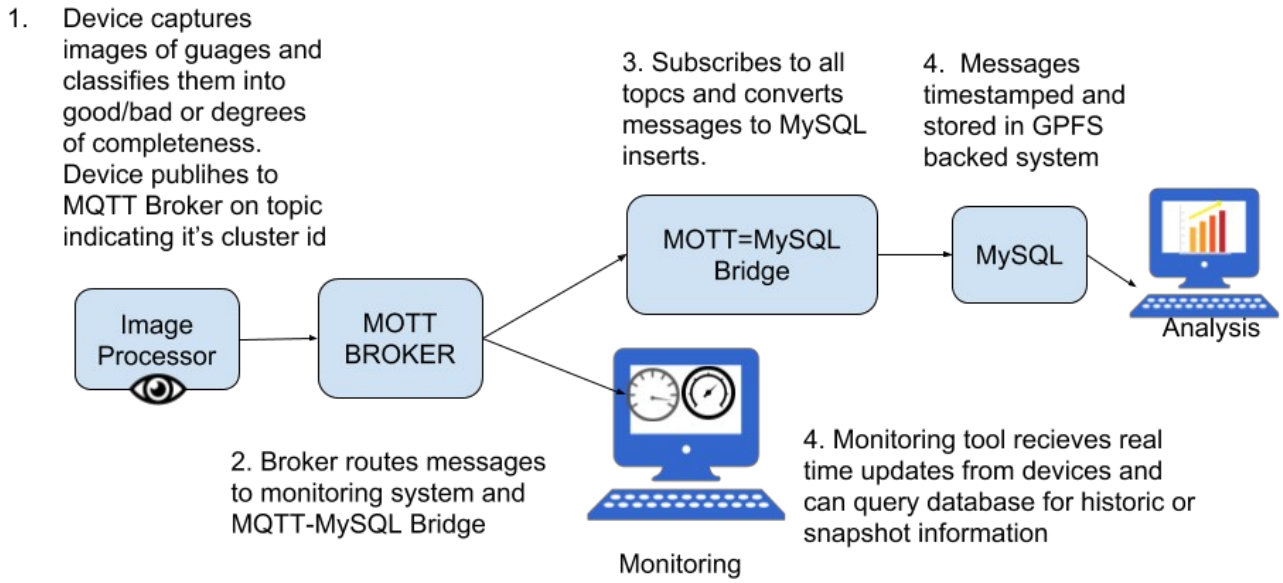
Image Processor: Rasbery Pi based image capture and classification of gauge images.

Monitoring: Report status to users



Data Flow

Data moves through the system primarily through the publish/Subscribe framework MQTT. Image Processing devices publish their findings to topics that describe their cluster id. Subscribers of this data are the MQTT-MySQL Bridge and the monitoring clients. The bridge reads each message, time stamps it, and inserts it into the MySQL database. Monitoring Clients can determine which clusters they are interested in monitoring and subscribe directly to their feed. Clients can also query the database for historical information. Left for later implementation is the ability for Image Processing devices to send an actual image for storage, which can be retrieved by Monitoring Clients. This may be useful in the cases where there are alarms and the operator wants to see an image of the gauge that caused the alarm before issuing a ticket. Finally, historic data stored in the database can be accessed by data scientists for later analysis.



Conclusion and proposed future work

We conclude that the Raspberry Pi can be used effectively to host a CNN for remote gauge monitoring.

Note that for this implementation, each Raspberry Pi monitored a single gauge; an obvious practical extension of the project would be to monitor multiple gauges per Raspberry Pi by additionally processing images to separate individual gauges into sub-images, and interpret the sub images.

Also, the webpage adds a color code indicating if the meter is in the high end of its range. This could be extended to alert on different conditions, even looking at combinations of conditions over multiple gauges. It could also generate email messages or take other alerting actions.

Left for later implementation is the ability for Image Processing devices to send an actual image for storage, which can be retrieved by Monitoring Clients. This may be useful in the cases where there are alarms and the operator wants to see an image of the gauge that caused the alarm before issuing a ticket. Finally, historic data stored in the database can be accessed by data scientists for later analysis

Finally, we used a back-end server (Ruby on rails) to present our JavaScript-enabled webpage. Currently this server doesn't perform any function beyond serving the page. It could be modified to perform further analysis of the MQTT stream – with user defined altering. It is also where we would implement user-access controls which, while not required for this project, would be required for a commercial or production-worthy system.

We initially planned to monitor only motion-detection on the gauges (i.e. a rapidly moving needle as being worthy of alerting a human operator.) As the project evolved, we become more interested in analyzing actual values. We made several tradeoffs – for example, instead of presenting our CNN with grayscale images, we pre-process into edge-detected line images. This was, in effect, de-facto feature engineering, which isn't making full use of the power of a NN. Yet, for our problem domain, the meters are fully represented by line drawings, so we don't regard this choice as "feature engineering" so much as "compact data representation" of the information in the images. Also, we used a relatively simple CNN topology (simpler than SqueezeNet, for example.) We found it would very effectively classify our hold-out data. Yet when run on real-world images, it didn't reach the level of accuracy achieved on the training set (indeed, is showed both inaccuracy and bias in identify high-range values as mid-range.) From informal experiments (i.e. moving the camera around and changing the lighting,) we find that a significant amount of this arises from camera position and angle. We suspect that adding additional layers to our CNN (including, perhaps, a full SqueezeNet) would improve this.