Build a cognitive IoT app in just 7 steps

Transform your cognitive IoT solutions by applying advanced machine learning and detect anomalies in real time on an IoT sensor data stream

Romeo Kienzler
March 14, 2018
(First published March 22, 2017)

Build a cognitive IoT solution, following an edge computing architecture. Push your analytics out to the gateway, and use advanced machine learning to detect anomalies.

Note: IBM Data Science Experience (DSX) is now IBM Watson Studio. Although the name has changed and some images may show the previous name, the steps and processes in this tutorial will still work.

What makes cognitive IoT so cognitive? How can cognitive IoT be achieved? For some people, it is simply that the interaction between you and things becomes more human. For example, in an elevator you might hear the latest news based on your social media psychological profile. Or, you might hear some relaxing music if it detects that you look stressed.

To me, cognitive IoT is more about what happens behind the scenes. Although human-computer interaction (HCI) is a key part of many cognitive IoT solutions, I'll focus less on HCI because it is already turning into state-of-the-art tech. With IBM Watson cognitive APIs (like text-to-speech (TTS), speech-to-text (STT), machine translation, and visual recognition) working behind the scenes in your cognitive IoT apps, advanced HCI is now only limited by your imagination.

Cognitive IoT and artificial intelligence (AI) are nothing more than advanced machine learning. And advanced machine learning has two drivers:

- Algorithms (or models) that are powerful enough to learn any required behavior from data.
- Unlimited data processing capabilities and storage to cope with the vast amount of data to be trained.

In cognitive IoT solutions, machine learning needs to take place in an edge computing architecture. Edge computing basically means that you push computing away from the cloud.
or data center out toward the sensors. Computations happen on the edge gateway near the sensors and actors (or even closer on a microcontroller between the gateway and the sensors and actors). We use edge computing architectures for two reasons because not every decision can be computed in the cloud:

- **Latency** impacts some critical decisions that make a cloud route trip untenable. Think of a smart-connected car. It is nice that it can phone home from time to time (that is, in the range of seconds or less), but if the car in front of you brakes suddenly, you want your car to respond immediately.
- **Transfer cost** can be too high if the amount of data that is created by a sensor is too much to transfer to the cloud completely. Either it is technically impossible due to link speed, or it is just too expensive, or both.

**Cognitive IoT architecture**

Don't be scared by the architecture diagram in Figure 1. This diagram outlines what is needed to do cognitive IoT. Your solution might use only a subset of these components, simplifying the architecture. And the good news is this: If you use the IBM Cloud as a service all these components are pre-installed and the operation is managed for you. In addition, these components are based on open standards and are running on open source technology, so lock-in is minimized to protect your investment.

**Figure 1. Architecture diagram for a cognitive IoT application**

The heart of every IoT solution is the sensors and actors. We sense the environment, make (cognitive) decisions, and act using actors.

Consider this evolution in computing architectures. We've evolved from silo-only thinking, to edge thinking, to cloud thinking, to edge-and-cloud thinking. Edge analytics, including machine learning, minimizes the data load on the cloud and dramatically reduces latency and data transfer cost. In addition, privacy is better preserved because not all data needs to be sent to the cloud or data can be masked.

It is important to have seamless, interchangeable deployment of analytic components among all those locations to decide the physical location during deployment and not during development time. Some platforms support dynamic relocation of analytic components during run time. In this tutorial, we are using Node-RED as such a platform.

We'll use a very simple scenario for illustration. Assume that you have a smart garden that uses a moisture sensor to decide when it is time to water your plants. It would use a stepper motor as an actor to open and close the valve of the watering hose.
Phase 1 – Silo-only computing

**Hysteresis in Control Theory** defines that a threshold is different from the direction it is approached. For example, if the valve would open and close at the same moisture level then the system will oscillate which can cause the system to over-react.

Review the architecture diagram in Figure 1. You would need a sensor, an actor, and a microcontroller. You would use the Micro_Model to read the thresholds on the moisture level to determine when to act. (You would probably use a dynamically trained hysteresis to prevent oscillation and hyper-hydration.)

Phase 2 – Edge-only computing

But what happens if it starts raining immediately after your smart garden system automatically watered your plants? Besides wasting water, your plants might get hyper-hydrated. Also, can we take weather forecasts into account? Of course. But now a microcontroller is not enough to do the job because we have to query a weather service. A machine-learning algorithm on the edge gateway can optimize the watering based on weather forecasts and the moisture sensor, and then update that model on the microcontroller (remember, we are using a hysteresis there and on the edge gateway itself).

Phase 3 – Edge and cloud computing

Now let's introduce a couple of more components from the architecture diagram. The first step is to connect the edge gateway to the cloud. The de facto standard for this connection is using MQTT and an MQTT message broker. Publish/subscribe models facilitate deployments of n-n connection scenarios. Sending data from the gateway to the queue is not sufficient, so an ETL (Extract, Transform, Load) component is needed. The ETL component picks up data from the message bus and stores it, transfers it over to some real-time processing component, or both. Finally, a batch analytics engine use this data and generate additional insights. To complete the architecture, cognitive API services can be called in real time or during batch processing.

In our scenario, let's link our smart garden system to your email account. Using Watson cognitive APIs it is very easy to extract your travel plans and calendar. By using this information, the smart garden system can ensure that your lawn is not wet when you are having dinner outside with your family. Or, it can automatically order fertilizer or pesticides based on the condition of your lawn. These conditions can be detected by using special chemical sensors or images taken by a webcam.

The next step in optimizing your system is to take advantage of an abundance of machine-learning algorithms. The most relevant types of machine-learning algorithms for cognitive IoT apps are forecasting, including time-series forecasting, anomaly detection, and optimization.

7 simple steps for building a cognitive IoT solution

Enough theory. Let's create a cognitive IoT solution using the IBM Cloud, the IBM Watson IoT platform, and IBM Watson Studio (formerly IBM Data Science Experience).
Our new scenario or use case is based on consumer electronics: a smart washing machine. Let's consider a device manufacturer who found out that the motor used in a specific model gets damaged when electric current is unstable. Around the globe, a stable electric current is not something that you can expect everywhere!

At first, the manufacturer relied on data from the cloud to detect anomalies in electric current. When the IoT solution received data that the electric current was unstable, a command was sent back to the washing machine to shut off the motor. Eventually, the manufacturer realizes that the latency is too high and the stability of the internet connection is too low. So, the manufacturer decides to implement analysis on the edge component, omitting the cloud for that particular process.

What you'll need to build your apps

- An IBM Cloud account. (Sign up for an IBM Cloud Lite account, a free account that never expires.)
- An IBM Watson Studio account. You can use the IBM ID you've created while registering for the IBM Cloud Account. To get started using IBM Watson Studio, including signing up for an account, watch the videos in this video collection on developerWorks TV.

Operational model for our use case

As you can see, the operational model for our use case in Figure 2 is quite different from (and far simpler than) the full architectural diagram in Figure 1. This operational model includes only the components that are necessary for our use case, and it replaces the generic component names with the specific components that are available in the IBM Cloud.

Figure 2. Operational model for a cognitive IoT app

Table 1 describes each component and its role in our use case. UML stereotypes are taken from the generic architecture diagram in Figure 1.

Table 1. Operational model components

<table>
<thead>
<tr>
<th>Component Name</th>
<th>Component Type</th>
<th>UML Stereotype</th>
<th>Component Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node-RED Edge</td>
<td>Node-RED</td>
<td>Edge Gateway</td>
<td>Used to create data flow application. Node-RED is an open source data flow editor written in JavaScript and running on Node.js. IBM created it and donated it to the JavaScript foundation.</td>
</tr>
</tbody>
</table>
1. Create an IoT app in IBM Cloud
The Internet of Things Platform Starter boilerplate contains a Node-RED engine that you will use later to process IoT messages.

   1. Log in to your IBM Cloud account.
   2. Create the IoT app using the Internet of Things Starter boilerplate. Follow these steps in this tutorial.

2. Create a device simulator to simulate device data

   1. Click Go to your Node-RED flow editor.

   2. To delete all existing nodes in the default flow, select them all and then press the Backspace or Delete key. (Note: The keyboard shortcut CTRL-A doesn't work.)
3. Go to my **CognitiveIoT** GitHub repository, and download a copy of the `flow1.json` file.
4. Open the file in a text editor. *(Note: Do not use a word processor!)*
5. Copy the entire contents of the file to your clipboard.
6. Import it into your Node-RED flow by clicking **Import > Clipboard** from the menu in the upper right corner.
7. In the Import nodes dialog box, paste it in the text area, and then click **Import**.

8. Click the canvas to fix the flow to it.
9. Click **Deploy**.

**3. Store the device data in cloud storage (a NoSQL database)**

1. In your Node-RED editor, create a new flow by clicking the plus (+) symbol in the upper right of the canvas area.

2. Go to my CognitiveIoT GitHub repository, and download a copy of the `flow2.json` file.

3. Like before, open the file in a text editor, and copy the entire contents of the file to your clipboard.

4. Import it into your Node-RED flow by clicking **Import > Clipboard** from the menu in the upper right.

5. Connect the “limit to max 5000 entries” node to the “washing” node.

6. Click **Deploy**.

Congratulations, you are now streaming IoT data into a Cloudant Apache CouchDB NoSQL database! *(Note: We are limiting the data to 5000 entries to have fast processing later, but you can store up to 1 GB worth of data for free.)*
4. Detect anomalies on the IoT sensor data stream by using the Data Science Experience

One of the simplest anomaly detection algorithms is moving z-score. So we'll implement this by using SQL on Apache Spark. To learn how this algorithm works, we will start with more simple measures like mean and standard deviation. Next, we will turn them into time-series-enabled algorithms by making them calculate moving mean and moving standard deviation and finally coming up with a moving z-score.

The first thing we need to do is create a Python notebook for interacting with the Apache Spark cluster by using Python. Don't worry if you don't know Python very well. Mainly, we are using SQL to issue queries against data that resides in Cloudant Apache CouchDB NoSQL by using Apache Spark SQL.

2. Add a Spark service to your project. Then, create a notebook and select the Spark service. Follow the steps in this tutorial.
3. On the New notebook dialog, in the Name field, enter Cognitive IoT. Then, click From URL, in the Name field add a name. Then, in the Notebook URL field, paste the following URL: https://raw.githubusercontent.com/romeokienzler/developerWorks/master/CognitiveIoT/CognitiveIoT.ipynb
4. Leave the remaining default values, and click Create Notebook.

5. In the notebook, you need to specify the host name, user, and password for the name of your Cloudant Service that you gathered in Step 1.
6. Click in the first text area, and then click the Run button (the toolbar button that looks like a music play button) twice. You should see the first 20 rows in your data set.

7. Follow the steps in the notebook to complete the rest of this step.

5. Transfer the detected anomaly to the real-time data processing component

There are a number of ways to transfer a model obtained in a batch environment to a real-time data processing component. The easiest way is through an HTTP call, which is the way that we’ll implement now. We update a model on the edge by using the ETL component that runs in the cloud. We have to create an HTTP endpoint in our Node-RED flow and then pass this message on by using MQTT to the edge gateway.

1. In the IBM Cloud Dashboard, click View app.

2. Click Go to your Node-RED flow editor.
3. Click Flow 2, which represents the ETL component that runs in the cloud.
4. From the palette on the left, in the input section, select http and drag it to the canvas of Flow 2.
5. Every request needs a response, so connect an **http** response node to it.

6. Double-click the http input node and in the **URL** field type `/edgemodelupdate` and then click **Done**.

7. From the output section of the palette, click **IBM IoT** and drag it to the canvas. Also, connect the **http** node to the **IBM IoT** node.
8. Double-click the **IBM IoT** node and complete these steps:
   a. Select **Bluemix Service** for **Authentication**. This option automatically pulls the credentials for the MQTT message broker from the IBM Cloud (cloud foundry service broker).
   b. Select **Device Command** as **Output Type**. This option is the type to choose if you want to send a message from an application to a device or gateway, which is what we are doing here.
   c. Enter **Washing01** as **Device ID**. We can directly address a receiving device through MQTT because the system uses a publish/subscribe message delivery model.
   d. Enter **modelupdate** as **Command Type**. Because we are using publish/subscribe, defining a command type allows devices to subscribe to specific messages relevant to them.
   e. Enter **json** as **Format** because the central means of data delivery in Node-RED is JSON-based.
   f. Enter **payload** as **Data** property because we are accessing the payload of the message that comes upstream from HTTP.
9. Click **Deploy**.

Now, we can send messages from our analytics workflow by using HTTP to edge gateways by using Node-RED in the cloud as the proxy between HTTP and MQTT. The advantage of using the Watson IoT Platform is that we don't have to worry about reaching the edge gateways directly – which is often not possible – because all the devices are connected to the MQTT message broker and the gateway is subscribed to the messages they are interested in. With this configuration, we can make sure that the required messages reach the edge gateways through the message bus.

### 6. Implement the flow on the Edge gateway

Now we can implement the flow on the edge gateway side by subscribing to those messages and reacting to them by updating a threshold value. For example, in case the current on a machine is abnormal we can shut off the motor to protect it. So let's implement this behavior:

1. Click **Flow 1** because this flow represents the edge gateway.
2. From the palette, drag the **IBM IoT** input node to the canvas.
3. Double-click the IBM IoT node and provide the following details:
   a. Select **Bluemix Service** for Authentication.
   b. Select **Device Command** as Input Type because we are expecting a command to be sent to the edge gateway from the cloud through MQTT.
   c. For **Device Type**, **Device ID**, **Command**, and **Format**, select **All**. For simplicity, we subscribe to all commands on the MQTT message bus, but in a real-life scenario we would subscribe only to commands of our interest and react accordingly.
   d. Click **Done**.
4. From the palette, drag a debug node from the output section to the canvas and connect it to the IBM IoT node.

5. Click **Deploy**.

7. **Test the HTTP endpoint**

We have created an HTTP endpoint, which needs to be tested.

1. In IBM Cloud, copy the URL of your application, and replace "red/#" with "edgemodelupdate". For example, in my case, the URL of the HTTP endpoint is: https://1234thisismyname.mybluemix.net/edgemodelupdate

2. Because we are using an HTTP GET endpoint, we can append a parameter to the URL to issue a request by using the browser (just open a new tab): https://1234thisismyname.mybluemix.net/edgemodelupdate?command=emergencyshutdown

3. Over in our Node-RED flow editor, in the debug panel, you should see a message similar to what is shown in the following screen capture.

Because we are simulating an edge gateway, there are no real actors in place, but in a real-world scenario this might deactivate the motor of the washing machine.

**Conclusion**

You have learned what a complete cognitive IoT pipeline can look like. From distributed sensing, to taking actionable decisions at a central location, to even immediately running an action on the edge. Only two things are missing now: usage of cognitive APIs (such as TTS, STT, or visual recognition) and analytics on the edge.
If you are interested in learning more, join my Coursera course *A developer's guide to Exploring and Visualizing IoT Data*, which helps get you up and running with IoT data analytics by using Apache Spark.

And, if you want to keep up with what I'm doing, subscribe to my [YouTube channel](https://www.youtube.com/).

**One last exercise... before you go...**

Let's conclude this tutorial with an optional exercise. It shows you how to push the z-score calculation down to the edge by using one simple Node-RED function.

We want to push the z-score calculation as close as possible to the sensor. We need to implement it directly on the Node-RED instance on the edge gateway, omitting the complete loop through the cloud. The z-score calculation is done by using a JavaScript function within Node-RED. You can either:

- Implement the nodes in your Node-RED flow yourself, by using the code walkthrough in this section
- Copy the complete function from the `edge_zscore.json` file that you can find in my CognitiveIoT GitHub repository, and paste it to the canvas of your Node-RED flow by using the import from clipboard function.

Let's walk through the source code below to understand what's happening.

First, we initialize a list that stores the last `n` values for the electric current by using the `voltage` parameter. In stream computing, this list is called a "sliding window of fixed size."

```javascript
var aggwindow = context.get('aggwindow')||[];
```

Then, we add values to that list – one on every message that is arriving to this function node.

```javascript
aggwindow.push(msg.payload.d.voltage);
```

We continue doing adding values to the list until we have exceeded 30 values, which basically defines the size of our sliding window.

```javascript
if (aggwindow.length> 30) {
```

<table>
<thead>
<tr>
<th>If you do not understand this code, review these resources on functional programming:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• My Coursera course, <em>A developer's guide to Exploring and Visualizing IoT Data</em>, introduces functional programming on Apache Spark.</td>
</tr>
<tr>
<td>• This developerWorks article, &quot;Use functional programming techniques to write elegant JavaScript,&quot; covers the basics pretty well.</td>
</tr>
<tr>
<td>• To understand map/reduce in JavaScript, check out this article, &quot;Using Map and Reduce in Functional JavaScript&quot;</td>
</tr>
</tbody>
</table>

To compute z-score, we need the mean and the standard deviation. It is always a good idea to calculate the sum of all elements within the window that we want to aggregate over.

```javascript
sum = aggwindow.reduce(((a,b)=>a+b),0);
```
In addition to the sum, we also need the number of elements, so let’s count them. (Note: We should always get 31, but you never know, right?)

\[ n = \text{aggwindow}.\text{length}; \]

With the sum and count values, it is easy to calculate the mean.

\[ \text{mean} = \frac{\text{sum}}{n}; \]

And, with the mean, you can calculate standard deviation.

\[ \text{sd} = \text{Math}.\text{sqrt}(\text{aggwindow}.\text{map}(x=>\text{Math}.\text{pow}(\text{mean}-x,2)).\text{reduce}((a,b)=>a+b,0)); \]

Next, we want to get rid of the oldest element in the list, which resembles a LIFO.

\[ \text{aggwindow}.\text{shift}(); \]

Now, add the values for mean and standard deviation in the formula for the z-score, and we are done. (Note: We are adding a very small value to the standard deviation because the standard deviation can become zero, which is mathematically undefined.)

\[ \text{msg}.\text{zscore} = \frac{\text{mean}-\text{msg}.\text{payload}.d.voltage}{(\text{sd}+0.0001)} \]

Finally, we store this list to a global context to preserve it over individual message lifetimes.

\[ \text{context}.\text{set}('\text{aggwindow}',\text{aggwindow}); \]

Just for debugging, we track the actual current (or voltage) and the z-score at the same time. (We can still access the individual z-score value at \text{msg}.\text{zscore} in case we want to raise an alert.)

\[ \text{msg}.\text{payload} = \text{msg}.\text{payload}.d.voltage+':'+\text{msg}.\text{zscore}; \]

\[ \text{return msg}; \]

Let’s finish this function by raising an alert. We will add a last JavaScript function to the flow. This alert is triggered by an abnormally high fluctuation of the current (either up and down) because it might damage our motor. So, we ignore all messages until we recognize a high anomaly score, and only then we react on it by raising an alert.

\[ \text{if (Math.abs(msg}.\text{zscore})>0.5) \{ \]
\[ \text{msg}.\text{payload} = \text{"ALERT ALERT ALERT!!!!!"}; \]
\[ \text{return msg}; \]

Your Node-RED flow should look like the flow in the following figure.
Related topics

- A beginner's guide to artificial intelligence, machine learning, and cognitive computing
- Watson IoT Platform Analytics
- Hands-on IoT: An introduction to Node-RED (video)
- Coursera: A developer's guide to the Internet of Things (IoT)
- Coursera: A developer's guide to Exploring and Visualizing IoT data
- Watson IoT Platform - From Sensors to Insights

© Copyright IBM Corporation 2017, 2018
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)