BROUGHT TO YOU IN PARTNERSHIP WITH EMQ

Messaging and Data Infrastructure for IoT

Getting Started With IoT Messaging at Scale

TIMOTHY SPANN
DEVELOPER ADVOCATE, STREAMNATIVE

CONTENTS

The Internet of Things (IoT) is a strange term to define software that runs continuously on any number and type of devices. The scale of IoT data is sometimes mind boggling with thousands of devices communicating messages in streams to message brokers that need to instantly receive and often acknowledge those events. The stream of data from devices requires new scalable infrastructure.

The most common protocol utilized for these messages is MQTT, which is lightweight and allows for easy publish-subscribe transfers of small data events. For super-fast messaging, these are commonly implemented in C or Python, and I will show you a short example of this later.

In this Refcard, we will walk through modern messaging and unified data infrastructure to derive business insights from IoT devices. Medium and large enterprises can now build diverse and innovative solutions with open-source IoT platforms. They will be able to filter, route, transform, enrich, distribute, and replicate messages at any scale for real-time analytics.

Once such an infrastructure is in place, any and all data sources can be joined with IoT data, enabling applications never before dreamt of.

IOT CHALLENGES AND SOLUTIONS

Before we begin, we need to look at the challenges of implementing a complex solution for IoT. One of the first challenges is that existing IoT systems are hobbled together by a mix of legacy proprietary systems and poorly connected solutions. The solution is a unified open-source platform that allows for building a modern IoT system that can populate modern data lakes and data warehouses.

Many solutions are hindered by the inability to dynamically scale up and down based on scale. IoT is often very dynamic since the number of devices can drastically change at any time. The solution is a platform that can scale across Kubernetes and clouds quickly to match rapid iterations. Modern messaging systems can scale to handle the number of messages occurring.

IoT is often hindered by fragile legacy protocols that don't work well for streaming real-time data. Modern systems support fast messaging protocols like MQTT and the Apache Pulsar binary protocol specification. These modern systems are asynchronous and support data processing frameworks like Apache Spark and Apache Flink. Pulsar is a unique messaging system that allows for transparent native support for multiple communication protocols, including the Apache Kafka protocol, MQTT, Advanced Message Queuing Protocol (AMQP), and the Apache RocketMQ protocol, all of which are supported in a pluggable non-proxy manner identical to the native binary protocol used by Pulsar. This allows for Pulsar to grow and adapt to new protocols without any performance degradation.

Modern Data Infrastructure for IoT

EMQ provides a modern data infrastructure for your business-critical IoT solutions from edge to cloud.

EMQX Enterprise: The World's # 1 Scalable MQTT Messaging Platform

- 100M concurrent MQTT connections
- 1M/s messages throughput under 1ms
- Extract, filter, enrich, and transform IoT data
- Integrate with over 30 cloud services and enterprise systems

EMQX Cloud: Fully Managed MQTT Service for IoT

- Get your MQTT messaging service up and running in 1 click
- Scale as you need, pay as you go
- Worldwide 24/7 technical support with 99.99% SLA
- Deploy in 15 regions from AWS, GCP and Azure

20K+
Global Users

100M+
Connected IoT Devices

Trusted by 300+ customers from startups to global leaders, across various industries, including Internet of Vehicles, Industrial IoT, Carrier, Finance, Energy, and Smart City.

TELSTRA · Hewlett Packard Enterprise · ERICSSON · CISCO · Verifone · Ruter#

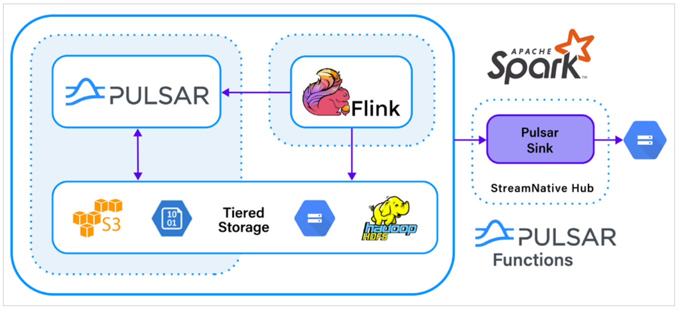
Get started for free

Though this is the case, to provide support for multiple MQTT versions and rich MQTT features, Pulsar is often used with MQTT brokers such as open-source EMQX.

UNIFIED MESSAGING AND DATA INFRASTRUCTURE FOR IOT

For advanced IoT solutions, we need a unified messaging and data infrastructure that can handle all the needs of a modern enterprise that is running IoT applications. In Figure 1, we can see an example of such a system that is built on scalable open-source projects that can support messaging, streaming, and all the data needs of IoT systems.

Figure 1



MESSAGING INFRASTRUCTURE NEEDS

A unified platform that supports IoT messaging needs to support MQTT and multiple streaming protocols. It must be able to handle past, current, and future needs as more devices are brought into solutions. These devices will have updated operating systems, languages, networks, protocols, and data types that require dynamic IoT platforms. These unified IoT messaging platforms will need to allow for extension, frequent updates, and diverse libraries to support the constantly changing and expanding ecosystem.

There are a few key requirements for this infrastructure:

- Low latency
- Flexibility
- Multiple protocol support
- Resiliency
- Geo-replication
- Multi-tenancy
- Data smoothing
- Infinite scalability

LOW LATENCY

In IoT, there are numerous common patterns formed around different edge use cases. For sensors, we often have a continuous stream of small events with little variation. We need the ability to configure for low latency with the option to drop messages for performance since they are mostly duplicates and we only care about the current state. A common pattern is to take samples, averages, or work against aggregates over a short time window. This can often be achieved further down the data chain in data processing via SQL.

FLEXIBILITY

A very important feature that is a minimum requirement for this IoT messaging infrastructure is support for flexibility. This flexibility is multifaceted as the system needs to allow for configurations to change important features such as persistence, message deduplication, protocol additions, storage types, replication types, and more. This may be the most important feature for IoT versus IT systems, with new vendors, devices, and requirements being added for almost every new application or use case.

New devices, sensors, networks, and data types appear from a heterogeneous group of IoT hardware vendors — we need our solution to support, adapt, and work with all of these when they become available. This can be difficult and requires our messaging infrastructure to support multiple protocols, libraries, languages, operating systems, and data subscription types.

Our system will also have to be open source to allow anyone to easily extend any piece of the infrastructure for when unique hardware requires something that doesn't fit into any of our existing pathways. This flexibility is something that can be found only in open-source platforms designed to grow and change. If you don't have the flexibility to meet new requirements based on these new devices, your system will not meet your needs. This could happen at any time. Change is the only constant in the world of IoT — be prepared.

MULTIPLE PROTOCOL SUPPORT

A corollary to flexibility is the support for multiple messaging protocols since no one protocol fits every device, problem, system, or language. MQTT is a common protocol, but many others can be used or requested. We need to also support protocols such as web sockets, HTTP, AMQP, and Kafka's binary messaging protocol over TCP/IP.

RESILIENCY

Most infrastructures for messaging are designed for some level of resiliency within some percentage of tolerance and that works fine for most use cases. This will not be adequate for IoT as workload changes, bad data, and sudden changes to networking, power, scale, demand, and data quality are to be expected. The modern IoT messaging system must bend but never break to these bursts of data and problems.

Along with resiliency, our system must scale to any size required by the applications in question. With IoT, this can start with dozens of devices but then scale to millions in very minimal time. We must not set a fixed or upper limit on how large our clusters, applications, or data can be. We must scale to meet these challenges.

We also need resiliency to survive when IoT devices get hacked or have issues. We need to support as many security paradigms, protocols, and options as feasible. At a minimum, we must enable full end-to-end encryption, full SSL, or HTTPS-encrypted channels. As for encrypted payloads, these will still handle millions of messages with low latency and fast throughput.

We also need to provide logging, metrics, APIs, feeds, and dashboards to help identify and prevent hacks or attacks. The system should allow for machine learning models to be deployed against queues and topics dynamically to intercept hacked messages or intrusion attempts.

Security should never be an afterthought when real-time systems interacting with devices and hardware are involved.

GEO-REPLICATION

Distributed edge computing is a fancy way of saying that we will have our IoT devices deployed and running in multiple areas in the physical and networking world. This requirement to get data to where it needs to be analyzed, transformed, enriched, processed, and stored is a key factor in choosing the right messaging platform for your use case.

To achieve this, one important and required feature is support within the platform for geo-replication between clusters, networks, availability zones, and clouds. This will often require active-active two-way replication of data, definitions, and code. Our solution has no choice but to support this data routing at scale.

As users of IoT expand, they quickly start deploying across new cloud regions, states, countries, continents, and even space. We need to be able to get data around the globe and often duplicate it to primary regions for aggregations. Our messaging infrastructure needs to be able to geo-replicate messaging data without requiring new installations, closed-source additions, or coding.

The system will need to replicate via configuration and allow for any number of clusters to communicate in both active-active and active-passive methods. This is not without difficulty and will require full knowledge of the systems, clouds, and security needed for all regions. Again, this is something that can only be achieved with open source.

MULTI-TENANCY

If we only had one IoT application, use case, company, or group working with the system, then things would be easy, but this is not the case. We will have many groups, applications, and often companies sharing one system. We require secure multi-tenancy to allow for independent IoT applications to share one conduit for affordability, ease of use, and scale.

DATA SMOOTHING

In some edge cases, all the data sent is time series data that needs to be captured for each time point — this could arrive out of sequence and the messaging system needs to be able to handle this.

In messaging systems like Kafka or Pulsar, you may want to handle this as an exactly once messaging system so that nothing is duplicated or lost before it lands in durable permanent storage, such as an AWS S3-backed data lake. The advantage of modern data streaming systems is that missing data can be resent, waited on, or interpolated based on previous data streams. This is often done by utilizing machine learning algorithms such as Facebook's Prophet library.

INFINITE SCALABILITY

Often in IoT, we need to rapidly scale up for short periods of time and then scale down, such as for temporary use cases like deployments of many sensors at sporting events. In some instances, we may need this to happen with no notice and on demand.

The ability to do this scaling without tremendous expense, difficulty, or damaging performance is a feature required by the needed infrastructure. This can often be done by using a messaging platform that runs natively on Kubernetes and has separate compute and storage of messaging.

IoT workloads can happen in large bursts as thousands of devices in the field can become active at once in, say, energy meters, and we need to be able to survive these bursts. These bursts should drive intelligent scaling, and where delays occur to infrastructure availability, we must be able to provide caching, tiered storage, and backpressure to never break in the face of massive torrents of device data.

We also need to cleanly shut down and remove extra brokers when the storm has subsided, and we can reduce our infrastructure costs cleanly and intelligently.

We have gone through a very tall order on what this magical, scalable messaging system must do. Fortunately, there are open-source options that you can investigate today for your edge use cases.

DATA INFRASTRUCTURE REQUIREMENTS

A new but important part of building IoT solutions is a modern data infrastructure that supports all the analytical and data processing needs of enterprises today. In the past, there was a great disconnect between how IoT systems data was handled and that of the rest of the IT data assets. This disconnect was driven by the differences in infrastructure used by IoT and other use cases. With a modern unified messaging infrastructure bridging the gap between systems, we no longer face those differences. Therefore, we must now update what systems our IoT data is fed into and how.

In the modern unified data infrastructure required for all use cases, including IoT, we must support some basic tenants.

DYNAMIC SCALABILITY

A key factor in handling the diverse workloads and bursty nature of IoT events is the support for dynamic scalability. This usually requires that the messaging system runs on Kubernetes and allows for scaling up and down based on metrics and workloads.

CONTINUOUS IOT DATA ACCESS

As soon as each event from any device enters the messaging system, we need to be able to get access to it immediately. This allows us to run aggregates and check for alerting conditions. This is often achieved with a real-time data-processing streaming engine, such as Apache Flink, connected to the event stream.

This goes with the standard requirements of robust library support for all the modern open-source data libraries such as Apache Spark, Apache NiFi, Apache Flink, and others. We also would want robust support by cloud data processing like Decodable.

SQL ACCESS TO LIVE DATA

Along with continuous data access via a programmatic or API, we need to allow analysts and developers to utilize SQL to access these live data streams as each event enters the system. This allows for aggregates, condition checks, and joining streams. This may be the most important feature of a modern messaging system to support IoT. Without SQL, the learning curve may reduce adoption within the enterprise.

OBSERVABILITY

If the events don't arrive, the system slows down, or things go offline, we must know not only instantly but also preemptively that things are starting to go astray. A smart observability system will trigger scaling, alerts, and other actions to notify administrators and possibly stop potential data losses. This is often added as an afterthought, but it can be critical. We also need to be able to replay, fill in missing data, or have data rerun.

SUPPORT FOR MODERN CLOUD DATA STORES AND TOOLS

Our IoT events must stream to any of the cloud data stores that we need it to. This may be as simple as a cloud object store like AWS S3 or Apache Pinot — or as advanced as a real database — and is a minimum requirement that cannot be skipped. We need our IoT events to be in the same database as all our other main data warehouse datasets. We need to support open data formats like Apache Parquet, JSON, and Apache AVRO.

HYBRID CLOUD SUPPORT

Finally, we need to be able to run our messaging architecture across various infrastructure hosting locations, including on-premises and multiple public clouds. The ability to be installed, run anywhere, and propagate messages is key.

GETTING STARTED: AN IOT ARCHITECTURE EXAMPLE

Now that we have looked at what we need, let's build an example of the architecture to accomplish all these requirements for an actual IoT use case. I have included two ways to get started with the most common open-source options for messaging infrastructure for IoT at scale: Apache Pulsar for streaming and EMQX as an MQTT broker.

As a quick start, you can install a standalone broker or docker instance of EMQX. You can download the latest stable release from the project's GitHub release directory here: https://github.com/emqx/emqx/releases

Recommended next steps:

- Read the EMQX and Documentation before installation
- Start on your laptop and test with one real device
- Go to production in batches of devices and test the results

If you wish to install the EMQX system on a Linux system, you can follow the reference guide here.

```
docker run -d --name emqx -p 1883:1883 -p 8083:8083
-p 8084:8084 -p 8883:8883 -p 18083:18083 emqx/
emqx:latest
```

An Apache project option is Apache Pulsar. The standalone version of Apache Pulsar will require a modern operating system such as Linux or Mac and a JDK version of 8 or higher. I recommend using JDK 17, as this has the greatest performance and features that enable a fast and stable system.

```
wget https://archive.apache.org/dist/pulsar/pulsar-
2.10.0/apache-pulsar-2.10.0-bin.tar.gz
tar xvfz apache-pulsar-2.10.0-bin.tar.gz
cd apache-pulsar-2.10.0
```

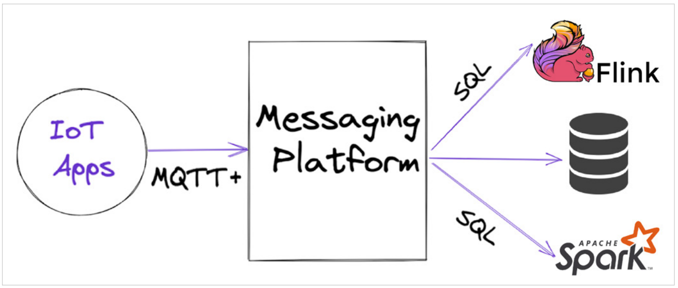
To get started with Apache Pulsar in Docker, you will need enough RAM and CPU as well as have Docker installed. I recommend at least eight gigabytes of RAM and four virtual CPUs. Once you meet these requirements, you can begin running your Apache Pulsar messaging broker on your machine:

```
docker run -it -p 6650:6650  -p 8080:8080 --mount
source=pulsardata,target=/pulsar/data --mount
source=pulsarconf,target=/pulsar/conf apachepulsar/
pulsar:2.10.0 bin/pulsar standalone
```

You can now build a sample IoT application and try out various examples via the standard documentation. I suggest sending and receiving a few messages first to make sure everything works. Also check logs, metrics, dashboards, and other tools listed for the respective messaging tools. If messages are appearing as expected, you are ready to start using simulated devices or a Raspberry Pi.

I have a simple Python example that you can use to start with. It generates standard JSON, which is a common and easy-to-work-with format for sensor applications. There are many affordable sensors available for the Raspberry Pi, and I linked a few examples at the end of this Refcard.

Figure 2



A use case that comes up with enterprises quite often is that of an IoT implementation with thousands of devices streaming data over MQTT and Pulsar into multiple topics. From here, our system makes data asynchronously available for subscribed data consumers. These feeds are continuously consumed by Flink SQL and auto-sync to a data lake, data stores, and fast Spark ETL. There are often data consumers that continuously update real-time dashboards as seen in InfluxDB.

A common way to process, especially when we are receiving millions of events a second, is to store them to a data lake. We can build a simple and fast ETL for this utilizing Apache Spark's Structured Streaming framework with some easy-to-write and deployable code. A simple Scala snippet for this code is shown below.

In this two-line code snippet, we read from a Pulsar topic containing our IoT sensor data, then select all the fields and store the data in Apache Parquet files in an AWS S3 object store. This can be substituted for other common data lake formats and object stores depending on the cloud or Kubernetes platform that you are running on.

```
val dfPulsar = spark.readStream.format("pulsar").
option("service.url", "pulsar://server:6650").
option("admin.url", "http://server:8080").
option("topic", "persistent://public/default/
topic").load()
val pQuery = dfPulsar.selectExpr("*").writeStream.
format("parquet").option("truncate", false)
.option("checkpointLocation", "/tmp/checkpoint").
option("path", "/s3").start()
```

If you wanted to run continuous SQL statements against a large stream of IoT messages, you could do this with SQL:

```
select max(equivalentco2ppm) as MaxCO2,
max(totalvocppb) as MaxVocPPB from topicname;
```

Using Flink SQL, a common use case is to aggregate our data over time windows and capture the result to any number of cloud data stores or to additional topics for further analytics or machine learning.

On a typical device, we could easily send our IoT messages via MQTT in Python with two short lines of code. The first connects to an MQTT broker and then publishes a message to a queue. In this example, the message is defined by `json_string`:

```
client.connect("broker", 1883, 180)
client.publish("persistent://public/default/queue",
payload=json_string, qos=0, retain=True)
```

We could also produce messages to MQTT brokers via Java:

```
MemoryPersistence persistence = new
MemoryPersistence();
IMqttClient mqttClient = new MqttClient("tcp://
host:1883", clientId, persistence);
mqttClient.connect(mqttConnectOptions());
MqttMessage mqttMessage = new MqttMessage();
mqttMessage.setPayload(payload);
mqttClient.publish("queuename", mqttMessage);
```

There are many other languages supported by MQTT libraries, and you should check the documentation for one that matches the development languages available on your device.

An advanced feature of one implementation of a scalable broker is support for standard SQL via Presto. This allows you to use simple SQL statements as your data consumer. This can be done from any notebook, application, query tool, or system that supports JDBC, ODBC, or SQL Alchemy driver.

```
select * from pulsar."public/default".iottopic;
```

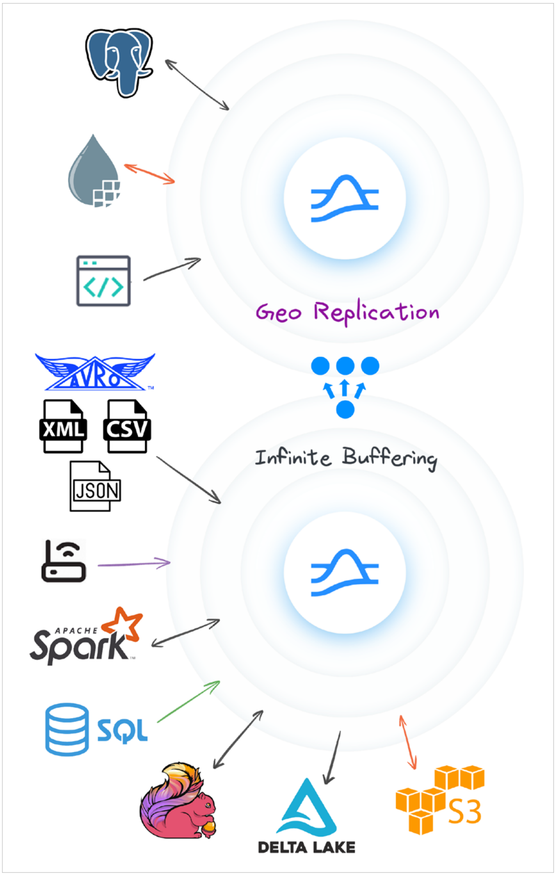
For SQL access to EMQX, you can use the SQL Rule Engine:

```
SELECT * FROM "#" WHERE field = 'valuex'
```

CONCLUSION

We found that to scale an IoT solution, you need open-source messaging and data infrastructure. We've found there are only a few platforms, including Apache Pulsar and EMQX, that have all the features to support solving these complex IoT problems. As we iterated through a long list of requirements, we have determined how to implement these in the real world. You can now be assured that these modern data messaging systems will handle your IoT workloads.

Figure 3



For example, geo-replication, as mentioned, is very important to propagate messages between various infrastructure facilities, which commonly occur in distributed edge technologies like IoT.

We also realized that without support for open-source data processing frameworks, our data will be sitting idle in formats we can't access, locking our data away from our analysts. At every need and every level, we should keep things as open, flexible, and scalable as possible to ensure our applications function at the necessitated degree and provide the analytics required by modern enterprises.

For more information on building out your messaging infrastructure, see the following references that include examples and further reading.

ADDITIONAL RESOURCES

- "FLiP-Pi-Weather" – https://github.com/tspannhw/FLiP-Pi-Weather
- "FLiP-Py-Pi-GasThermal" – https://github.com/tspannhw/FLiP-Py-Pi-GasThermal
- "FLiP-Py-Pi-EnviroPlus" – https://github.com/tspannhw/FLiP-Py-Pi-EnviroPlus
- "FLiP-Pi-BreakoutGarden" – https://github.com/tspannhw/FLiP-Pi-BreakoutGarden
- "Data Management for Industrial IoT" Refcard – https://dzone.com/refcardz/data-management-for-industrial-iot

WRITTEN BY TIMOTHY SPANN,

DEVELOPER ADVOCATE, STREAMNATIVE

Tim Spann is a developer advocate for StreamNative. He works with StreamNative Cloud, Apache Pulsar, Apache Flink, Apache NiFi, MQTT, AMQP, Apache Kafka, Edge AI, TensorFlow, Apache Spark, InfluxDB, Aerospike, ElasticSearch, Lakehouses, and deep learning. Tim has over a decade of experience with the IoT, big data, distributed computing, messaging, streaming technologies, and Java programming.