AWS OSI Pi Integration Library

OSI Pi Real Time Data Connector

Developers Guide

Nov 2023
V0.0.07

Dean Colcott
AWS IoT / Industrial Data Specialists Solution Architect.

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided “as is” without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

Table of Contents

AWS OSI Pi Integration Library

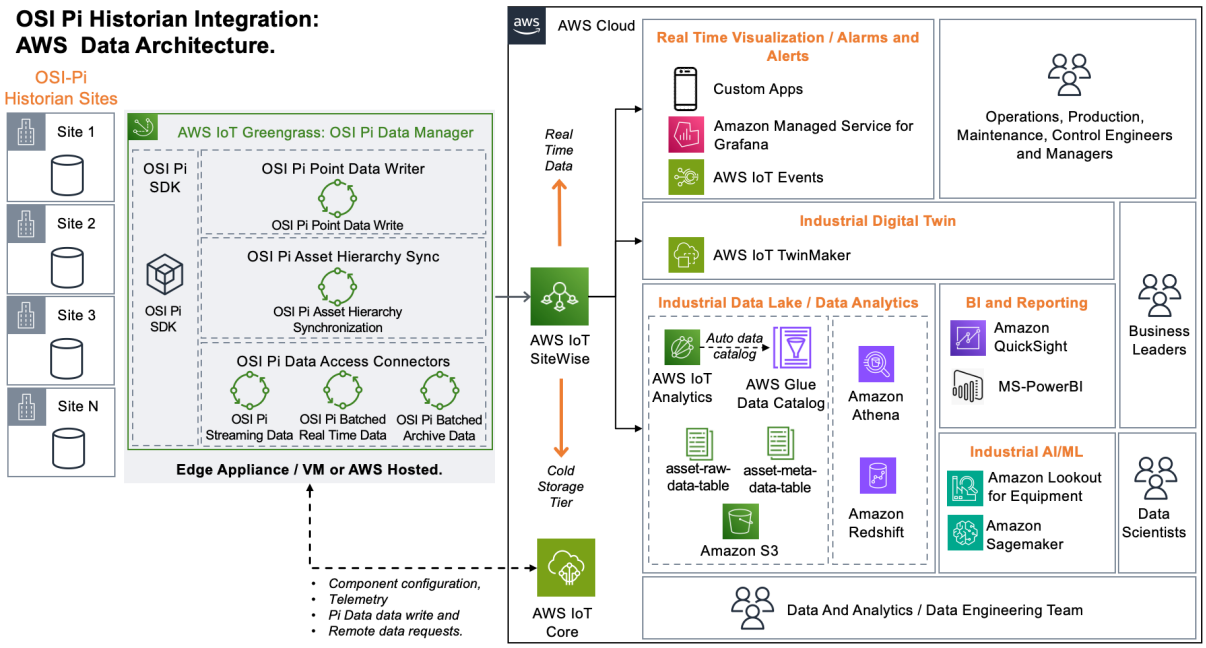
The AWS OSI Pi integration library consists of a series of data connectors and control plane integrations to the OSI Pi Data Historian and Asset Framework into an AWS modern data architecture.

The AWS OSI Pi Integration Library interfaces to OSI PI applications to provide:

- Enterprise-wide real time data visualisations, alarms, alerts and notifications,
- A modern data architecture on AWS for deeper insights of operational data,
- Integrations to Business Intelligence and Reporting Tools for greater visibility of industrial assets,
- Digital Twin solutions,
- AI/ML and Predictive maintenance outcomes and
- Closed loop industrial automation.

The OSI Pi Real Time Data Connector described in this guide is one of the above-mentioned data connectors. The components of the AWS OSI Pi Integration Library are deployed and managed on the edge by AWS IoT Greengrass.

AWS OSI Pi Integration Library Architecture Diagram:



The OSI Pi Integration library is an AWS IoT managed distributed application that allows the user to remotely select OSI Pi data points for ingestion to AWS via the centralized AWS IoT Core. This pattern enables centrally managed load balancing of OSI Pi data point ingestion using low cost, horizontally scalable commoditized edge hardware. The simple demonstration below shows a Raspberry Pi4 ingesting over 7000 Pi data points per second using the OSI Pi Integration library.

Scaling horizontally using low-cost edge data collectors and centralized OSI Pi data point load balancing builds a highly available, scalable, redundant and cost-effective distributed industrial data management solution.

AWS OSI Pi SDK

At the core of the AWS OSI Pi Integration Library is the AWS OSI Pi SDK. This library provides control plane integration to the OSI Pi Asset Framework and the Data Archive server. The AWS OSI Pi SDK is imported into each of the components of the AWS OSI Pi Integration Library providing them with a common set of base functionality, implementation and configuration.

OSI Pi Data Connectors and Components

The AWS OSI Pi Integration Library provides the following connectors and components:

1. **OSI Pi Asset Framework Synchronisation Component:** Used to synchronise the OSI PI asset Framework to the AWS IoT Sitewise Asset Hierarchy.
   Status: Dev in progress.

2. **OSI Pi Streaming Data Connector:** The focus of this developer guide, the OSI Pi Streaming Data Connector streams real-time (on-change) data from OSI Pi over Secure WebSocket (wss://) connections to the Pi Server. This connector should be used to ingest OSI PI data points where low-latency is required. Load on the OSI Pi server is proportional to the number of PiPoints per second being ingested.
   Status: First release complete.

3. **OSI Pi Batch Data Connector:** This connector reads in archive data from a defined start date/time through to continuously poll the live edge of the incoming data via REST'ful polling. The latency experienced is a function of the configured polling rate and the number of data points requested. In this way, load on the OSI Pi server is decoupled from the number of PiPoints selected for ingestion and so can be tuned for latency, scale or Pi Server load. Use this connector for high scale data archive ingestion and / or when load on the OSI Pi server needs to be more closely managed.
   Status: Final testing – Awaiting Sitewise Feature Release.

4. **OSI PiPoint Data Writer:** This connector adds the ability to write back to the OSI Pi data server from an AWS IoT Greengrass managed component. While it can be used to upload data via AWS IoT MQTT messages, its intended purpose is for closed loop automation where ingested OSI Pi data triggers an automated response from application logic, business rules and state machines implemented in AWS.
   Status: First release complete - Under internal security review.

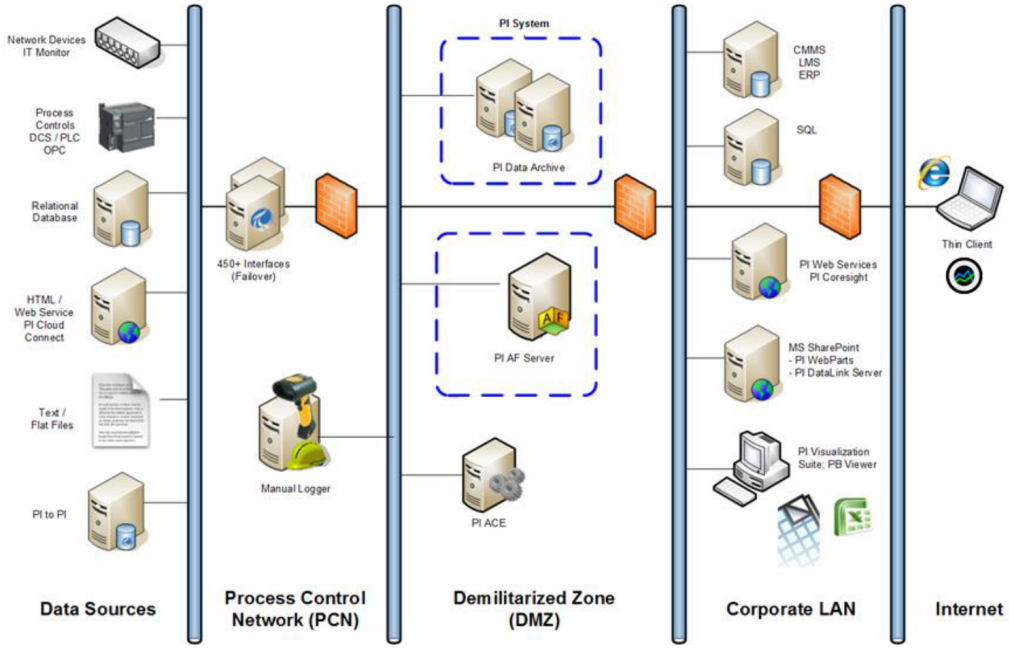
OSI Pi System Overview

OSI PI (Now Aveva) is an industrial data historian for collecting, enriching, storing, and accessing industrial data. Analysist indicate that elements of the PI System are deployed at more than 20,000 sites worldwide, have been adopted by over 1,000 power utilities and manage data flows from close to 2 billion real-time sensors.

The OSI Pi systems consists of a number of software and data management solutions, primarily the OSI Pi Data Archive and the OSI Pi Asset Framework. The OSI Pi Data Archive is a proprietary industrial time-series database that manages data points that are referred to as PiPoints, data tags (or just tags). These data tags are typically generated from sensors and physical assets. A large Pi Data Archive can maintain data for over a million data tags.

The Asset Framework allows industrial control engineers to manage asset inventory and hierarchy, data point Unit of Measures (UoM) and other relevant data point meta-data. The Asset Framework server is backed by MS-SQL. Asset entries in the Asset Framework can maintain a reference to a data tag in the Pi Data Archive that adds meaning to the otherwise non-contextualised timeseries data.

The OSI Pi system is deployed on the Window Server operating system and provides a number of interfaces and integrations to access both the Pi Asset Framework and the Pi Data Archive. These include OLDBC / SQL connectors, .NET based SDK (AF-SDK) and a REST'ful web interface via the Pi WebAPI. This OSI Pi Streaming Data connector interfaces to the Pi WebAPI.



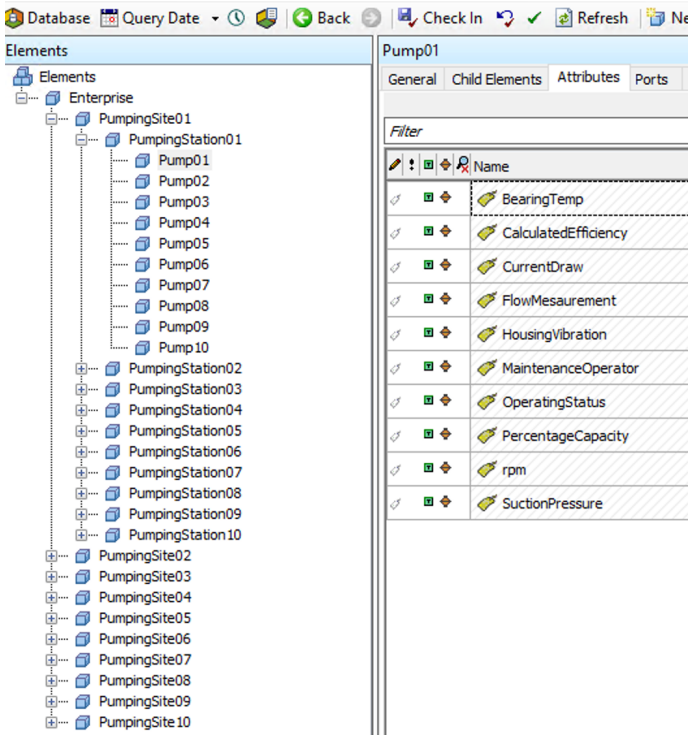
Ref: http://cdn.osisoft.com/learningcontent/pdfs/BuildingPISystemAssetsWorkbook.pdf

OSI Pi Server Test Environment

In the examples given throughout this document, we are testing against a hypothetical set of Pumping Sites / Stations with an OSI Pi Asset Database structure of:

- Enterprise (common parent element)
    - 10 x Pumping Sites (PumpingSite01 – 10) with:
        - 10 x Pumping Stations (PumpingStation01 – 10) with:
            - 10 x Pumps (Pump01 – 10) with:
                - 10 x Attributes:
                    - Bearing Temperature
                    - Calculated Efficiency
                    - Current Draw
                    - Flow Measurement
                    - Housing Vibration
                    - Maintenance Operator
                    - Operating Status
                    - Percentage Capacity
                    - RPM
                    - Suction Pressure

In this test set up, each Pump attribute has an associated Pi Data Archive PiPoint giving a total of 10K individual PiPoints. Each of these have an associated calculation generating a data point at 1Hz (once per second) and so 10K data points per second are being generated and tested. These data points have a distribution of 60% doubles, 20% integers, 10% Boolean, and 10% Strings.



AWS OSI Pi Streaming Data Connector

The AWS OSI Pi Streaming Data Connector provides real-time data ingestion from an OSI Pi Data Historian into an AWS Modern Data Architecture. The connector is an AWS IoT Greengrass component that integrates to the OSI Pi REST'ful web interface provided by the OSI Pi WebAPI to access and ingest real-time streaming data over WebSocket's into AWS IoT Sitewise.

OSI Pi data points (known as PiPoints) can be dynamically enabled or disabled for live streaming:

- Individually by requesting the PiPoints from the OSI PI Data Archive,
- In bulk by the Pi Tag path (with wildcard) (i.e: //server/database/location/asset/*) or
- From the Pi Element representing an asset in the OSI Pi Asset Framework.

The OSI Pi Streaming Data Connector's primary purpose is to ingest real-time data from OSI Pi Historians to AWS IoT Sitewise. End-to-end latency is in the order of single digit seconds.

When using the OSI Pi streaming data connector, load on the OSI Pi server is proportional to the number of data points per second being ingested. When using the batched connector, load is a function of the configurable API requests per second against the OSI Pi server. In this case, scale is traded off against latency which in some cases is more desirable. The Batch connector can also request archive data with guaranteed delivery whereas the Streaming Data connector will lose real time data if the connection to the Pi Server is lost.

A recommended design pattern is to deploy both this Streaming Data Connector for selected low-latency Pi data points and batch data connectors for bulk and archive data ingestion.

OSI Pi Streaming Data Connector Configuration

The OSI Pi Streaming Data Connector has static and dynamic configurations. Static configurations such as the OSI PI server URL and user credentials are managed via an AWS IoT Device Shadow document. Dynamic parameters such as the OSI PI data points (PiPoints) to stream to AWS IoT Sitewise are enabled / disabled over a defined MQTT PubSub (API like) interface presented to the AWS IoT Core. These are both detailed in following sections.

The static config parameters maintained in the IoT Shadow Document can be updated at any time during use of the OSI Pi Streaming Connector:

- If changing AWS IoT Sitewise buffering or OSI PI rate limiting values, these will be applied immediately with no loss of any active streaming data points,
- If updating the AWS region, OSI Pi server URL or root path, credentials, authentication or certificate verification mode then the connector will be reinitiated to its initial default state. All PiPoints configured for streaming will be disabled.

These settings are described in more detail in following sections.

OSI PiPoint Data (WebSocket) Channels

The OSI Pi streaming data connector requests real-time data by initiating WebSocket requests to the OSI Pi server. The WebSocket URL parameters inform the OSI Pi server which PiPoints should be streamed over the WebSocket. The OSI Pi streaming data connector batches groups of 100 PiPoints (by default) into a single WebSocket request to the OSI Pi server. In this system, we call these **Channels**. A channel represents the WebSocket connection and its state as well as the associated PiPoints or asset attributes that have been requested with that WebSocket.

Each channel is given a Chanel ID for reference in the PubSub Messaging Interface. The OSI Pi streaming data connector manages the creation of channels based on the users requested streaming PiPoints.

Initiating the streaming PiPoints is a two-step process. First, the PiPoints are queued for processing by requesting PiPoint or Pi Asset Attributes with tag and name filters via the PubSub interface. Once the required points are queued, they are activated by a similar PubSub call. This two-step process allows better control of how data points are aggregated into individual WebSocket channels.

For example, using the Pumping Site asset structure previously described:

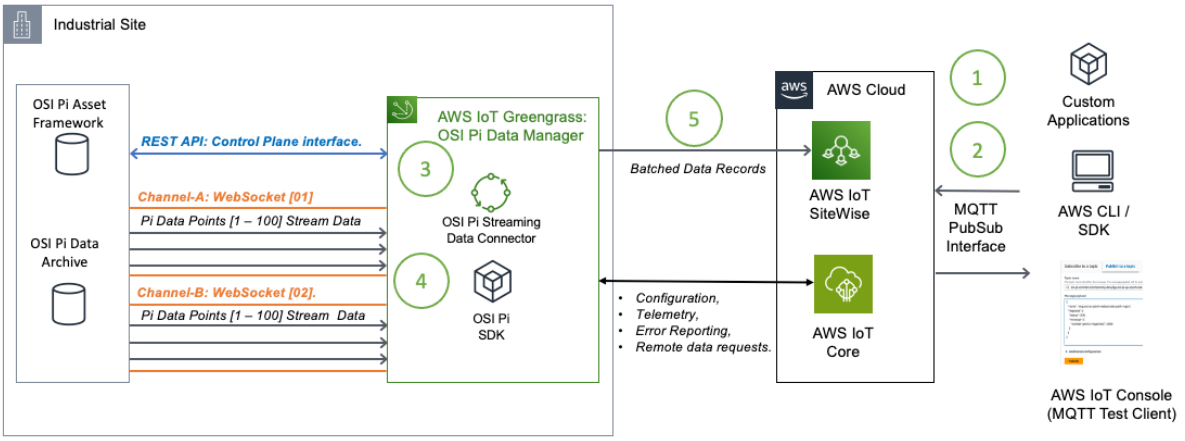
1) If requesting Pi tag paths “PumpingSite01*” it would request 1000 individual Pi tag paths which would be queued for processing. Once the activate command is called, the OSI Pi streaming data connector would then create 10 WebSocket channels with 100 PiPoints each and begin streaming data for these tags.

2) However, if you wanted all tags for specific asset such as Pump01 to be allocated to a single WebSocket channel then follow the same procedure to queue PiPoints using the more specific filter “PumpingSite01.PumpinStation01.Pump01*”and activate the queue. In this case there are only 10 PiPoints in the queue which will be assigned to a single WebSocket channel.

The requested PiPoints are queued in this way to allow assets to be associated to the same channel or to bulk activate and let the connector aggregate as needed. However, if allocating small numbers of PiPoints to an individual WebSocket channel, it’s possible to exceed WebSocket limits on the connector and the OSI PI server.

The workflow used to enable streaming selected OSI Pi data points from the OSI PI server to AWS Sitewise via the OSI Pi Streaming Data Connector is shown below:

OSI Pi Streaming Data Connector Workflow:



1) User (optionally) polls the OSI Pi WebAPI Via the AWS OSI Pi SDK PubSub Messaging Interface for number of tags and tag paths that a given PiPoint request filter returns.
2) User requests OSI Pi streaming data connector to queue tags that match a filter (i.e: ‘tag:=PumpingSite01*’)
3) OSI PI SDK polls the OSI Pi server control plane for WebIDs of matching tags and queues them locally.
4) User requests OSI Pi Streaming Data Connector to activate queued PiPoint Channels with 100 PiPoints per WebSocket which begins streaming data to the connector.
5) The OSI Pi Streaming Data Connector batches and uploads the OSI Pi data to AWS IoT Sitewise.

Memory Management and Sitewise TQV Rate Limiting.

The OSI Pi Streaming Data connector batches ingress OSI Pi data into AWS IoT Sitewise Bulk Publish actions to reduce the number of Sitewise messages required which is more efficient and reduces AWS consumption charges.

If Sitewise becomes unreachable, or OSI PI is ingesting more data points than the configured quota for AWS IoT Sitewise in the connectors Shadow document, the connector will buffer the data locally. At this point the connector will to buffer until 50% of the processes allocated memory is consumed. After which, it will continue to try and upload data to Sitewise but will not accept any new data form OSI Pi until the consumed memory is reduced.

OSI Pi Streaming Data Connector PubSub Interface

The OSI Pi Streaming Data Connector is remotely managed using an API like PubSub messaging interface over MQTT that is (typically) signalled via the AWS IoT Core. This interface is based on the AWS OSI Pi SDK library and so is common in format and functionality across all AWS OSI Pi Integration Library components.

PubSub Topic Schema:

A defined PubSub topic schema is provided to reduce dependencies across distributed application teams and otherwise simplify the PubSub messaging interface. These topics are referred to as the **control topics** as they manage control messages to and from the OSI Pi Streaming Data Connector:

Control Request Topic:
osi-pi/streaming-data-connector/[IOT_THING_NAME]/ingress

Control Response Topic:
osi-pi/streaming-data-connector/[IOT_THING_NAME]/egress

Async Update Topics:
The following topics report asynchronous telemetry and state changes as described in Appendix B: Asynchronous PubSub Message Updates.

osi-pi/streaming-data-connector/[IOT_THING_NAME]/telemetry
osi-pi/streaming-data-connector/[IOT_THING_NAME]/websocket-state/opened
osi-pi/streaming-data-connector/[IOT_THING_NAME]/websocket-state/errored
osi-pi/streaming-data-connector/[IOT_THING_NAME]/websocket-state/closed
osi-pi/streaming-data-connector/[IOT_THING_NAME]/websocket-state/failed-deleted

Request / Response MQTT Message Format:

Requests to the OSI Pi Streaming Data Connector for control plane access to the OSI Pi system or to enable and disable streaming OSI Pi data points are via a common Request / Response message format.

MQTT Request Format:

```
{
  "route": "[REQUEST-COMAND-TO-TRIGGER]",
  "params": {
    "param01":"optional request param01",
    "param02":"optional request param02"
   }
}
```

A number of '**route'** commands are supported by the OSI Pi Streaming Data Connector. The route field is comparable with a REST API path and is used to identify the command to execute on the connector. Param's are added to the request for values that are specific to the route.

A full list of supported routes and their function is given in **Appendix A: PubSub Messaging Interface.**

MQTT Response Format:

```
{
  "id": 10,
  "route": "[REQUEST-ROUTE]",
  "status": [1XX || 20x || 40x || 50x],
  "response": {
    "param01": "[OPTIONAL RESONSE MESSAGE / OBJECT OR ERROR DETAILS]",
    "param02": "[OPTIONAL RESONSE MESSAGE / OBJECT OR ERROR DETAILS]"
  }
}
```

The response adds an incrementing (message) ID to each message to simplify deduplication of received messages and a status code that follows HTTP standards is given. The route used in the request is reflected in the response for identifying the message command.

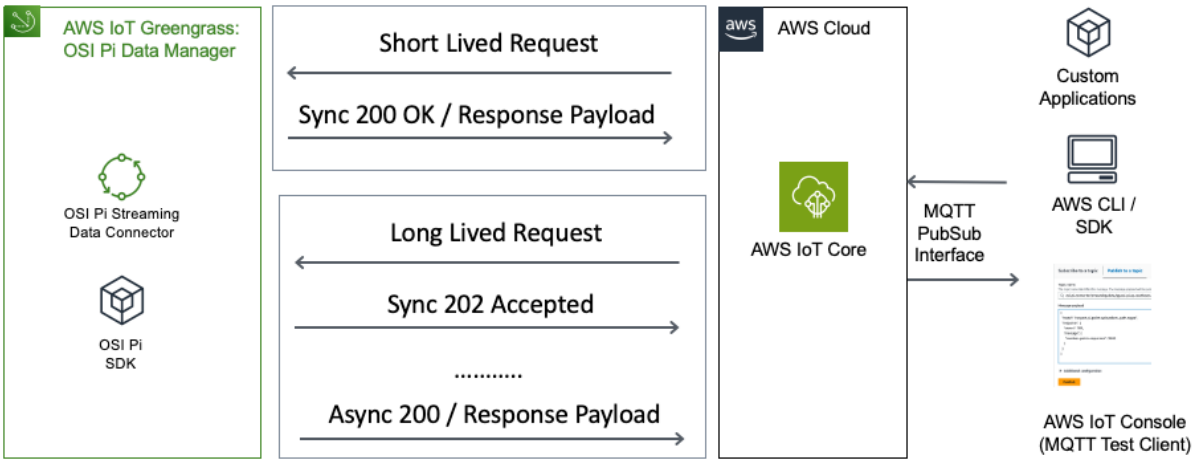
Within the response field, an optional message string or object is added to provide the response values or any error details.

PubSub Topic Message Patterns:

PubSub messages sent to the control topic provide an internally blocking Request / Response message flow. While PubSub messages are inherently asynchronous, this is to emulate a synchronous interface and operation over MQTT.

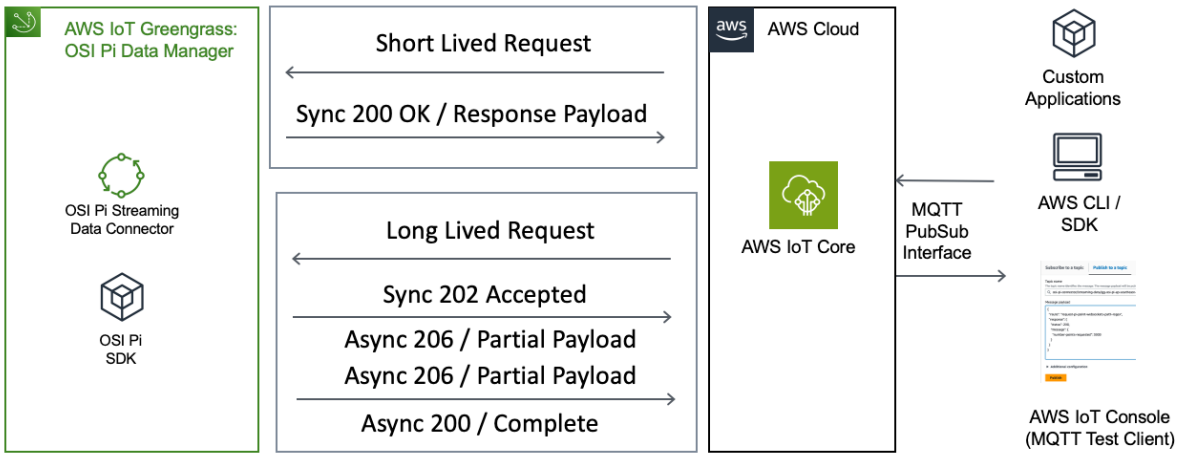
Any request that can be served in this way will respond with a status code 200 and the response values in the MQTT message payload. Requests that require a longer wait period will respond with a status code 202 (Accepted) to acknowledge the request and will send an asynchronous 200 code with the requested payload once available as shown in the following diagram.

OSI Pi Streaming Data Connector PubSub Sync / Async Message Patterns:



OSI Pi Streaming Data Connector PubSub Partial Content Message Patterns:

Where a response isn't likely to fit in a single MQTT message, the content received from the OSI Pi server is segmented and returned with a sequence of status code 206 Partial Payload messages. These each contain the start index and number of elements in each message. Once no more elements remain, a status 200 OK / Complete is returned to close the request.



Example:
Requesting all OSI Pi Elements in Asset Framework request

MQTT Request:

```
{
  "route": "publish-pi-asset-elements-by-query",
  "params" : {
    "databaseWebId" : "F1RDA4FInZtev0y51vYd6bRwFAUD-8-
fACEkOR2eBwn13jJgRUMyQU1BWi01QjFCNFJMXFBJV0VCQVBJU0NBTEVURVNU",
     "queryString" : "name:=*"
  }
}
```

MQTT Response: 202 Request Ack

```
{
  "id": 905,
  "route": "publish-pi-asset-elements-by-query",
  "status": 202,
  "response": "request-ack-processing"
}
```

MQTT Response: 206 Partial Payload Response Messages x N:


```
{
  "id": 906,
  "route": "publish-pi-asset-elements-by-query",
  "status": 206,
  "response": {
    "numberPiAssetElements": 250,
    "startIndex": 500,
    "queryString": "name:=*",
    "piAssetElements": [
      {
        "webid": "F1EmA4FInZtev….lPTjA5XFBVTVAwNQ",
        "name": "Pump05",
        "path": "\\\\EC2AMAZ-
123456\\PiWebApiScaleTest\\Enterprise\\PumpingSite10\\PumpingStation09\\Pump05",
…
[OUTPUT REMOPVED FOR BREVITY]
}
```


MQTT Response 200: Request complete with total Items Returned Count:


```
{
  "id": 911,
  "route": "publish-pi-asset-elements-by-query",
  "status": 200,
  "response": {
    "databaseWebId": "F1RDA4FInZtev0y51vYd6bRwFAUD-8-
fACEkOR2eBwn13jJgRUMyQU1BWi01QjFCNFJMXFBJV0VCQVBJU0NBTEVURVNU",
    "queryString": "name:=*",
    "itemsReturned": 1111
  }
}
```

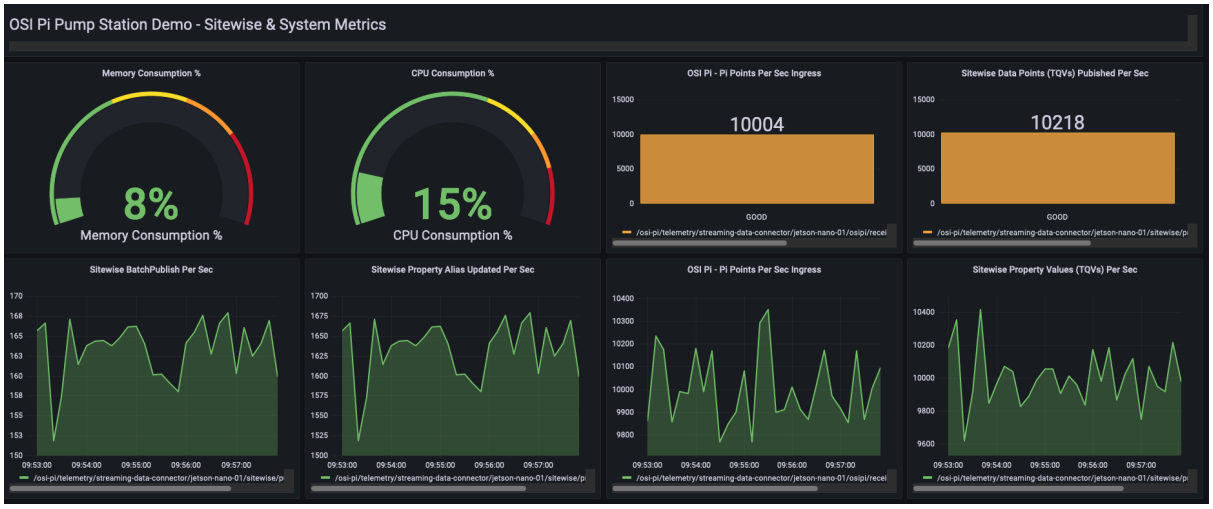
Asynchronous Messages:

There are also purely asynchronous events that need to be reported such as an OSI Pi sever WebSocket changing state, the component itself becoming active, AWS IoT SiteWise and the components memory usage telemetry. An asynchronous OSI Pi Streaming Data Connector telemetry message example is shown below:


```
{
  "id": 170,
  "route": "telemetry",
  "status": 200,
  "response": {
    "device": "gg-osi-pi-ap-southeast-2-amz-linux",
    "timestamp": 1687245208649,
    "sitewise": {
      "publishPerSec": 100.7,
      "propAliasPerSec": 1007,
      "propValuesPerSec": 10070,
      "queuedPropAlias": 10000,
      "propValuesPerAlias": 10,
      "publishErrorCount": 7,
      "publishErrorReceived": {
        "ThrottlingException": "You've reached the maximum rate of
BatchPutAssetPropertyValue entries ingested per asset-property/data-stream."
       }
    },
    "osipi": {
      "receivedPiPointsPerSec": "9790.00"
    }
    "system": {
      "total_heap_size": 92651520,
      "total_heap_size_executable": 2883584,
      "total_physical_size": 88357424,
      "total_available_size": 1075917960,
      "used_heap_size": 46416336,
      "heap_size_limit": 1124073472,
      "malloced_memory": 532552,
      "peak_malloced_memory": 3167448,
      "does_zap_garbage": 0,
      "number_of_native_contexts": 1,
      "number_of_detached_contexts": 0,
      "total_global_handles_size": 81920,
      "used_global_handles_size": 71168,
      "external_memory": 5895591,
      "cpuUsage": {
        "user": 74847955,
        "system": 4482616,
        "percent": 7
      }
    }
  }
}
```


This message can be routed to Amazon CloudWatch or AWS IoT Sitewise via the AWS IoT message routing function. Monitoring the system telemetry messages allows you to view and graph the number of data tags and Type, Value, Quality (TQV) data points being ingested per second into AWS IoT Sitewise to manage service quotas as well as the system memory and CPU usage.

The diagram below shows an example of the telemetry data being displayed in AWS Managed Grafana. Here we are monitoring the OSI Pi ingress data points and AWS IoT Sitewise data point (TQV) publish rates as well as the memory and CPU usage of the OSI Pi streaming data connector in the Grafana console:



Full details of the asynchronous messages are provided in Appendix B: Asynchronous PubSub Message Updates.

OSI Pi Streaming Data Connector Installation

The OSI Pi Streaming Data Connector is deployed as an AWS IoT Greengrass custom component using the AWS IoT Greengrass Development Kit (GDK). The following provides an installation guide for the connector.

OSI PI Server Requirements

Below is a list of requirements and common considerations when deploying the OSI Pi Streaming Data Connector to access an OSI PI server.

1) **OSI Pi WebAPI:** The OSI Pi Streaming Data Connector integrates to the OSI PI server via the REST'ful Pi WebAPI and so this connector must be installed. (Min version TBA)

2) **Basic Authentication:** The OSI Pi Streaming Data Connector in its current form only supports Basic Authentication for the REST'ful API access and so the OSI Pi server must be configured to allow this authentication method.

3) **User Credentials:** The OSI Pi Streaming Data Connector will need suitable user credentials on the OSI pi server. These are stored securely in AWS Secrets Manager.

4) **Private Certificate:** If the OSI Pi server is configured with a private SSL certificate (as is often the case) the OSI Pi Streaming Data Connector will need the private certificate to be trusted to allow SSL verification of the API connection. See following section: *OSI Pi WebAPI SSL Certificate Verification* for more detail.

5) **Network Access:** The host of the OSI Pi Streaming Data Connector requires network access on port 443 for REST API control plane functions and WebSocket connections to the OSI Pi server.

OSI Pi WebAPI SSL Certificate Verification

The AWS OSI Pi SDK communicates with the OSI Pi server via the Pi WebAPI using a REST'ful HTTPs interface over TLS. This connection needs a publicly trusted SSL Certificate to validate the identity of the OSI Pi server to the AWS OSI Pi SDK client.

There are three supported scenarios for verifying the OSI Pi WebAPI SSL Certificate:

1) **Highly Recommended / Production Use:** Use SSL certificates with a publicly trusted CA chain to identify the OSI Pi Server. In this highly recommended mode, no further configuration is required, the AWS OSI Pi SDK is configured to verify the SSL certificate by default.

2) **Only Recommended for Development:** It's the default option; and so somewhat common, especially in development environments for OSI Pi server installs to use a self-signed SSL certificate which can't be verified by the AWS OSI Pi SDK without a public trust chain. In this case, the recommended option is to install the OSI Pi self-signed certificates into the trusted certificates list of the AWS OSI Pi SDK client host. The procedure for this will be specific to the host operating system.

3) Not Recommended: While not a recommended configuration, you can prevent the AWS OSI Pi SDK from verifying the Servers SSL certificate by setting '**verifySSL=0'** in the IoT Shadow document configuration. Only use this in highly trusted environments to verify initial connectivity and operations.

AWS IoT Greengrass Host Minimum Requirements

The OSI Pi Streaming Data Connector is deployed to the edge on an AWS IoT Greengrass managed host.

Minimum Requirements for the AWS IoT Greengrass managed hosts are:

a) Supports minimum requirements for AWS IoT Greengrass Core software,
b) Recent release of CentOS / RHEL / Amazon Linux2. Debian / Ubuntu is supported but were found to limit WebSocket connections to 5000 by default,
c) 1 vCPU / 1GiB for each OSI Pi connector deployed to the host,
d) 10G storage,

Software Dependencies:

a) Node v16+,
b) NPM v8+,
c) JDK 11+ (Recommended AWS Java 17 Corretto),

AWS IoT Greengrass Edge Host Build Example:

In this example, we will build an Amazon EC2 instance with Amazon2 Linux to host AWS IoT Greengrass and install all the dependencies needed for the OSI Pi Streaming Data Connector. This instance will need network access to the OSI Pi server to request data.

1. In your AWS EC2 console, initiate a new instance and select the latest Amazon Linux2 x86-64 architecture.
2. In this example we will select a t3.medium instance. This gives 2vCPU and 4GiB Memory, enough for the operating system and 1 x OSI Pi data connector.
3. Add an SSH key and create a security group that allows SSH for your IP address (or as needed for SSH access in your specific environment),
4. Add 10G of GP2 storage and click Launch Instance.
5. Once the instance is initialized, follow the below steps:

```
# SSH to the instance
ssh -i ~/.ssh/[Your SSH Key] ec2-user@[instance-url / IP]

# Run update
sudo yum update

# Install Developer Tools
sudo yum groupinstall "Development Tools" -y

# As per Set up a Linux device for AWS IoT Greengrass V2, config sudoers file:
# Note: Needed for Amazon Linux2 but is default config in some other Linux distros.

sudo vi /etc/sudoers
# If needed, replace: root ALL=(ALL) ALL
# With: root ALL=(ALL:ALL) ALL

# Install Node16 / NPM
sudo yum install https://rpm.nodesource.com/pub_16.x/nodistro/repo/nodesource-
release-nodistro-1.noarch.rpm -y
sudo yum install nodejs -y --setopt=nodesource-nodejs.module_hotfixes=1

# Verify node and npm install
node -v
npm -v

# Install JDK17 Corretto
sudo yum install java-17-amazon-corretto-devel -y

# Verify Java VM / JDK install
java --version
javac --version

# The instance is now ready to go!
```

Below is a similar build procedure for Debian (apt) based systems (Tested on Ubuntu20):

```
# SSH to the instance
ssh -i ~/.ssh/[Your SSH Key] [USERNAME]@[instance-url / IP]

# Update and Install Dependencies
sudo apt update
sudo apt install -y build-essential software-properties-common
sudo apt-get install -y ca-certificates curl gnupg

# As per Set up a Linux device for AWS IoT Greengrass V2, config sudoers file:
# Note: This is already applied as default config for the tested Ubuntu distro.

sudo vi /etc/sudoers
# If needed, replace: root ALL=(ALL) ALL
# With: root ALL=(ALL:ALL) ALL

# Install Node16 / NPM
sudo mkdir -p /etc/apt/keyrings
curl -fsSL https://deb.nodesource.com/gpgkey/nodesource-repo.gpg.key | sudo gpg --
dearmor -o /etc/apt/keyrings/nodesource.gpg

NODE_MAJOR=16

sudo apt-get update
sudo apt-get install nodejs -y

# Verify node and npm install
node -v
npm -v

# Install JDK17 Corretto
wget -O- https://apt.corretto.aws/corretto.key | sudo apt-key add -
sudo add-apt-repository 'deb https://apt.corretto.aws stable main'
sudo apt-get update; sudo apt-get install -y java-17-amazon-corretto-jdk

# Verify Java VM / JDK install
java --version
javac –version

# The instance is now ready to go!
```

Manually Testing OSI PI Access

Once the AWS IoT Greengrass host is deployed, perform a manual check to test the accessibility and authentication of the OSI Pi Server from the AWS IoT Greengrass host by emulating an OSI Pi WebAPI request.

We can do this with a cURL command to request the root path of the OSI PI WebAPI. From the AWS IoT Greengrass host terminal, enter the below:

```
PI_USERNAME=[PI WebAPI Basic Auth Username]
PI_PASSWORD=[PI WebAPI Basic Auth Password]
PI_SERVER_URL=[PI Server URL]

curl -u $PI_USERNAME:$PI_PASSWORD  -X GET https://$PI_SERVER_URL/piwebapi

# If using self-signed certs on OSI Pi server will need to add the -k switch.
# Note: This ignores certificate verification, only use in trusted environments.

curl -k -u $PI_USERNAME:$PI_PASSWORD  -X GET https://$PI_SERVER_URL/piwebapi
```

If network access and OSI Pi WebAPI authentication is successful, you will receive a message similar to below (this has been pretty printed here):

```
{
  "Links": {
    "Self": "https://[OSI_PI_SERVER_URL]/piwebapi/",
    "AssetServers": "https://[OSI_PI_SERVER_URL]/piwebapi/assetservers",
    "DataServers": "https://[OSI_PI_SERVER_URL]/piwebapi/dataservers",
    "Omf": "https://[OSI_PI_SERVER_URL]/piwebapi/omf",
    "Search": "https://[OSI_PI_SERVER_URL]/piwebapi/search",
    "System": "https://[OSI_PI_SERVER_URL]/piwebapi/system"
  }
}
```

If the cURL request fails, stop here and fault-find the network connectivity and OSI Pi WebAPI authentication before proceeding.

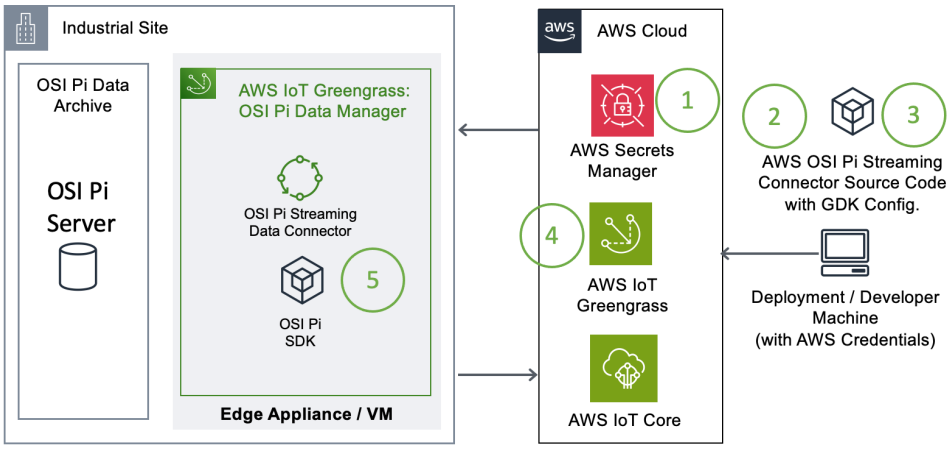
Install AWS IoT Greengrass on Edge Host

1) Ensure the compute host meets the AWS IoT Greengrass host requirements including installing all software dependencies described previously.
2) Install AWS IoT Greengrass on the host by following this AWS IoT Greengrass getting started guide during which:
   a. Enter a unique Core device name: i.e: osi-pi-greengrass-connector01
   b. Enter a unique Thing group name: i.e: osi-pi-greengrass-connector01-group
   c. Follow the instructions in 'Step 3: Install the Greengrass Core software'
3) Return to the AWS IoT Greengrass console and confirm the instance status is Healthy.

Deploy the OSI Pi Streaming Data Connector

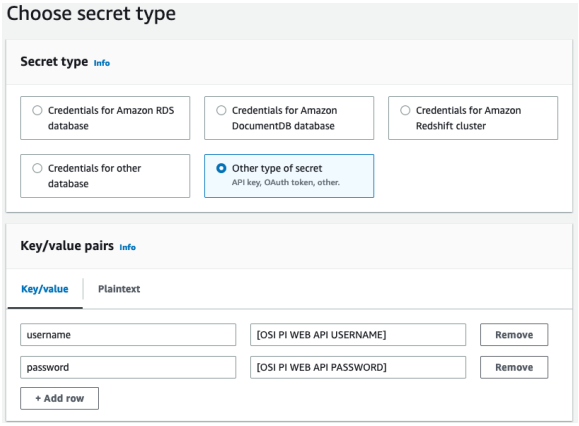
At this point, its assumed that a suitable AWS IoT Greengrass instance is accessible to deploy the OSI Pi Streaming Data Connector. The below procedure uses the AWS IoT Greengrass Development Kit (GDK) to build and publish the OSI Pi Streaming Data Connector to AWS IoT Greengrass component library in your AWS account. This procedure can be run on any Linux / Mac development machine that has suitable AWS credentials applied. It's also common in a development environment to use the Greengrass host itself as the development machine as well, also providing it has suitable AWS credentials available.

The following describes the process to download, build, publish and deploy the OSI Pi streaming data connector to the AWS IoT Greengrass instance.



1. Add OSI Pi WebAPi Basic Credentials in AWS Secrets Manager:

- Create an entry in AWS Secrets Manager (in your desired region) with individual keys for “username”: [value] and “password”: [value] to securely store the OSI Pi WebAPI Basic credentials.
    - In the AWS Console, go to Secrets Manager,
    - Click “Store a new Secret”,
    - Select “Other Type Of secret” and enter key/value pairs for username and password as shown below:



Click Next, give the secret a name such as “osi-pi-credentials” and click through till the Store button.

- Note: Don’t use /path/creds naming format as it causes problems with IAM permissions later on.

- Record the secret ARN for later use.

2. Clone the OSI Pi Streaming Data Connector:

- Clone the OSI Pi Streaming Data Connector repository and cd into **src** directory:

Note: *This is currently an internal AWS repository. Until this is made public the source directory will be distributed via a zip file in AWS Content Portal.*

```
git clone --recursive git@ssh.gitlab.aws.dev:aws-osi-pi-integeration-library/osi-pi-streaming-data-connector.git
```

3. Configure the AWS IoT Greengrass Development Kit Config

- Using your preferred IDE or text editor, update src/gdk-config.json:
    - Set the desired **region**.
    - The bucket name configured will be appended with account and region then created to host the component artifacts. It is by default *aws-greengrass-components*, don't change this name as has other deployment dependencies.
    - Leave remaining fields unchanged.
    - Save and close gdk-config.json

4. Configure Minimum AWS IoT Greengrass IAM Permissions

- Configure the minimum IAM permissions for the OSI Pi Streaming Data Connector. In the AWS IoT Console; select Security >> Role-Alias and the IAM Role Alias that was attached to the IoT Greengrass Device. In the role alias, click on the attached IAM role.

  Edit the role to add the below in-line policy that allows:
    - Access to upload data to AWS IoT Sitewise,
    - To access AWS IoT Greengrass software artifacts in Amazon S3 and
    - Access the OSI Pi credentials in AWS Secrets Manager.

Apply the below in line policy to the Role listed in the AWS IoT Greengrass Role Alias:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "iotsitewise:BatchPutAssetPropertyValue",
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": "s3:GetObject",
      "Resource": "arn:aws:s3:::aws-greengrass-components-[REGION]-[ACCOUNT-
Number]/*"
    },
    {
      "Effect": "Allow",
      "Action": "secretsmanager:GetSecretValue",
      "Resource": "arn:aws:secretsmanager:us-east-1:1234567890:secret:osi-pi-
credentials-Abcd1234"
    }
  ]
}
```

- aws-greengrass-comonents is the bucket name provided in the gdk_config.json configuration below. This bucket will be created with the name appended with [REGION]-[ACCOUNT-NUMBER].

  For Example: *aws-greengrass-components-us-east-1-1234567890*, update the value here to suit in the current environment

- arn:aws:secretsmanager:us-east-1:1234567890:secret:osi-pi-credentials-Abcd1234 is the ARN of the AWS Secrets Manager entry. Update accordingly with the ARN of the previously created entry.

5. Build and Publish the OSI Streaming Data Connector to AWS IoT Greengrass

- If not already done, install or update the AWS IoT Greengrass Development Kit (GDK) on the developer machine (assuming Mac / Linux):

```
python3 -m pip install -U git+https://github.com/aws-greengrass/aws-greengrass-gdk-
cli.git
```

- Build the component using GDK (from the osi-pi-streaming-data-connector/src folder):

```
cd osi-pi-streaming-data-connector/src
gdk component build
```

Expect an output such as:

```
src# gdk component build
INFO - Getting project configuration from gdk-config.json
INFO - Found component recipe file 'recipe.json' in the  project directory.
INFO - Building the component 'com.amazon.osi-pi-streaming-data-connector' with the
given project configuration.
INFO - Using 'zip' build system to build the component.
WARNING - This component is identified as using 'zip' build system. If this is
incorrect, please exit and specify custom build command in the 'gdk-config.json'.
INFO - Zipping source code files of the component.
INFO - Copying over the build artifacts to the greengrass component artifacts build
folder.
INFO - Updating artifact URIs in the recipe.
```

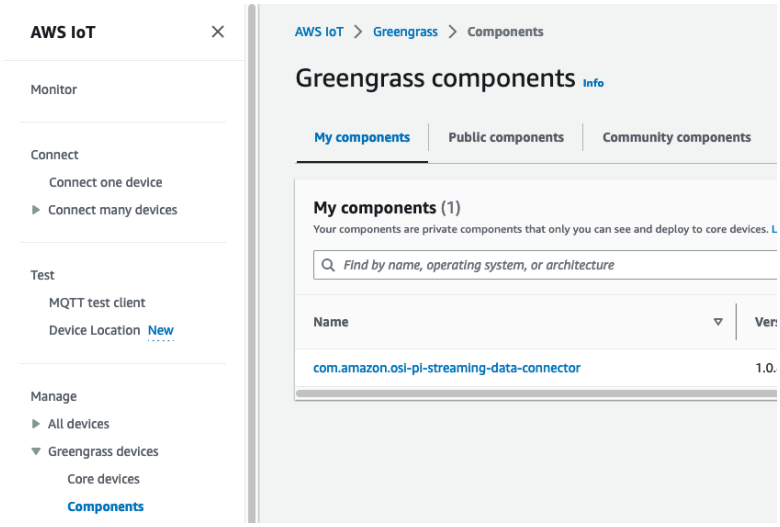
- Publish the component using GDK to AWS IoT Greengrass in your AWS account:

```
gdk component publish
```

Expect an output such as:

```
INFO - Getting project configuration from gdk-config.json
INFO - Found component recipe file 'recipe.json' in the  project directory.
INFO - Found credentials in shared credentials file: ~/.aws/credentials
INFO - Using '1.0.4' as the next version of the component 'com.amazon.osi-pi-
streaming-data-connector' to create.
INFO - Publishing the component 'com.amazon.osi-pi-streaming-data-connector' with
the given project configuration.
INFO - Uploading the component built artifacts to s3 bucket.
INFO - Uploading component artifacts to S3 bucket: aws-greengrass-components-aus-
east-1-1234567890. If this is your first time using this bucket, add the
's3:GetObject' permission to each core device's token exchange role to allow it to
download the component artifacts. For more information, see
https://docs.aws.amazon.com/greengrass/v2/developerguide/device-service-role.html.
INFO - Not creating an artifacts bucket as it already exists.
INFO - Updating the component recipe com.amazon.osi-pi-streaming-data-connector-
1.0.4.
INFO - Creating a new greengrass component com.amazon.osi-pi-streaming-data-
connector-1.0.4
INFO - Created private version '1.0.4' of the component 'com.amazon.osi-pi-
streaming-data-connector' in the account.
```

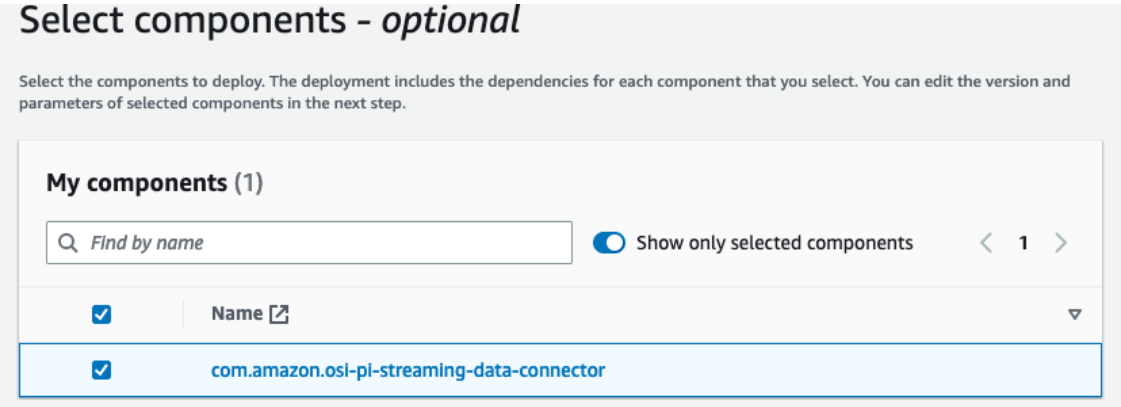
The component will now be available from the AWS IoT Greengrass console as shown below:



6. Deploy the OSI Pi Streaming Data Connector to AWS IoT Greengrass Host

- In the AWS IoT Console, open the MQTT Test client and subscribe to:
  - osi-pi/streaming-data-connector/+/egress
  - This is the egress control PubSub topic from the OSI Pi Streaming data component.

- Following the AWS IoT Greengrass Create Deployment guide, deploy the OSI Pi Streaming Data Connector to the AWS IoT Greengrass host. During deployment:
  - Select the com.amazon.osi-pi-streaming-data-connector component in the deployment as shown below,
  - No additional configuration is required for this component.



- If successfully deployed, the component will automatically initialise and then publish the following command to the components control egress topic which you will see in the console MQTT Test client:
- Topic: **osi-pi/streaming-data-connector/[THING_NAME]/egress**

```
{
  "id": 1,
  "route": "osi-pi-streaming-data-connector-action",
  "status": 200,
  "response": "OSI Pi Streaming Data Connector successfully initialized!"
}
```

- In the first install, the component will look for and if needed, create a default IoT Device shadow configuration and transmit the following message:

```
{
  "id": 2,
  "route": "aws-iot-shadow-controller-action",
  "status": 200,
  "response": {
    "action": "created-default-shadow-config",
    "shadow": "osi-pi-streaming-data-connector-config",
    "shadowUpdate": {
      "state": {
        "desired": {
          "region": "us-east-1",
          "piServerUrl": "",
[OUTPUT REMOPVED FOR BREVITY]
}
```

7. Configure the OSI Pi Data Connector Static (IoT Shadow) Config

- Once the component has initialized, it will look for an AWS IoT device shadow named osi-pi-streaming-data-connector-config. If in the first instance the device shadow doesn't exist, it will create it with the below default shadow document:


```
{
  "state": {
    "desired": {
      "region": "us-east-1",
      "osiPiServerConfig": {
        "piServerUrl": "",
        "piApiRootPath": "piwebapi",
        "maxPiApiRequestPerSec": 25,
        "maxPiApiQueryResponseItems": 1000,
        "authMode": "basic",
        "verifySsl": 1
      },
      "awsSitewisePublisherConfig": {
        "sitewiseMaxTqvPublishRate": 5000,
        "sitewisePropertyAliasBatchSize": 10,
        "sitewisePropertyValueBatchSize": 10,
        "sitewisePropertyPublishMaxAgeSecs": 300
      },
      "osiPiWebSocketManagerConfig": {
        "maxPiDataPointWebSockets": 5000,
        "maxPiPointsPerWebSocket": 100
      },
      "awsSeceretsManagerConfig": {
        "piCredentialsAwsSecretsArn": ""
      },
      "systemTelemetryConfig": {
        "telemetryUpdateSecs": 10
      }
    },
    "reported": {
      "region": "us-east-1",
      "osiPiServerConfig": {
        "piServerUrl": "",
        "piApiRootPath": "piwebapi",
        "maxPiApiRequestPerSec": 25,
        "maxPiApiQueryResponseItems": 1000,
        "authMode": "basic",
        "verifySsl": 1
      },
      "awsSitewisePublisherConfig": {
        "sitewiseMaxTqvPublishRate": 5000,
        "sitewisePropertyAliasBatchSize": 10,
        "sitewisePropertyValueBatchSize": 10,
        "sitewisePropertyPublishMaxAgeSecs": 300
      },
      "osiPiWebSocketManagerConfig": {
        "maxPiDataPointWebSockets": 5000,
        "maxPiPointsPerWebSocket": 100
      },
      "awsSeceretsManagerConfig": {
        "piCredentialsAwsSecretsArn": ""
      },
      "systemTelemetryConfig": {
        "telemetryUpdateSecs": 10
      }
    }
  }
}
```

Where:

- **region:** AWS Region the OSI Pi data connector and architecture is deployed.
  String | Mandatory |Default: us-east-1 | Valid: Any IoT Greengrass / Sitewise supported AWS regions.

- **piServerUrl:** URL of OSI Pi Server. Include port number if not default (443) port.
  E.g: my-pi-server.com:3000
  String | Mandatory | Default: N/A |Valid: Any valid URL

- **piApiRootPath:** The API Path root configured by the OSI Pi WebAPI. Used to form the full PiWebAPI URL path with *piServerUrl***.**
  String | Mandatory | Default: piwebapi | Valid: Any HTTP safe string

- **maxPiApiRequestPerSec:** The components maximum rate of Web (REST) API calls permitted to the OSP PI Server. Use this to manage load on the OSI Pi Servers.
  Integer | Mandatory | Default: 50 | Valid: 1 – 250

- **maxPiApiQueryResponseItems:** The maximum objects or items that will be requested from the OSI Pi server in a single API call. The OSI Pi server has a query size limit that is configurable by the OSI Pi administrator that is by default set to 1000. This value can't overwrite that configuration and must be less than or equal to the value set in the OSI Pi server for correct operation.
  Integer | Mandatory | Default: 1000 | Valid: 100 – 1000

- **maxPiDataPointWebSockets:** The maximum WebSocket's that the connector will be allowed to connect to the OSI PI Server. Use this to manage load on the OSI Pi Server.
  Integer | Mandatory | Default: 5000 | Valid: 1 – 10000

- **maxPiPointsPerWebSocket:** The maximum PiData points that will be allocated to a single WebSocket. See the section OSI PiPoint Data (WebSocket) Channels for more detail. May limit scalability if set to low as Max number of supported WebSocket's is reached.
  Integer | Mandatory | Default: 100 | Valid: 1 – 100

- **sitewiseMaxTqvPublishRate:** Sets the maximum TQVs that will be published to AWS IoT Sitewise per second. Use this to manage buffering for Sitewise Quota limits.
  Integer | Mandatory | Default: 5000 | Valid: 1000 – 100000

- **sitewisePropertyAliasBatchSize:** Minimum number of Pi Data Tags (~Sitewise Property Alias) entries to batch per Sitewise ingest message. Set lower to reduce latency / buffering and connector memory use and higher to reduce Sitewise message count, consumption costs and quota limit requirements.
  Integer | Mandatory | Default: 10 | Valid: 1 – 10

- **sitewisePropertyValueBatchSize:** Minimum number of individual time-series data points per Pi Data Tag (~Sitewise Property Value) to batch per Sitewise ingest message. Set lower to reduce latency / buffering and connector memory/CPU use and higher to reduce Sitewise message count, consumption costs and quota limit requirements.
  Integer | Mandatory | Default: 10 | Valid: 1 – 10

- **sitewisePropertyPublishMaxAgeSecs:** Maximum age in seconds that an OSI Pi data point will be buffered before a publish is forced. This ensures that data points that don't reach the batch parameter criteria for publishing in a reasonable period from becoming stale before being published. Set low to reduce oldest data point latency time and higher to reduce Sitewise message count, consumption costs and quota limit requirements.
  Integer | Mandatory | Default: 300 | Valid: 30 – 3600

- **systemTelemetryUpdateIntervalSec:** Interval (in seconds) to publish component telemetry messages.
  Integer | Mandatory | Default: 10 | Valid: 5 – 60

- **piCredentialsAwsSecretsArn:** AWS Secrets Manager ARN of configured OSI Pi WebAPI credentials.
  String | Mandatory | Default: N/A | Valid: Any valid AWS Secrets ARN.

- **authMode:** OSI Pi REST'ful WebAPI authentication method. Currently only Basic supported.
  String | Mandatory | Default: basic | Valid: basic

- **verifySsl:** Allows API calls to OSI Pi server to ignore SSL certificate verification if using private certificates.
  See: Pi WebAPI SSL Certificate Verification section above for important security considerations.
  Values:
  0 – No Certificate Verification. **(Not recommended)**
  1 - Verify SSL Certificates.
  Integer | Mandatory | Default: 1 | Valid: 0 – 1

  - Update the shadow document in the AWS IoT console:
    - Go to IoT Things and select the AWS IoT Thing associated with the Greengrass device, select Device Shadows and click the shadow named osi-pi-streaming-data-connector-config and click Edit,
    - The Desired State is that which you would like to apply to the connector remotely via AWS IoT. The Reported State is that reported as current from the OSI PI Streaming data connector itself. Only make changes / configure the desired state.
    - Update the region, piServerUrl and piCredentialsAwsSecretsArn parameters to suit your local environment.
    - If using a self-signed certificate on the OSI PI server, set verifySSL = 0.
    - Tune the remaining parameters as / if needed.
    - Click Update on the AWS IoT Shadow Document. This will send the desired state to the OSI Pi Streaming data component.

If successful, the Desired state will be applied to the OSI Pi Streaming Data Connector which will update the IoT shadow document reported state to match and publish the below message to the control egress topic.

```
{
  "id": 3,
  "route": "aws-iot-shadow-controller-action",
  "status": 200,
  "response": {
    "action": "shadow-config-applied",
    "reported-state": {
      "region": "ap-southeast-2",
      "piServerUrl": "ec2-3-25-133-88.ap-southeast-2.compute.amazonaws.com",

[OUTPUT REMOPVED FOR BREVITY]
}
```

Deployment Troubleshooting

If the deployment succeeds but the OSI Pi Data Connectors itself encounters an error, it will (in all controllable cases) be reported to the PubSub interface on the components egress control topic.

Where the component doesn't report back to the PubSub interface, investigate on the AWS IoT Greengrass device logs. On the Greengrass instance, the Greengrass application logs are by default available in the /greengrass/v2/logs directory.

Here there are two logs of Interest:

1. **greengrass.log:** The Greengrass (nucleus) log which provides logs for the Greengrass agent itself and native service and functions,
2. **com.amazon.osi-pi-streaming-data-connector.log**: The application log for the OSI Pi Streaming Data Connector.

If the com.amazon.osi-pi-streaming-data-connector.log is not present its because it failed to deploy. In which case, verify the issue by looking into the greengass.log first.

Using the OSI PI Connector PubSub Interface

In this section, we will use the OSI Pi Streaming Data Connector PubSub messaging interface to discover existing OSI Pi Asset Framework and Data Archive servers, databases, assets elements and then stream OSI PiPoints as an example of how to use the OSI Pi Connector PubSub messaging interface.

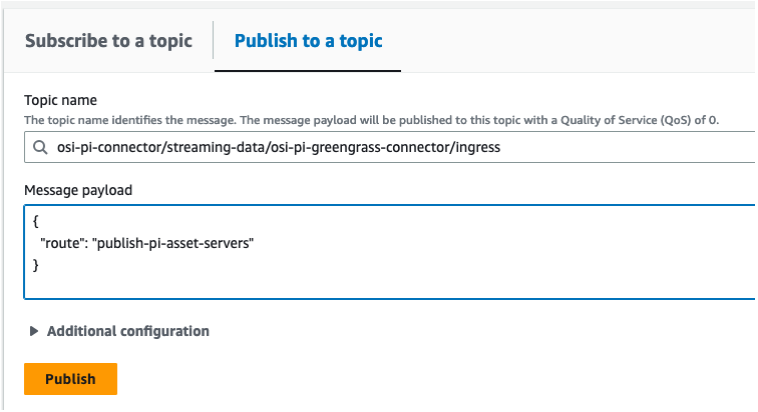
For this example, use the AWS IoT console MQTT test client and:

- Publish requests messages to the control ingress topic:
    - **osi-pi/streaming-data-connector/IOT-THING-NAME]/ingress**

- Subscribe to the control egress topic for response messages:
    - **osi-pi/streaming-data-connector/+/egress**

- Subscribe to other interesting asynchronous message topics:
    - **osi-pi/streaming-data-connector/+/telemetry**
    - **osi-pi-connector/streaming-data/+/websocket-state/#**

AWS IoT MQTT Test Console Publish Example:



AWS IoT MQTT Test Console Response Message Example:

Let do a little discovery of the OSI Pi Asset Framework Server first:

- **List the Asset Framework server/s details:**

Publish the below message to the control ingress topic from the AWS IoT MQTT Test console. The OSI Pi Streaming data connector will query the OSI Pi API and publish the response with the Asset Framework server details to the control egress topic.

```
{
  "route": "publish-pi-asset-servers"
}
```

Note: Save the Asset Server WebId in the response message for the next command.

- **List the databases configured on the selected OSI Pi Asset Framework server:**

In the same way, publish the below command to query the list of OSI PI asset databases configured on the Asset Framework server. Use the WebId of the Asset Framework server queried above for the assetServerWebId parameter.

```
{
  "route": "publish-pi-asset-databases-by-asset-server-webid",
  "params": {
    "assetServerWebId": "[ENTER-ASSET-SERVER-WEB-ID]"
  }
}
```

Note: Save the webid of the database of interest for the next command

- **List all OSI Pi elements in the selected OSI Pi Asset Framework database.**

The below will request all elements saved on the selected OSI Pi Asset Framework database. Here we have used the OSI Pi queryString name:=* so all elements will be returned. If you know of an asset element tree to search you can filter for example “queryString name:=PumpingStation01*“ to only return elements that match the filter. Use the WebId of the asset database queried above.

The queryString is passed directly to the OSI Pi server and so supports any query function as defined in the OSI Pi WebAPI.

```
{
  "route": "publish-pi-asset-elements-by-query",
  "params": {
    "databaseWebId": "[ENTER-ASSET-DATABASE-WEB-ID]",
     "queryString": "name:=*"
  }
}
```

Note: This request may return many thousands of elements and so, the response is broken into 250 elements per MQTT message.

- **List number of elements in a query return:**

If you just want to know the number of asset database elements a query will return without returning the full elements themselves, send the publish-number-asset-elements-by-query route as below. Use this to make sure your filter is as expected.

```
{
  "route": "publish-number-asset-elements-by-query",
  "params": {
    "databaseWebId":  "[ENTER-ASSET-DATABASE-WEB-ID]",
    "queryString": "name:=Pump01*"
   }
}
```

- **List element by parameter:**

Similarly, if you want to view the details of a specific asset element with a known WebId or path you can request its details by publishing the following message:

```
{
  "route": "publish-pi-asset-element-by-param",
  "params": {
        "path": " \\PiAssetDatabase\\Enterprise\\PumpingSite01\\PumpingStation01"
  }
}
```

- **List attributes by parameter:**

Attributes are meta-data wrappers around individual data points (known as PiPonts) of an asset. You can request the details of a specific asset element attribute by publishing the following message:

```
{
  "route": "publish-pi-attribute-by-param",
  "params": {
    "path": "\\\\EC2AMAZ-
5B1B4RL\\PiWebApiScaleTest\\Enterprise\\PumpingSite01\\PumpingStation01\\Pump01|Bea
ringTemp"
  }
}
```

- **List attributes by Asset Element:**

It's also useful to be able to view the full list of attributes associated with a specific asset element. To request all attributes of an asset element, publish the following message:

```
{
  "route": "publish-pi-attribute-by-element-webid",
  "params": {
    "elementWebId": "F1EmA4Fin…f6xXFBVTVAwMQ"
  }
}
```

Now let's take a look at the OSI Pi Data Archive Server:

- **List the Data Archive server/s details:**

This command will respond with a list of the known OSI Pi Data Archive Servers.

```
{
  "route": "publish-pi-data-servers"
}
```

Note: Save the Data Archive server WebId in the response message for the next command.

- **List OSI Pi PiPoint data tags on Data Archive Server:**

Publishing the following command will return a list of al OSI Pi data tags (PiPoints) listed on the data archive server. The queryString is passed directly to the OSI Pi server and so supports any query function as defined in the OSI Pi WebAPI.

```
{
  "route": "publish-pi-points-by-query",
  "params": {
      "dataServerWebId": "[ENTER-DATA-ARCHIVE-SERVER-WEBID]"
      "queryString": "Tag:=*",
      "searchOption":  "contains"
    }
}
```

Note: This request may return many thousands of PiPoints and so, the response is broken into 250 PiPoints per MQTT message.

- **List number of PiPoints in a query return:**

If you just want to know the number of OSI Pi Data Points a query will return without returning the PiPoints themselves, send the publish-number-pi-points-by-query route as below. Use this to make sure your filter is as expected.

```
{
  "route": "publish-number-pi-points-by-query",
  "params": {
      "dataServerWebId": "[ENTER-DATA-ARCHIVE-SERVER-WEBID]"
      "queryString": "Tag:=Pump01*",
      "searchOption":  "contains"
   }
}
```

Requesting PiPoint streaming data ingestion to AWS IoT Sitewise.

- **Queue PiPoint for ingestion to AWS IoT Sitewise**

The OSI Pi Streaming Data Connector accepts requests to stream PiPoints and by default, batches the PiPoints into groups of 100 per WebSocket channel. This is described in more detail in the OSI PiPoint Data (WebSocket) Channels section earlier.

Publish the below message to queue the requests to stream all PiPoints with a Tag that matched “queryString Tag:=PumpingSite01*“. Using the example OSI Pi framework discussed in previous sections, this will match on 1000 individual PiPoint tags. These will be queued and grouped into 10 Channels with 1 WebSocket per channel and 100 PiPoints streaming over each.

```
{
  "route": "queue-pi-points-for-streaming-by-query",
  "params": {
      "dataServerWebId": "[ENTER-DATA-ARCHIVE-SERVER-WEBID]"
      "queryString": "Tag:=PumpingSite01*",
      "searchOption":  "contains"
   }
}
```

Note: Before using this command, it is recommended to test the queryString filter by calling *publish-number-pi-points-by-query* and / or *publish-pi-points-by-query* described above so that you don’t accidently stream more data points than intended.

- **Verifying Queued PiPoint for ingestion to AWS IoT Sitewise**

Before activating the queued PiPoints for streaming, its good practice to validate the state of the PiPoint queue with the following commands.

To View the number of Queued PiPoints:

```
{
  "route": "publish-number-queued-pi-points"
}
```

To view the detail of each queued PiPoint:

```
{
  "route": "publish-queued-pi-points"
}
```

If for any reason the PiPoint streaming data queue is not as expected, you can clear the queue before its activated with the following command:

```
{
  "route": "clear-queued-pi-points-for-streaming"
}
```

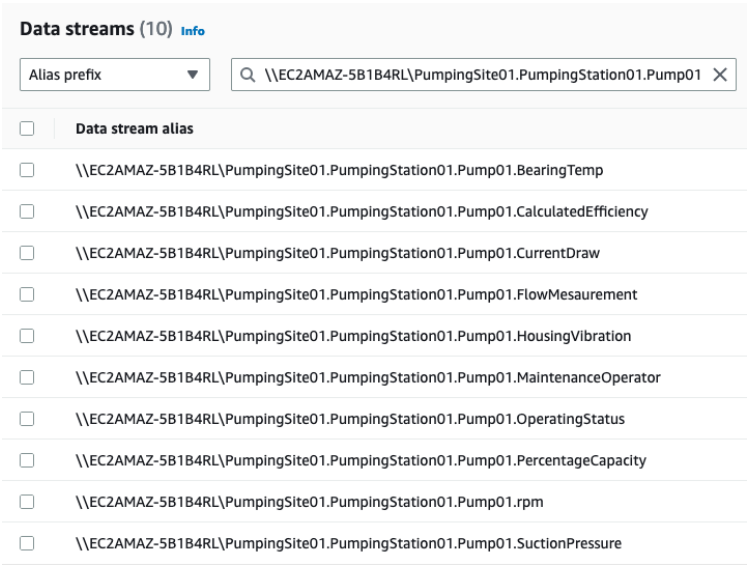
- **Activating Queued PiPoint for ingestion to AWS IoT Sitewise**

Once PiPoints are queued for streaming data then they need to be activated before they are allocated a WebSocket channel and data ingestion begins. Process the queued PiPoints to begin streaming data by publishing the following command:

```
{
  "route": "activate-queued-pi-points-for-streaming"
}
```

Note: As soon as the queued PiPoints are activated, the PiPonts will be processed into WebSocket channels and real time (on-change) PiPoint data will begin streaming to AWS IoT Sitewise. You can monitor the Channel state via the channel state change asynchronous PubSub topics as described in Appendix B of this document.

The PiPoint tag is provided as the Sitewise Data stream alias as shown for the Pump01 asset tags from the AWS IoT Sitewise console below:



Monitoring / Managing / Verifying WebSocket Channels

- **List WebSocket Channels Initiated**

After queued PiPoints are processed, they will be allocated into channels. Publish the below command to see the list of channels and number of PiPoints attached.

```
{
  "route": "publish-channels"
}
```

- **List WebSocket Channel with given PiPPoint**

Publish the below command to return the full details of any channel that is streaming data for any PiPoint path that matches the piPointPathRegex parameter.

```
{  "route": "publish-channels-by-pi-point-path-regex",
  "params": {
     "piPointPathRegex": "PumpingSite01.PumpingStation01.Pump01*"
   }
}
```

Note: There is a significant difference to this path Regex compared to the OSI Pi queryString filtering used in previous examples. QueryString is actioned directly against the OSI Pi WebAPI search query parameter. This is a function of the OSI Pi WebAPI itself and supports a number of keys and search options like wildcards (*) and logical (<, >, =, etc).

The RegEx filter here is applied against the list of PiPoint paths maintained internally by the OSI Pi Streaming Data Connector and uses traditional programmatic RegEx concepts. For example, to see details of all channels, use "piPointPathRegex": "/*"

- **Close Channels**

And finally, if you want to close WebSocket channel/s and disable streaming data from the associated PiPoints, publish one of the following:

To Close all Channels:

```
{
  "route": "close-all-channels"
}
```

To close a specific Channel ID:

```
{
  "route": "close-channel-by-channel-id",
  "params": {
     "channelId": "channel-w5sp01edl1q"
   }
}
```

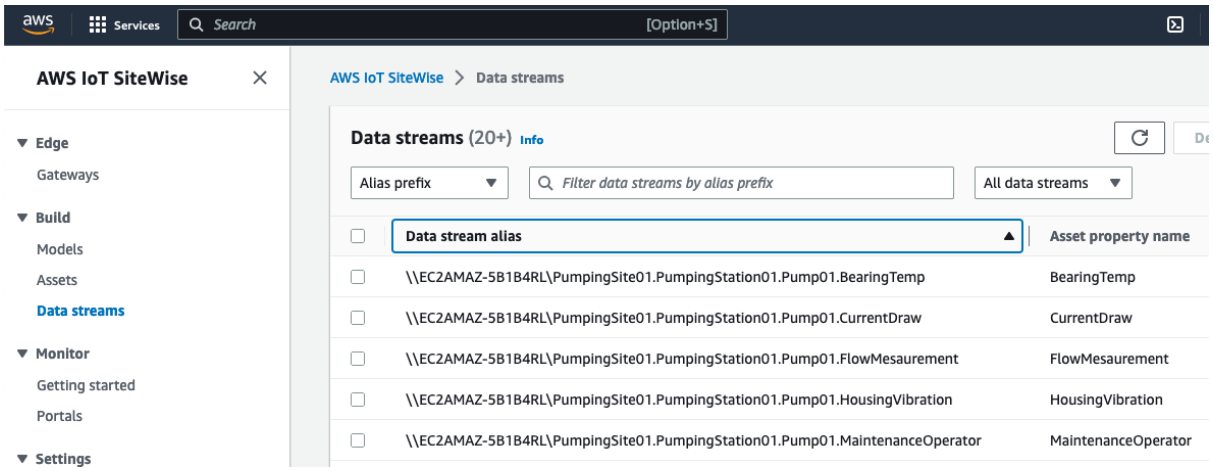
To close any channel containing PiPoint with a path that matches the RegEx parameter.

```
{
  "route": "close-channels-by-pi-point-path-regex",
  "params": {
     "piPointPathRegex": "PumpingSite01.PumpingStation02*"
   }
}
```

This section has provided an introduction to the common functions and basic workflow to discover the OSI Pi server resources and then discover and stream individual PiPoints to AWS IoT Sitewise. The full PubSub command reference is given in Appendix A: PubSub Messaging Interface.

AWS IoT Sitewise Streaming Data

Once the OSI Pi streaming data connector is configured to access selected OSI PiPoints they will be published to AWS IoT Sitewise as dissociated data streams. These can be viewed in the AWS IoT Sitewise console under Data Streams such as shown below.



The data steams Property Alias is set to the OSI PiPoint tag path as a refence to the original data source.

Real Time Visualisation

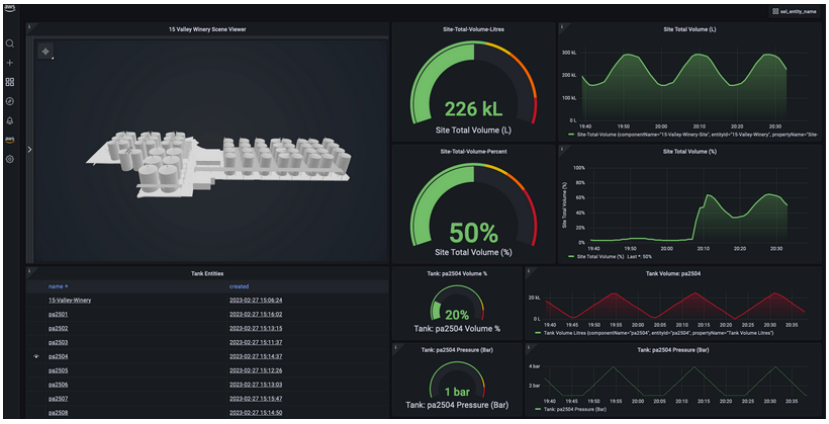
These data points can be visualised by AWS Managed Grafana using the AWS IoT Sitewise Plugin:

Data and Analytics / BI Reporting Integration:

Using the AWS IoT Sitewise cold storage tier and AWS IoT Analytics, these data points can be exported to an AWS data lake and exposed through common BI and reporting tool such as Amazon QuickSight.

Digital Twin Integration:

The AWS IoT TwinMaker service also has a native integration to AWS IoT Sitewise making it possible to visualise this data as part of an AWS managed Digital Twin.



Appendix A: PubSub API

The following provides the full list of PubSub message commands supported by the OSI Pi Streaming Data Connector.

Request Topic: osi-pi/streaming-data-connector/IOT-THING-NAME]/ingress
Response Topic: osi-pi/streaming-data-connector/IOT-THING-NAME]/egress

PubSub Request Format:

```
{
  "route": "[REQUEST-COMAND-TO-TRIGGER]",
  "params": {
    "param01":"optional request param01",
    "param02":"optional request param02"
   }
}
```

PubSub Response Format:

```
{
  "id": 10,
  "route": "[REQUEST-ROUTE]",
  "status": [20x || 40x || 50x],
  "response": {
    "param01": "[OPTIONAL RESONSE MESSAGE / OBJECT OR ERROR DETAILS]",
    "param02": "[OPTIONAL RESONSE MESSAGE / OBJECT OR ERROR DETAILS]"
  }
}
```

Query OSI Pi WebAPI Root Path

publish-pi-root-path:

Request to publish a OSI Pi WebPI root path server links.

MQTT Request:

```
{
  "route": "publish-pi-root-path"
}
```

MQTT Response


```
{
  "id": 12,
  "route": "publish-pi-root-path",
  "status": 200,
  "response": {
    "piRootPath": {
      "links": {
        "Self": "https://ec2-1-2-3-4. us-east-1.compute.amazonaws.com/piwebapi/",
        "AssetServers": "https://ec2-1-2-3-4-us-east-
1.compute.amazonaws.com/piwebapi/assetservers",
        "DataServers": "https://ec2-1-2-3-4-us-east-
1.compute.amazonaws.com/piwebapi/dataservers",
        "Omf": "https://ec2-1-2-3-4-us-east-1.compute.amazonaws.com/piwebapi/omf",
        "Search": "https://ec2-1-2-3-4-us-east-
1.compute.amazonaws.com/piwebapi/search",
        "System": "https://ec2-1-2-3-4-us-east-
1.compute.amazonaws.com/piwebapi/system"
      }
    }
  }
}
```


Query Pi Asset Framework Server Functions

publish-pi-asset-servers:

Request to publish a list of all known Pi Asset Framework Servers.

MQTT Request:


```
{
  "route": "publish-pi-asset-servers"
}
```


MQTT Response


```
{
  "id": 2,
  "route": "publish-pi-asset-servers",
  "status": 200,
  "response": {
    "numberPiAssetServers": 1,
    "piAssetServers": [
      {
        "webid": "F1RSA4……QjFCNFJM",
        "name": "EC2AMAZ-1234567",
        "path": "\\\\EC2AMAZ-1234567",
        "id": "9d488103-5e9b-4cbf-b9d6-f61de9b47014",
       "isConnected": true,
       "serverVersion": "2.10.9.593",
       "serverTime": "2023-07-20T12:35:39.5312815Z"
      }
    ]
  }
}
```

publish-pi-asset-server-by-param:

Request to publish the OSI Pi Asset Server that matches the param search criteria.
Must provide exactly one search parameter, does not accept multiple values.

MQTT Request:

```
{
  "route": "publish-pi-asset-server-by-param",
  "params": {
    "webid": "F1RSA4...jFCNFJM" ||
    "path": "\\\\EC2AMAZ-1234567" ||
    "name": "EC2AMAZ-1234567"
  }
}
```

MQTT Response

```
{
  "id": 40,
  "route": "publish-pi-asset-server-by-param",
  "status": 200,
  "response": {
    "piAssetServer": {
      "webid": "F1RSA4...jFCNFJM",
      "name": "EC2AMAZ-1234567L",
      "path": "\\\\EC2AMAZ-1234567L",
      "id": "9d488103-5e9b-4cbf-b9d6-f61de9b47014",
      "isConnected": true,
      "serverVersion": "2.10.9.593",
      "serverTime": "2023-06-03T14:32:02.6673645Z"
    }
  }
}
```

Query Pi Asset Database Functions

publish-pi-asset-databases-by-asset-server-webid:

Request to publish a list of all OSI Pi Asset Databases on the given Asset Server.

MQTT Request:

```
{
  "route": "publish-pi-asset-databases-by-asset-server-webid",
  "params": {
    "assetServerWebId": "F1RSA4...jFCNFJM"
  }
}
```

MQTT Response

```
{
  "id": 49,
  "route": "publish-pi-asset-databases-by-asset-server-webid",
  "status": 200,
  "response": {
    "numberPiAssetDatabases": 2,
    "piAssetDatabases": [
      {
        "webid": "F1RDA4F…JR1VSQVRJT04",
        "name": "Configuration",
        "path": "\\\\EC2AMAZ-1234567L\\Configuration"
      },
[OUTPUT REMOPVED FOR BREVITY]
}
```

publish-pi-asset-database-by-param:

Request to publish the Pi Asset Database that matches the param search criteria.
Must provide exactly one search parameter, does not accept multiple values.

MQTT Request:

```
{
  "route": "publish-pi-asset-database-by-param",
  "params": {
    "webid": "F1RDAjF...EVURVNU"||
    "path": "\\\\EC2AMAZ-1234567\\PiAssetDatabase"
  }
}
```

MQTT Response

```
{
  "id": 2013,
  "route": "publish-pi-asset-database-by-param",
  "status": 200,
  "response": {
    "piAssetServer": {
      "webid": "F1RDA4FI…JU0NBTEVURVNU",
      "name": "PiAssetDatabase",
      "path": "\\\\EC2AMAZ-1234567\\PiAssetDatabase"
    }
  }
}
```

Query Pi Asset Framework Element Functions

publish-pi-asset-elements-by-query:

Request to publish a list of all OSI Pi Asset Elements that matches the param queryString.
Values are returned via MQTT with 250 Pi Asset Element objects in each response message.
The queryString is passed directly to the OSI Pi server and so supports any query function as defined in the OSI Pi WebAPI.

Note: Using * wildcard query parameters can return all items available on the Pi system. Use wildcards with caution at your own risk!

MQTT Request:

```
{
  "route": "publish-pi-asset-elements-by-query",
  "params": {
    "databaseWebId": "F1RDAjF...EVURVNU",
     "queryString": "name:=Pump01*"
  }
}
```

MQTT Response:

```
{
  "id": 66,
  "route": "publish-pi-asset-elements-by-query",
  "status": 200,
  "response": {
    "queryString": "name:=Pump01*",
    "numberPiAssetElements": 100,
    "startIndex": 0,
    "piAssetElements": [
      {
        "webid": "F1EmA4F...TjAxXFBVTVAwMQ",
        "name": "Pump01",
        "path": "\\\\EC2AMAZ-
1234567L\\PiAssetDatabase\\Enterprise\\PumpingSite01\\PumpingStation01\\Pump01",
        "templateName": "Pump",
        "categoryNames": [],
        "extendedProperties": {},
        "hasChildren": false
      },
[OUTPUT REMOPVED FOR BREVITY]
}
```

publish-pi-asset-element-by-param:

Request to publish the Pi Asset Element that matches the param search criteria.
Must provide exactly one search parameter, does not accept multiple values.

MQTT Request:

```
{
  "route": "publish-pi-asset-element-by-param",
  "params": {
        "webid": "F1EmA4F...TjAxXFBVTVAwMQ", ||
        "path":
"\\PiAssetDatabase\\Enterprise\\PumpingSite01\\PumpingStation01\\Pump01",
  }
}
```

MQTT Response

```
{
  "id": 97,
  "route": "publish-pi-asset-element-by-param",
  "status": 200,
  "response": {
    "piAssetElement": {
      "webid": "F1EmA4F...TjAxXFBVTVAwMQ",
      "name": "Pump01",
      "path": "\\\\EC2AMAZ-
1234567L\\PiAssetDatabase\\Enterprise\\PumpingSite01\\PumpingStation01\\Pump01",
      "templateName": "Pump",
      "categoryNames": [],
      "extendedProperties": {}
    }
  }
}
```

publish-number-asset-elements-by-query:

Request number of Pi Asset Elements that match the search queryString. Will return the given value via MQTT response message without returning the elements themselves. Use this to validate a query string before further actions. The queryString is passed directly to the OSI Pi server and so supports any query function as defined in the OSI Pi WebAPI.

Note: On the OSI Pi server side, all points are still queried but this is still more efficient as doesn't also publish every value in MQTT.

MQTT Request:

```
{
  "route": "publish-number-asset-elements-by-query",
  "params": {
    "databaseWebId": "F1RDAjF...EVURVNU",
    "queryString": "name:=Pump01*"
   }
}
```

MQTT Response:

```
{
  "id": 100,
  "route": "publish-number-asset-elements-by-query",
  "status": 200,
  "response": {
    "queryString": "name:=Pump01*",
    "returnedAssetElements": 100
  }
}
```

publish-pi-asset-elements-by-asset-database-webid:

Request to publish a list of Pi Asset Elements in the given Pi Asset Database. If searchFullHierarchy is true, it will return all child Elements of the database recursively iterating through the full element tree. If false, will only return direct children of the given Asset Database.

searchFullHierarchy defaults to false if not provided.

MQTT Request:

```
{
  "route": "publish-pi-asset-elements-by-asset-database-webid",
  "params": {
        "assetDatabaseWebId": "F1RDAjF...EVURVNU",
        "searchFullHierarchy" : false
  }
}
```

MQTT Response

```
{
  "id": 6,
  "route": "publish-pi-asset-elements-by-asset-database-webid",
  "status": 200,
  "response": {
    "numberPiAssetElements": 1,
    "piAssetElements": [
      {
        "webid": "F1EmA4Fin…EVOVEVSUFJJU0U",
        "name": "Enterprise",
        "path": "\\\\EC2AMAZ-1234567L\\PiAssetDatabase\\Enterprise",
        "templateName": "Enterprise",
        "categoryNames": [],
        "extendedProperties": {},
        "hasChildren": true
      }
    ]
  }
}
}
```

publish-pi-asset-element-children-by-webid:

Request to publish a list of Pi Asset Elements on that are children of the given Parent Element. If searchFullHierarchy is true, it will return all child Elements recursively iterating through the full element tree. If false, will only return direct children of the given Parent Element.

searchFullHierarchy defaults to false if not provided.

MQTT Request:

```
{
  "route": "publish-pi-asset-element-children-by-webid",
  "params": {
    "piElementWebId": "F1EmA4FInZ…EVSUFJJU0U",
     "searchFullHierarchy" : false
  }
}
```

MQTT Response:

```
{
  "id": 10,
  "route": "publish-pi-asset-element-children-by-webid",
  "status": 200,
  "response": {
    "piElementWebId" : "F1EmA4FInZ…EVSUFJJU0U",
    "numberPiAssetElements": 10,
    "piAssetElements": [
      {
        "webid": "F1EmA4FInZt…lOR1NJVEUwMQ",
        "name": "PumpingSite01",
        "path": "\\\\EC2AMAZ-1234567L\\PiAssetDatabase\\Enterprise\\PumpingSite01",
        "templateName": "PumpingSite",
        "categoryNames": [],
        "extendedProperties": {},
        "hasChildren": true
      },
[OUTPUT REMOPVED FOR BREVITY]
}
```

Query Pi Attribute Functions

publish-pi-attribute-by-param:

Request to publish the Pi Attribute that matches the param search criteria.
Must provide exactly one search parameter, does not accept multiple values.

MQTT Request:

```
{
  "route": "publish-pi-attribute-by-param",
  "params": {
    "webid":"F1AbEA4FI…VTVAwMXxCRUFSSU5HVEVNUA" ||
    "path": "\\\\EC2AMAZ-
123456\\PiWebApiScaleTest\\Enterprise\\PumpingSite01\\PumpingStation01\\Pump01|Bear
ingTemp"
  }
}
```

MQTT Response:

```
{
  "id": 41,
  "route": "publish-pi-attribute-by-param",
  "status": 200,
  "response": {
    "piAttribute": {
      "webid": " F1AbEA4FI…VTVAwMXxCRUFSSU5HVEVNUA ",
      "name": "BearingTemp",
      "path": "\\\\EC2AMAZ-
5B1B4RL\\PiWebApiScaleTest\\Enterprise\\PumpingSite01\\PumpingStation01\\Pump01|Bea
ringTemp",
      "type": "Single",
      "defaultUnitsName": "degree Celsius",
      "defaultUnitsNameAbbreviation": "°C",
      "hasChildren": false
    }
  }
}
```

publish-pi-attribute-by-element-webid:

Request to publish the Pi Attribute that matches the param search criteria.

MQTT Request:

```
{
  "route": "publish-pi-attribute-by-element-webid",
  "params": {
    "elementWebId": "F1EmA4FInZ…EVSUFJJU0U"
  }
}
```

MQTT Response:

```
{
  "id": 4,
  "route": "publish-pi-attribute-by-element-webid",
  "status": 200,
  "response": {
    "numberPiApiAttributes": 10,
    "startIndex": 0,
    "piAttributes": [
      {
        "webid": "F1AbEA4FInZ-…SSU5HVEVNUA",
        "name": "BearingTemp",
        "path": "\\\\EC2AMAZ-
123456\\PiWebApiScaleTest\\Enterprise\\PumpingSite01\\PumpingStation01\\Pump01|Bear
ingTemp",
        "type": "Single",
        "defaultUnitsName": "degree Celsius",
        "defaultUnitsNameAbbreviation": "°C",
        "hasChildren": false
      },
[OUTPUT REMOPVED FOR BREVITY]
}
```

Query Pi Asset Element Template Functions

publish-pi-element-template-by-param:

Request to publish the Pi Asset Element Template that matches the param search criteria. Must provide exactly one search parameter, does not accept multiple values.

MQTT Request:

```
{
  "route": " publish-pi-element-template-by-param",
  "params": {
    "webid": "F1ETA4FInZtev0y5...VNbRU5URVJQUklTRV0" ||
    "path": "\\\\EC2AMAZ-1234567\\PiAssetDatabase\\ElementTemplates[Enterprise]"
  }
}
```

MQTT Response:

```
{
  "id": 15,
  "route": "publish-pi-element-template-by-param",
  "status": 200,
  "response": {
    "piAssetElementTemplate": {
      "webid": "F1ETA4FInZtev0y5...VNbRU5URVJQUklTRV0",
      "name": "Enterprise",
      "path": "\\\\EC2AMAZ-
1234567L\\PiAssetDatabase\\ElementTemplates[Enterprise]",
      "baseTemplate": "",
      "namingPattern": "",
      "categoryNames": [],
      "instanceType": "Element",
      "extendedProperties": {}
    }
  }
}
```

publish-pi-element-templates-by-asset-database-webid:

Request to publish the Pi Asset Element Template that matches the param search criteria. Must provide exactly one search parameter, does not accept multiple values.

MQTT Request:

```
{
  "route": "publish-pi-element-templates-by-asset-database-webid",
  "params" : {
    "assetDatabaseWebId": "F1RDAjF...EVURVNU"
  }
}
```

MQTT Response:

```
{
  "id": 203,
  "route": "publish-pi-element-templates-by-asset-database-webid",
  "status": 200,
  "response": {
    "numberPiAssetElementTemplates": 4,
    "piAssetElementTemplates": [
      {
        "webid": "F1ETA4FInZtev…RU5URVJQUklTRV0",
        "name": "Enterprise",
        "path": "\\\\EC2AMAZ-
1234567\\PiWebApiScaleTest\\ElementTemplates[Enterprise]",
        "baseTemplate": "",
        "namingPattern": "",
        "categoryNames": [],
        "instanceType": "Element",
        "extendedProperties": {}
      },
[OUTPUT REMOPVED FOR BREVITY]
}
```

Query Pi Template Attribute Functions

publish-pi-template-attribute-by-param:

Request to publish the Pi Element Template Attribute that matches the param search criteria. Must provide exactly one search parameter, does not accept multiple values.

MQTT Request:

```
{
  "route": "publish-pi-template-attribute-by-param",
  "params": {
    "webid" : "F1ATEA4FInZtev0y……18UlBN" ||
    "path": "\\\\EC2AMAZ-5B1B4RL\\PiWebApiScaleTest\\ElementTemplates[Pump]|rpm"
  }
}
```

MQTT Response:

```
{
  "id": 1059,
  "route": "publish-pi-template-attribute-by-param",
  "status": 200,
  "response": {
    "piTemplateAttribute": {
      "webid" : "F1ATEA4FInZtev0y……18UlBN”,
      "name": "rpm",
      "path": "\\\\EC2AMAZ-5B1B4RL\\PiWebApiScaleTest\\ElementTemplates[Pump]|rpm",
      "type": "Single",
      "defaultUnitsName": "revolution per minute",
      "defaultUnitsNameAbbreviation": "rpm",
      "hasChildren": false
    }
  }
}
```

publish-pi-template-attributes-by-template-webid:

Request to publish all Pi Element Template Attributes that belong to Element Template that matches the given templateWebid.

MQTT Request:

```
{
  "route": "publish-pi-template-attributes-by-template-webid",
  "params": {
 "templateWebid": "F1ETA4FInZ…1QTEFURVNbUFVNUF0"
  }
}
```

MQTT Response:

```
{
  "id": 935,
  "route": "publish-pi-template-attributes-by-template-webid",
  "status": 206,
  "response": {
    "numberPiTemplateAttributes": 10,
    "startIndex": 0,
    "piTemplateAttributes": [
      {
        "webid": "F1ATEA4FI…rxuVElPTlBSRVNTVVJF",
        "name": "SuctionPressure",
        "path": "\\\\EC2AMAZ-
5B1B4RL\\PiWebApiScaleTest\\ElementTemplates[Pump]|SuctionPressure",
        "type": "Double",
        "defaultUnitsName": "pound-force per square inch",
        "defaultUnitsNameAbbreviation": "psi",
        "hasChildren": false
      },
[OUTPUT REMOPVED FOR BREVITY]
}
```

Query Pi Data Archive Server Functions

publish-pi-data-servers:

Request to publish a list of all known Pi Data Servers.

MQTT Request:

```
{
  "route": "publish-pi-data-servers"
}
```

MQTT Response:

```
{
  "id": 12,
  "route": "publish-pi-data-servers",
  "status": 200,
  "response": {
    "numberPiDataServers": 1,
    "piDataServers": [
      {
        "webid": "F1DStNdEET...1QjFCNFJM",
        "name": "EC2AMAZ-1234567",
        "path": "\\\\PIServers[EC2AMAZ-1234567]",
        "id": "1144d7b4-8d38-4d9b-8ce0-5605828acab5",
        "isConnected": true,
        "serverVersion": "3.4.445.688",
        "serverTime": "2023-07-20T12:44:20.9524614Z"
      }
    ]
  }
}
```

publish-pi-data-server-by-param:

Request to publish the Pi Data Server that matches the param search criteria.
Must provide exactly one search parameter, does not accept multiple values.

MQTT Request:

```
{
  "route": "publish-pi-data-server-by-param",
  "params": {
    "webid": "F1DStNdEET...1QjFCNFJM" ||
    "path": "\\\\EC2AMAZ-1234567" ||
    "name": "EC2AMAZ-1234567"
  }
}
```

MQTT Response:

```
{
  "id": 8,
  "route": "publish-pi-data-server-by-param",
  "status": 200,
  "response": {
    "piDataServer": {
      "webid": "F1DStNdEET...1QjFCNFJM",
      "name": "EC2AMAZ-1234567L",
      "path": "\\\\PIServers[EC2AMAZ-1234567L]",
      "id": "1144d7b4-8d38-4d9b-8ce0-5605828acab5",
      "isConnected": true,
      "serverVersion": "3.4.445.688",
      "serverTime": "2023-06-03T15:46:13.2039947Z"
    }
  }
}
```

Query PiPoint Functions

publish-pi-points-by-query:

Requests meta data of all OSI PiPoints that match the OSI Pi Endpoint search format queryString. The queryString is passed directly to the OSI Pi server and so supports any query function as defined in the OSI Pi WebAPI.

Values are returned via MQTT with 250 PiPoints in each response message.
Note: Using * wildcard query parameters can return all items available on the Pi system. Use with caution at your own risk!

searchOption (optional): Search mode to apply. Default: contains

MQTT Request:

```
{
  "route": "publish-pi-points-by-query",
  "params": {
      "dataServerWebId": "F1DSAB2jBM...QQjFCNFJM",
      "queryString": "Tag:=PumpingSite01*",
      "searchOption":  "contains" || "exact-match" || "starts-with" || "ends-with"
   }
}
```

MQTT Response:

```
{
  "id": 293,
  "route": "publish-pi-points-by-query",
  "status": 200,
  "response": {
    "queryString": "Tag:=PumpingSite*",
    "searchOption": "contains",
    "numberPiPoints": 250,
    "startIndex": 9750,
    "piPointItems": [
      {
        "webid": "F1DPtNdE…VBUklOR1RFTVA",
        "name": "PumpingSite10.PumpingStation08.Pump06.BearingTemp",
        "path": "\\\\EC2AMAZ-
1234567L\\PumpingSite10.PumpingStation08.Pump06.BearingTemp",
        "pointClass": "base",
        "pointType": "Float32",
        "engUnits": "",
        "zero": 0,
        "span": 100,
        "future": false
      },
[OUTPUT REMOPVED FOR BREVITY]
}
```

publish-number-pi-points-by-query:

Request number of OSI PiPoints that match the search queryString. Will return the given value via MQTT response message. The queryString is passed directly to the OSI Pi server and so supports any query function as defined in the OSI Pi WebAPI.

Note: On the OSI Pi server side, all points are still queried but is still more efficient than get-pi-points-by-server-query as it doesn't also publish every value in MQTT.

searchOption (optional): Search mode to apply. Default: contains

MQTT Request:

```
{
  "route": "publish-number-pi-points-by-query",
  "params": {
      "dataServerWebId": "F1DSAB2jBM...QQjFCNFJM",
      "queryString": "Tag:=PumpingSite01*",
      "searchOption":  "contains" || "exact-match" || "starts-with" || "ends-with"
   }
}
```

MQTT Response:

```
{
  "id": 3,
  "route": "publish-number-pi-points-by-query",
  "status": 200,
  "response": {
    "queryString": "Tag:=PumpingSite01*",
    "searchOption": "contains",
    "piPointsReturned": 1000
  }
}
```

Stream PiPoint Data Functions

queue-pi-points-for-streaming-by-query:

Request to queue PiPoints for live streaming by associating with a WebSocket channel. This command queues the PiPoints for streaming, you must call the activate-queued-pi-points-for-streaming to begin data transfer. The queryString is passed directly to the OSI Pi server and so supports any query function as defined in the OSI Pi WebAPI.

Note: * wildcard query parameters will try and open a WebSocket channel to all available PiPoints on the Pi Data server. Use wildcards with caution at your own risk!

searchOption (optional): Search mode to apply. Default: contains

MQTT Request:

```
{
  "route": "queue-pi-points-for-streaming-by-query",
  "params": {
      "dataServerWebId": "F1DSAB2jBM...QQjFCNFJM",
      "queryString": "Tag:=PumpingSite01*",
      "searchOption":  "contains" || "exact-match" || "starts-with" || "ends-with"
   }
}
```

MQTT Response:

```
{
  "id": 657,
  "route": "queue-pi-points-for-streaming-by-query",
  "status": 200,
  "response": {
    "queryString": "Tag:=PumpingSite01*",
    "searchOption": "contains",
    "piPointsQueued": 1000,
    "totalPiPointsQueued": 2000
   }
}
```

publish-queued-pi-points:

Request to publish the list of PiPoints queued for live streaming but not yet activated.

MQTT Request:

```
{
  "route": "publish-queued-pi-points"
}
```

MQTT Response:


```
{
  "id": 419,
  "route": "publish-queued-pi-points",
  "status": 200,
  "response": {
    "startIndex": 9750,
    "numberQueuedPiPoints": 250,
    "queuedPiPoints": [
      {
        "webid":
"F1DPtNdEETiNm02M4FYFgorKtQJWEAAARUMyQU1BWi01QjFCNFJMXFBVTVBJTkdTSVRFMTAuUFVNUElOR1
NUQVRJT04wOC5QVU1QMDYuQkVBUklOR1RFTVA",
        "name": "PumpingSite10.PumpingStation08.Pump06.BearingTemp",
        "path": "\\\\EC2AMAZ-
5B1B4RL\\PumpingSite10.PumpingStation08.Pump06.BearingTemp"
      },
[OUTPUT REMOPVED FOR BREVITY]
}
```


publish-number-queued-pi-points:

Request to publish the number of PiPoints queued for live streaming but not yet activated.

MQTT Request:

```
{
  "route": "publish-number-queued-pi-points"
}
```

MQTT Response:

```
{
  "id": 424,
  "route": "publish-number-queued-pi-points",
  "status": 200,
  "response": {
    "numberQueuedPiPoints": 10000
  }
}
```

activate-queued-pi-points-for-streaming:

Request to process PiPoints queued for live streaming by associating with a channel (over WebSocket) to OSI Pi server that will accept on-change PiPoint data and ingest it to AWS IoT. Once calls, all active channels will begin streaming data to AWS IoT Sitewise

MQTT Request:

```
{
  "route": "activate-queued-pi-points-for-streaming"
}
```

MQTT Response:

```
{
  "id": 57,
  "route": "activate-queued-pi-points-for-streaming",
  "status": 200,
  "response": {
    "processedChannels": 30,
    "processedPiPoints": 3000
  }
}
```

Query WebSocket Channel Functions

publish-channels:

Publishes detail of all configured WebSocket channels including connection status and number of associated PiPoints streaming over this channel.

MQTT Request:

```
{
  "route": "publish-channels"
}
```

MQTT Response:

```
{
  "id": 179,
  "route": "publish-channels",
  "status": 200,
  "response": {
    "numChannels": 110,
    "totalNumPiPoints": 11000,
    "channels": [
      {
        "channelId": "channel-abcdef123456",
        "websocket-status": "OPEN",
        "numPiPoints": 100
      },
[OUTPUT REMOVED FOMR BREVITY]
    ]
  }
}
```

publish-channel-stats:

Publishes the aggregate number of channel and configured PiPoints per connection state of all configured (WebSocket) channels.

MQTT Request:

```
{
  "route":"publish-channel-stats"
}
```

MQTT Response:

```
{
  "id": 255,
  "route": "publish-channel-stats",
  "status": 200,
  "response": {
    "channelStatesCount": {
      "connecting": 0,
      "open": 30,
      "closing": 0,
      "closed": 20
    },
    "channelPiPointsCount": {
      "connecting": 0,
      "open": 3000,
      "closing": 0,
      "closed": 2000
    },
    "totalChannels": 50,
    "totalPiPoints": 5000
  }
}
```

publish-channels-by-state:

Publishes detail of all configured (WebSocket) channels with the matching WebSocket connection state.

MQTT Request:

```
{
  "route": "publish-channels-by-state",
    "params" : {
     "websocketState" : "connecting" || "open" || "closing" || "closed"
   }
}
```

MQTT Response:

```
{
  "id": 173,
  "route": "publish-channels-by-state",
  "status": 206,
  "response": {
    "startIndex": 0,
    "numberChannels": 80,
    "channels": [
      {
        "channelId": "channel-r9umvhe57v",
        "websocket-status": "open",
        "numPiPoints": 100
      },
      {
        "channelId": "channel-krmk1afqh5s",
        "websocket-status": "open",
        "numPiPoints": 100
      },
  [OUTPUT REMOVED FOMR BREVITY]
}
```

publish-channels-by-pi-point-path-regex:

Publishes detail of any channel Id (WebSocket) that includes a PiPoint path that matches the RegEx parameter.

You can for example request the detail of all configured channels by using "piPointPathRegex": "/*"

MQTT Request:

```
{
"route": "publish-channels-by-pi-point-path-regex",
  "params" : {
     "piPointPathRegex" : "PumpingSite01.PumpingStation01.Pump01*"
   }
}
```

MQTT Response:

```
{
  "id": 25159,
  "route": "publish-channels-by-pi-point-path-regex",
  "status": 200,
  "response": {
    "piPointPathRegex": "PumpingSite01.PumpingStation01.Pump01*",
    "numChannels": 2,
    "totalNumPiPoints": 150,
    "channels": [
      {
        "channelId": "channel-abcdef123456",
        "websocket-status": "OPEN",
        "numPiPoints": 75
      },
      [OUTPUT REMOVED FOMR BREVITY]
}
```

publish-channel-by-channel-id:

Publishes detail of the provided channel Id including connection status and full details of associated PiPoints streaming over this WebSocket channel.

MQTT Request:

```
{
  "route": "publish-channel-by-channel-id",
  "params": {
     "channelId": "channel-abcdef123456"
   }
}
```

MQTT Response:

```
{
  "id": 1772,
  "route": "publish-channel-by-channel-id",
  "status": 200,
  "response": {
    "channel": {
      "channelId": "channel-abcdef123456",
      "websocketState": "OPEN",
      "numberPiPoints": 75
      "piPoints": [
        {
          "webid": "F1DPtNdEETiN…R1RFTVA",
          "name": "PumpingSite01.PumpingStation01.Pump01.BearingTemp",
          "path": "\\\\EC2AMAZ-
1234567L\\PumpingSite01.PumpingStation01.Pump01.BearingTemp"
        },
[OUTPUT REMOVED FOMR BREVITY]
    ]
  }
}
```

publish-channel-by-pi-point-webid:

Publishes full detail of all channels that are streaming the PiPoint with the given WebID. Includes connection status and full details of associated PiPoints streaming over this channel.

MQTT Request:

```
{
  "route": "publish-channel-by-pi-point-webid",
  "params": {
     "piPointWebId": " F1DPtNdEETiNm0...1VSUkVOVERSQVc"
   }
}
```

MQTT Response:

```
{
  "id": 17386,
  "route": "publish-channel-by-pi-point-webid",
  "status": 200,
  "response": {
    "webid": "[SEARCH-WEBID]",
    "channel": {
      "channelId": "channel-abcdef123456",
      "websocketState": "OPEN",
      "numberPiPoints": 75
      "piPoints": [
        {
          "webid": "F1DPtNdEETiN…OR1RFTVA",
          "name": "PumpingSite01.PumpingStation02.Pump06.BearingTemp",
          "path": "\\\\EC2AMAZ-
1234567L\\PumpingSite01.PumpingStation02.Pump06.BearingTemp"
        },
        {
          "webid": "F1DPtNETiNm…RklDSUVOQ1k",
          "name": "PumpingSite01.PumpingStation02.Pump06.CalculatedEfficiency",
          "path": "\\\\EC2AMAZ-
1234567L\\PumpingSite01.PumpingStation02.Pump06.CalculatedEfficiency"
        },
[OUTPUT REMOVED FOMR BREVITY]
}
```

Close WebSocket Channel Functions

Closing a channel closes the associated WebSocket and stops the attached PiPoints streaming data while retaining the channel details such as the channelId and associated PiPoints on the system. Use the close channel functions when you may want to reopen this channel again (using the open channel functions.). Use the delete channel functions to close and remove all reference to the channel from the system state.

close-all-channels:

Close all WebSocket channels and disable streaming data on all associated PiPoints.

MQTT Request:

```
{
  "route": "close-all-channels"
}
```

MQTT Response:

```
{
  "id": 960,
  "route": "close-all-channels",
  "status": 200,
  "response": {
    "response": "Closed 100 WebSocket Channels and 10000 PiPoints streaming data
sessions."
  }
}
```

close-channels-by-pi-point-path-regex:

Close all WebSocket channels and disable streaming data on all associated PiPoints for any channel that's streaming data for a PiPoint that matches the RegEx.

MQTT Request:

```
{
  "route": "close-channels-by-pi-point-path-regex",
  "params": {
     "piPointPathRegex": "PumpingSite01.PumpingStation02*"
   }
}
```

MQTT Response:

```
{
  "id": 1303,
  "route": "close-channels-by-pi-point-path-regex",
  "status": 200,
  "response": {
    "response": "Closed and Deleted 9 OSI Pi WebSocket Channel/s - 900 associated
PiPoints removed."
  }
}
```

close-channel-by-channel-id:

Close the WebSocket channel and disables streaming data on all associated PiPoints for the channel with the given channel-id.

MQTT Request:

```
{
  "route": "close-channel-by-channel-id",
  "params": {
     "channelId": "channel-abcdef123456"
   }
}
```

MQTT Response:

```
{
  "id": 1090,
  "route": "close-channel-by-channel-id",
  "status": 200,
  "response": {
    "response": "Closed and Deleted OSI Pi WebSocket Channel channel-kbm7gddemd -
100 associated PiPoints removed."
  }
}
```

close-channel-by-pi-point-webid:

Close the WebSocket and disables streaming data on all associated PiPoints for the channel that is streaming data for the PiPoint with the given WebId.

MQTT Request:

```
{
  "route": "close-channel-by-pi-point-webid",
  "params": {
      "piPointWebId": "F1DPtNdEETiNm0...1VSUkVOVERSQVc"
   }
}
```

MQTT Response:

```
{
  "id": 1108,
  "route": "close-channel-by-pi-point-webid",
  "status": 200,
  "response": {
    "response": "Closed and Deleted OSI Pi WebSocket Channel channel-3cc5flqgsco -
100 associated PiPoints removed."
  }
}
```

Open WebSocket Channel Functions

Opening a channel reinitializes the channel WebSocket and begins streaming the attached PiPoints data again. If preforming an open channel function on a channel that is already open it will drop and reinitialize the WebSocket leading to a momentary loss of data.

Use the open channel functions to reinitialize a channel that was manually closed or was closed due to a network outage or by the OSI Pi server to restart the WebSocket and the associated PiPoint streaming data.

#open-all-closed-channels:

(Re)Open all WebSockets currently in the closed state and enable streaming data on all associated PiPoints for the given channels.

MQTT Request:

```
{
  "route": "open-all-closed-channels"
}
```

MQTT Response:

```
{
  "id": 334,
  "route": "open-all-closed-channels",
  "status": 200,
  "response": {
    "response": "Open OSI Pi WebSockets for 9900 Channels with 9900 total PiPoints
request complete - monitor websocket-state change topics for status."
  }
}
```

#open-channel-by-channel-id:

(Re)Open the WebSocket and enable streaming data on all associated PiPoints for the channel with the given channel-id.

MQTT Request:

```
{
  "route": "open-channel-by-channel-id",
  "params": {
     "channelId": "channel-hgchplf36g"
   }
}
```

MQTT Response:

```
{
  "id": 568,
  "route": "open-channel-by-channel-id",
  "status": 200,
  "response": {
    "response": "Open OSI Pi WebSocket Channel channel-hgchplf36g request complete
- monitor websocket-state change topics for status."
  }
}
```

Delete WebSocket Channel Functions

Deleting a channel closes the associated WebSocket (if open), stops the attached PiPoints streaming data and deletes all channel details and resources from the system.

Use the delete channel functions to close and remove all reference to the channel from the system state. This function is not reversable and PiPoints removed will have to be re-added individually to restart streaming data flow.

#delete-all-channels:

Close and delete all WebSocket channels and disable streaming data on all associated PiPoints.

MQTT Request:

```
{
  "route": "delete-all-channels"
}
```

MQTT Response:

```
{
  "id": 638,
  "route": "delete-all-channels",
  "status": 200,
  "response": {
    "response": "Deleted 100 WebSocket Channels and 10000 PiPoints streaming data
sessions."
  }
}
```

#delete-channel-by-channel-id:

Close and delete the WebSocket and disables streaming data on all associated PiPoints for the channel with the given channel-id.

MQTT Request:

```
{
  "route": "delete-channel-by-channel-id",
  "params": {
     "channelId": "channel-kbm7gddemd"
   }
}
```

MQTT Response:

```
{
  "id": 740,
  "route": "delete-channel-by-channel-id",
  "status": 200,
  "response": {
    "response": "Deleted and Closed OSI Pi WebSocket Channel channel-qoiq0pk7v7 -
100 associated PiPoints removed."
  }
}
```

#delete-channels-by-pi-point-path-regex:

Close and Delete all WebSocket channels and disable streaming data on all associated PiPoints for any channel that's streaming data for a PiPoint with a path that matches the RegEx parameter.

MQTT Request:

```
{
  "route": "delete-channels-by-pi-point-path-regex",
  "params" : {
     "piPointPathRegex" : "PumpingSite01.PumpingStation02"
   }
}
```

MQTT Response:

TBA…..

Delete Queued / Buffered PiPoints

#delete-pi-data-buffer-queue:

Deletes a percentage of the PiPoints received from the Pi server that have been buffered / queued for publish to Sitewise. Use this to manually clear the buffer to reclaim memory and reduce CPU consumption when under heavy load.

```
{
  "route": "delete-pi-data-buffer-queue",
  "params": {
     "deletePercentQueue": 50
   }
}
```

System / Process Functions:

publish-node-v8-memory-stats:

Request to publish the NodeJS (V8 engine) memory stats of the OSI Pi Streaming Data Connector process.

MQTT Request:

```
{
  "route": "publish-node-v8-memory-stats"
}
```

MQTT Response:


```
{
  "id": 1417,
  "route": "publish-node-v8-memory-stats",
  "status": 200,
  "response": {
    "memoryStats": {
      "total_heap_size": 664911872,
      "total_heap_size_executable": 3407872,
      "total_physical_size": 664748744,
      "total_available_size": 3994691080,
      "used_heap_size": 336947208,
      "heap_size_limit": 4345298944,
      "malloced_memory": 532552,
      "peak_malloced_memory": 3920456,
      "does_zap_garbage": 0,
      "number_of_native_contexts": 1,
      "number_of_detached_contexts": 0,
      "total_global_handles_size": 106496,
      "used_global_handles_size": 75680,
      "external_memory": 48609742
    }
  }
}
```

Appendix B: Asynchronous PubSub Message Updates

The asynchronous PubSub topics and messages to monitor the OSI Pi Streaming Data Connector are as follows:

OSI PI WebSocket Channel State Changed

Publish Topic/s:
osi-pi/streaming-data-connector/IOT-THING-NAME]/websocket-state/opened
osi-pi/streaming-data-connector/IOT-THING-NAME]/websocket-state /errored
osi-pi/streaming-data-connector/IOT-THING-NAME]/websocket-state /closed
osi-pi/streaming-data-connector/IOT-THING-NAME]/websocket-state /failed-deleted

Published when a WebSocket to the OSI Pi server changed state. *websocketStatus* options include [opened | errored | closed | failed-deleted].

Opened, Errored and Closed represent a state change for an initialized and managed WebSocket. A close WebSocket can be requested to be re-opened as it remains on the system.

Failed-Deleted is a special case when a WebSocket to the OSI Pi server fails to initialize. This represents a more critical failure than closed or errored. It is published when, for example; the underlying operating system will not accept any more WebSocket connections on the AWS IoT Greengrass instance that is hosting the OSI Pi Streaming Data Connector.

In this case the WebSocket itself has failed to be created and no reference to it is retained.

Message Example:


```
{
  "id": 245,
  "route": "websocket-status-change",
  "status": 200,
  "response": {
    "channelId": "channel-abcdef123456",
    "websocketStatus": "opened",
    "event": "{}"
  }
}
```

OSI Pi Streaming Data Connector Telemetry

Publish Topic: osi-pi/streaming-data-connector/IOT-THING-NAME]/telemetry

Once initialized, the OSI Pi Streaming Data Connector will begin publishing asynchronous telemetry updates of the TQV upload rate and other relevant metrics of the components upload to AWS IoT Sitewise in the following message format:


```
{
  "id": 4150,
  "route": "system-telemetry-update",
  "status": 200,
  "response": {
    "timestamp": 1691105490045,
    "system": {
      "total_heap_size": 89776128,
      "total_heap_size_executable": 3932160,
      "total_physical_size": 85226400,
      "total_available_size": 1069786616,
      "used_heap_size": 52512120,
      "heap_size_limit": 1124073472,
      "malloced_memory": 532560,
      "peak_malloced_memory": 4220592,
      "does_zap_garbage": 0,
      "number_of_native_contexts": 1,
      "number_of_detached_contexts": 0,
      "total_global_handles_size": 106496,
      "used_global_handles_size": 78368,
      "external_memory": 10537700,
      "memoryPercentUsed": "7.99",
      "cpuUsage": {
        "user": 16246832000,
        "system": 866692000,
        "percent": 90
      }
    },
    "sitewise": {
      "publishPerSec": "166.80",
      "propAliasPerSec": "1668.00",
      "propValuesPerSec": "10077.50",
      "queuedPropAlias": 9350,
      "propValuesPerAlias": "6.04",
      "publishErrorCount": 7,
      "publishErrorReceived": {
        "ThrottlingException": "You've reached the maximum rate of
BatchPutAssetPropertyValue entries ingested per asset-property/data-stream."
       }
    "osipi": {
      "receivedPiPointsPerSec": "10901.00"
    }
  }
}
```

Where:

- **System Values:** NodeV8.getHeapStatistics() response from the OSI Pi Streaming Data connector process.
- **Sitewise:**
    - **publishPerSec**: Number of messages published to AWS IoT Sitewise peer second,
    - **propAliasPerSec:** Number of OSI Data tags (~ Sitewise Property Alias) that are being updated per second on average,
    - **propValuesPerSec:** Number of OSI Pi time series data points (~ Sitewise Property Values or TQVs) that are being uploaded per second on average,
    - **propValuesPerAlias:** Number of time series data points that are being updated per Sitewise Property Alias per second.
    - **publishErrorCount:** Number of errors returned from AWS IoT Sitewise publish actions in the Sitewise telemetry publish interval defined in the shadow-config file.
    - **publishErrorReceived:** Aggregated error messages returned from AWS IoT Sitewise publish actions in the Sitewise telemetry publish interval.

- **osipi:**
    - **receivedPiPointsPerSec:** Number of PiPoints received per sec for the duration of the telemetry publish interval.