SaaS OverCloud 기술문서 #17

엣지 컴퓨팅 패러다임을 지향하는 마이크로서비스 구조 기반의 IoT-Cloud SaaS 응용 스마트 에너지 서비스 설계 및 검증

Document No. SaaS OverCloud #17
Version 1.0
Date 2019-01-15
Author(s) GIST#1 Team

■ **문서의 연혁**

버전	날짜	작성자	비고
초안 - 0.1	2019. 01. 15	이승형, 권진철, 한정수	

본 문서는 2018년도 정부(미래창조과학부)의 재원으로 정보통신기술진흥센터의 지원을 받아 수행된 연구임 (No.R7117-16-0218, 이종 다수 클라우드 간의 자동화된 SaaS 호환성 지원 기술 개발)

The research was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP) (No.R7117-16-0218, Development of Automated SaaS Compatibility Techniques over Hybrid/Multisite Clouds)

Contents

엣지 컴퓨팅 패러다임을 지향하는 마이크로서비스 구조 기반의 IoT-Cloud SaaS 응용 스마트 에너지 서비스 설계 및 검증

그림 목차

SaaS OverCloud 기술문서 #17.
엣지 컴퓨팅 패러다임을 지향하는 마이크로서비스 구조 기반의 IoT-Cloud SaaS 응용 스마트 에너지 서비스 설계 및 검증

1. 엣지 컴퓨팅 패러다임 지향 마이크로서비스 구조 기반 IoT-Cloud SaaS 응용 서비스 연구

1.1. 연구의 개요

o 본 문서에서는 마이크로서비스 구조에 기반하여 IoT-Cloud SaaS 응용을 대상으로 서비스를 설계하여 배포하고 관리하는 컨테이너 오케스트레이션을 통해 서비스를 검증하는 과정을 다룬다. 대상이 되는 예제 IoT-Cloud 응용 서비스는 소규모 데이터 센터의 효율적인 에너지 사용 개념이 적용된 스마트 에너지 서비스이다. 이 과정에서 표준적인 IoT 오픈소스 도구들의 활용을 통해 IoT-Cloud 서비스 적용 대상의 호환성 및 확장성을 지원 가능성을 검증한다.

1.2. 연구의 배경

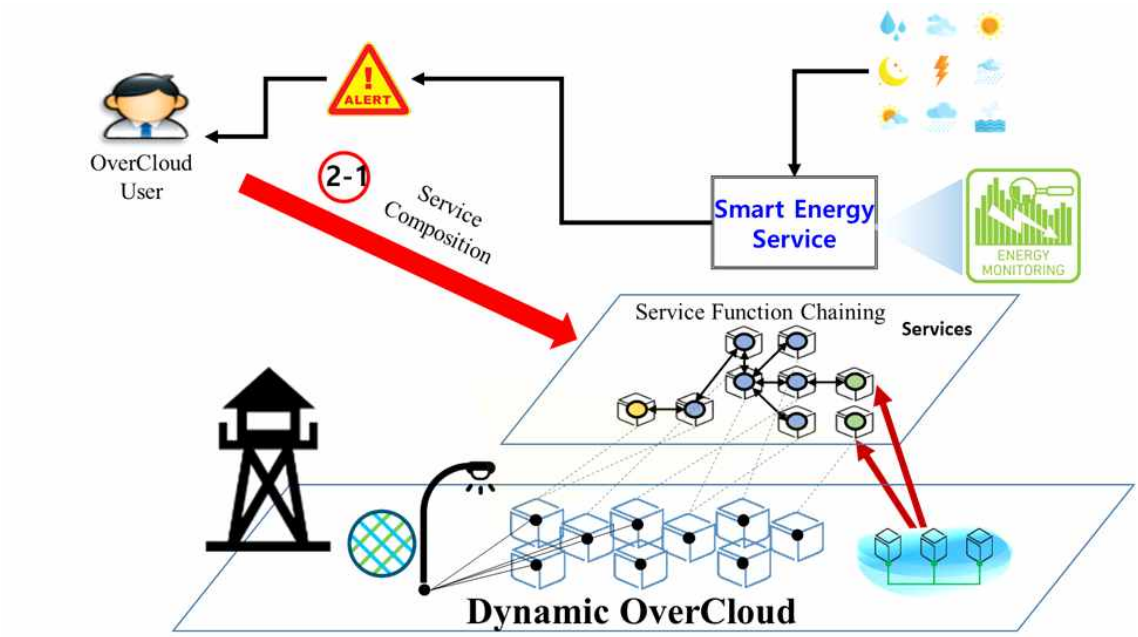

그림 1 컨테이너 오케스트레이션 기반의 서비스 합성을 바탕으로 하는 스마트 에너지 서비스 소프트웨어

o 기존의 마이크로서비스 구조로 구성되는 서비스의 각 요소 기능들은 VMware ESX, Linux KVM(Kernel-based Virtual Machine) 등의 하이퍼바이저 기반의 가상 머신을 활용하였지만, 최근에는 유연하고 경량화되어 효율적으로 분리된 컴퓨팅 자원의 활용을 지원할 수 있도록 하는 컨테이너 기반의 요소 기능들을 활용하여

서비스를 구성하는 추세로 전환되고 있다. 또한 IoT 서비스를 독립적으로 제공하는 것보다 클라우드와 연계되어 더 높은 가치를 내는 IoT-Cloud 서비스에 대한 수요가 급격하게 증가함에 따라 IoT 서비스에 컨테이너 기술의 적용은 더욱 중요해지고 있다. IoT 기기의 열악한 자원 보유 상황이 기존의 가상 머신 위주의 가상화 기술보다 경량화된 컨테이너 기술의 필요성을 더욱 촉진하고 있다.

o 최근의 마이크로서비스 구조 기반의 응용 서비스들은 컨테이너 오케스트레이션 기술을 활용하여 응용 서비스를 구성하는 다양한 요소 기능들을 배포하거나 조율하고 결합하고 있다. 자동화된 컨테이너 오케스트레이션의 목적은 여러 컨테이너의 배포 프로세스를 최적화하며, 컨테이너와 호스트의 수가 증가함에 따라 그 가치는 더욱 높아진다. 컨테이너 오케스트레이션은 컨테이너의 효율적인 자동 배포, 컨테이너 그룹별 로드 밸런싱, 컨테이너 장애 복구, 컨테이너 기능 서비스들 간의 네트워크 연결 제어와 같은 역할들을 포함한다.

o 본 문서에서는 자체적으로 지속 개발해온 시범적 IoT-Cloud SaaS 응용인 스마트 에너지 서비스를 대상으로 마이크로서비스 구조의 경량화된 컨테이너 기술에 기반한 IoT-Cloud 서비스를 다룬다. 먼저 대상 서비스를 작업 상황에 맞게 기능 단위로 나누고 대상 환경에 적절하게 배포하여 서비스 기능들을 배치시키고 기능들을 연결하는 일련의 과정을 거친다.

o 본 문서에서 대상으로 하고 있는 스마트 에너지 서비스는 엣지 컴퓨팅 패러다임을 지향하는 IoT-Cloud 서비스이다. 최근 IoT 서비스는 기존의 클라우드 컴퓨팅(Cloud computing)에서 엣지 컴퓨팅 패러다임(Edge computing paradigm)으로 전환되고 있는 추세이다. 단말기 디바이스에 더 가까운 곳에 위치하여 엣지 데이터센터와 소통하며 2차 작업물을 중앙 클라우드에 맡기는 방식이 데이터 처리속도를 향상시키고 효율적인 서비스를 지원할 수 있기 때문이다.

o IoT 서비스를 구성하는 다수의 기능들은 확장성과 호환성을 고려하여 표준적인 소프트웨어로써 동작하는 것이 바람직하다. IoT 서비스는 일반적으로 실시간의 다양한 디바이스로부터 센서 데이터를 가져와서 축적하며, 모니터링하고 다시 디바이스 측으로 명령을 내리는 비슷한 패턴(Pattern)들을 가지고 있다. 본 문서에서 다루고자 하는 IoT-Cloud SaaS 응용에 대한 스마트 에너지 서비스는 이와 같은 패턴을 갖는 서비스를 쉽고 빠르게 개발하고 확장성과 호환성에 대응할 수 있도록 엣지 디바이스(Edge Device)부터 게이트웨이까지 필요한 요소 기능들을 표준적인 프레임 워크를 사용하여 개발하고 자체적으로 실증하였다. IoT에서의 표준적인 기능 개발은 서비스의 상호운용성을 확보하는 측면에서 큰 이점을 가져올 것으로 기대할 수 있다.

1.3. 연구의 필요성

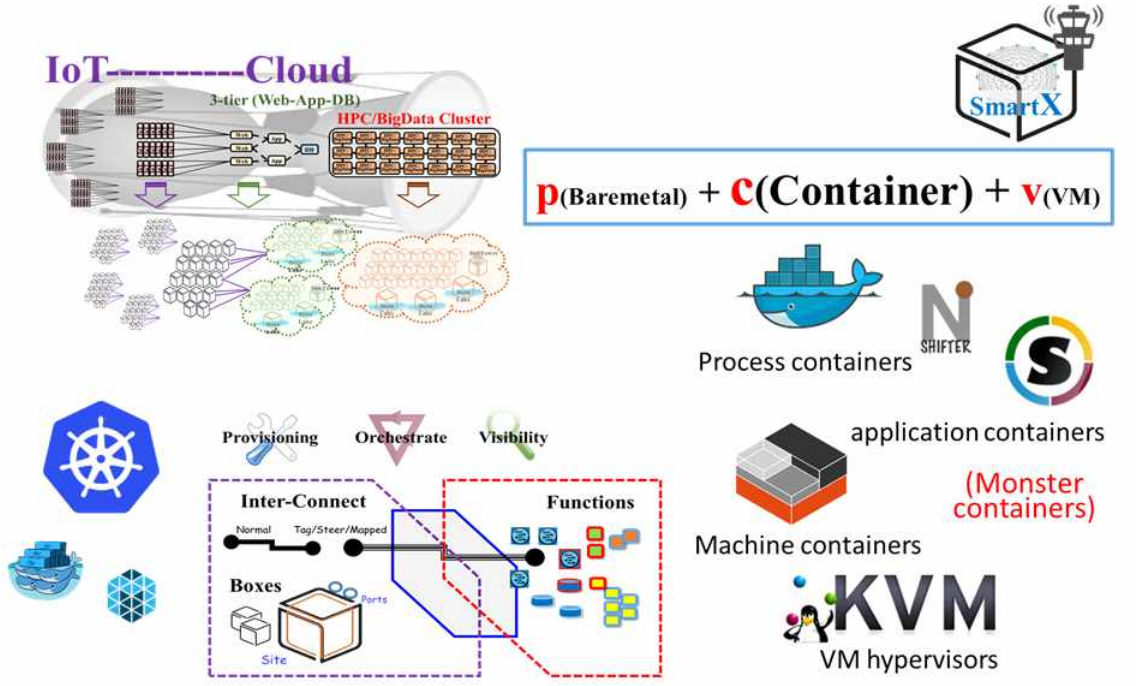

그림 2 마이크로서비스 구조 기반 IoT-Cloud 를 대상으로 하는 SaaS 응용과 컨테이너 오케스트레이션

o 사물 인터넷과 빅데이터의 영향으로 데이터의 중요성이 날로 더해가고 있으며, 방대한 데이터를 수용하고 처리하기 위한 클라우드 컴퓨팅 기술과 사물 인터넷이 연계된 서비스에 대한 수요가 폭발적으로 증가하고 있다. 이에 따라 IoT-Cloud 기반의 서비스에 대한 이해가 필수적이며, 그 과정에서 컨테이너 기반의 가상화 기술의 활용이 무엇보다 중요한 시점이다. 마이크로서비스 기반의 대상 IoT-Cloud 서비스를 다양한 환경에서 활용하기 위해서는 호환성을 향상시킴으로써 다양한 환경에서 반복되는 동일 작업으로 낭비되는 시간과 노력의 절감이 필요하다. 이를 위해 서비스를 이루고 있는 컨테이너 기능들을 컨테이너 오케스트레이션을 통해 자동으로 배포하고 관리하며 안전하게 유지할 수 있는 서비스 운영 방안의 모색이 필요하다.

2. 엣지 컴퓨팅 패러다임을 지향하는 마이크로서비스 구조 기반 스마트 에너지 서비스

2.1. IoT-Cloud SaaS 응용 개발 방향

o 본 문서에서는 IoT-Cloud SaaS 응용의 예로써 Web-App-DB로 구성되는 3-tier SaaS 응용을 스마트 에너지 서비스라는 자체적인 서비스 소프트웨어에 대한 개발 및 시범적인 실증 경험을 바탕으로 서술한다. 마이크로서비스 기반의 컨테이너 기술을 통해 IoT-Cloud SaaS 응용에 대한 호환성 확보를 고려하였으며, 엣지 컴퓨팅 패러다임을 지향하는 표준적인 IoT-Cloud SaaS 응용 방안 마련을 위해 리눅스 재단의 엣지엑스 파운드리(EdgeX Foundry)의 오픈소스 프레임워크에 기반하여 서비스 기능들을 개발하였다. 전체적인 서비스의 개발 및 실증은 운영 자동화 측면에서 필수적인 서비스의 배포와 합성과 관련된 내용을 중점적으로 다룬다. 해당 서비스는 동적으로 생성 및 소멸이 가능한 OverCloud 환경 위에서 운용되며 이러한 환경은 OverCloud를 구성하는 DevOps Post, DataLake 등의 개체를 포함하며 UnderCloud와의 관계를 매개하는 유연한 클라우드 환경임을 전제로 한다.

2.2. 엣지 컴퓨팅 패러다임을 지향하는 IoT-Cloud SaaS 응용 지원을 위한 요구사항

o IoT-Cloud 서비스를 마이크로서비스 기반의 SaaS 응용으로 구성하기 위해서 요소 기능들은 모두 도커 컨테이너를 통해 구성되어야 한다. 서비스를 구성하는 컨테이너 기반의 분리된 요소 기능들은 마이크로서비스 형태에 부합하도록 설계되어야 하며, 이를 통해 서비스 구성 기능들을 개발하여 배포하고 테스트 할 수 있다.

o 해당 IoT-Cloud 서비스를 마이크로서비스 기반의 SaaS 응용으로 구성하기 위한 요소 기능들을 도커 컨테이너로 구성하기 위해서는 개발한 기능별로 도커 이미지를 만들 수 있도록 하는 빌드 작업이 필요하다. 각 기능들이 활용하는 소프트웨어 패키지 의존성이나 담고 있는 소스 코드의 내용을 명세서에 담아서 빌드를 통해 이미지를 만들고나면 컨테이너 오케스트레이션을 통해 기능들을 명세한대로 배치하고 체이닝을 거쳐 서비스 합성을 진행할 수 있다.

o 엣지 컴퓨팅 패러다임을 지향하는 IoT-Cloud SaaS 응용은 실제 단말 디바이스인 IoT 뿐만 아니라 엣지 클라우드와 IoT 디바이스 사이에 배치된 IoT-Gateway라는 물리적인 컴포넌트와 그 안에 배치되는 기능들을 포함하는 IoT-Cloud Hub라는 소프트웨어적인 컴포넌트가 필요하다. 본 문서에서 대상 서비스에 활용하는 IoT-Cloud Hub라는 소프트웨어적인 컴포넌트는 표준적인 엣지 컴퓨팅을 위한

요소 기능들을 담고 있다.

2.3. 엣지 컴퓨팅 패러다임을 지향하는 IoT-Cloud SaaS 응용 기능 설계

o 앞 절에서 명시한대로 IoT-Cloud SaaS 응용 기능을 개발할 때 엣지 컴퓨팅의 패러다임을 지향한 서비스를 위해 EdgeX Foundry의 표준적인 오픈소스 프레임워크를 활용한다. EdgeX Foundry의 프레임워크를 통해 IoT-Cloud 대응 서비스 기능들의 개발 부담을 줄여 서비스 환경 구성을 용이하게 할 수 있다. 또한 EdgeX Foundry의 오픈소스 프레임워크는 IoT 기기의 종류나 서버의 하드웨어, 운영체제, 응용에 종속되지 않고 개방적으로 활용할 수 있다. 다시 말해서, 마이크로서비스 지향 프레임워크를 대상 환경에 필요한 만큼 활용하여 다양한 IoT-Cloud 서비스를 제공하는 서비스 환경을 표준적으로 간편하게 구성할 수 있다.

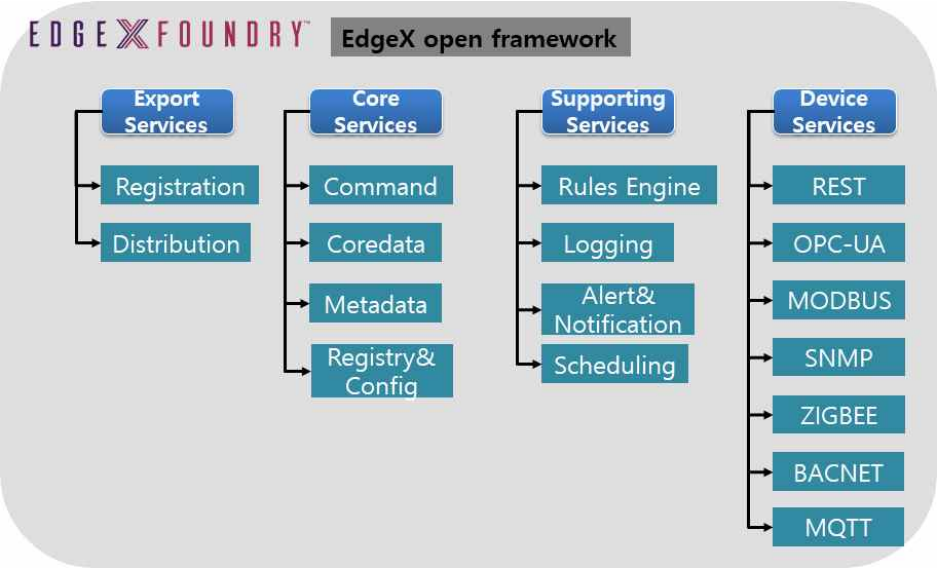

그림 3 EdgeX Foundry의 오픈소스 프레임워크 구조

o EdgeX Foundry의 프레임워크는 크게 Export Services, Device Services, Core Services, Supporting Services와 같이 구조화된다. 각각의 서비스 구조 레벨에는 각기 다른 기능들을 가지고 있다. 본 논문에서는 실제로 IoT-Cloud SaaS 응용 서비스 기능 개발에 활용된 Core Services와 Supporting Services를 중심으로 기술한다.

o 기본적으로 모든 EdgeX Foundry에서 제공하는 프레임워크를 활용하는 서비스 기능들은 MongoDB를 기반으로 동작한다. 수집되는 센서 데이터나 IoT 디바이스에 대한 정보가 저장되는 것뿐 아니라 기능들이 수행되며 남기는 로그까지 모두

MongoDB에 저장된다. EdgeX Foundry 프레임워크를 활용하여 서비스를 합성하기 위해서는 필요로 하는 모든 Collection들이 MongoDB 내에 만들어져야 하지만 서비스들을 접합하면 자동적으로 Collection들이 생성되어 서비스에 연계된다.

o Core Services는 Command, Core Data, Metadata, Registry&Config와 같은 마이크로서비스 구조 기반의 기능들로 이루어져 있으며, 시험적인 IoT-Cloud SaaS 응용 서비스에서는 Command, Core Data, Metadata 기능들을 활용하였다. Core Data는 실질적인 데이터들이 EdgeX 서비스에 수집될 수 있도록 하는 기능을 한다. Core Data는 MongoDB 내에 Event, Reading, Value Descriptor 클래스들을 정의하고 동작한다. 먼저 Value Descriptor는 EdgeX 서비스에서 수집할 수 있는 IoT 디바이스의 데이터 형식을 저장하며 저장되지 않은 형식의 데이터는 분류하여 에러 처리한다. 또한 디바이스로부터 오는 데이터를 저장할 때 모든 발생하는 이벤트 로그(Event Log)를 Event 저장하며, Reading에는 실질적인 데이터가 저장되게 된다. Metadata의 경우에는 Core Data 기능과 연계되어 IoT 디바이스의 프로필을 등록하여 서비스에 활용할 수 있도록 하며, Command 기능의 경우 IoT 디바이스를 등록한 후 EdgeX Foundry에서 제공하는 API를 활용하여 서비스를 개발할 수 있게 한다.

o Supporting Services 에서는 Rule을 정해놓고 IoT 디바이스에 피드백을 줄 수 있는 Rules Engine, 데이터 수집과 동시에 발생하는 event 데이터 자체에 대해 자동으로 삭제하거나 정리할 수 있는 Scheduling, 사전에 연결된 서비스 사용자(User)에게 SMS나 Mail서비스를 할 수 있는 Alerts and Notifications 기능들이 있지만 EdgeX 서비스에서 제공하는 Restfule API를 활용하여 데이터 입출력시에 생기는 로그(Log)와 타임스탬프(Temestamp)등을 활용할 수 있도록 하는 Logging 기능을 활용한다.

2.4. 서비스 합성 설계

o 대상으로 하는 IoT-Cloud SaaS 응용 서비스는 컨테이너 오케스트레이션을 통해 컨테이너 기반의 자동화된 서비스 합성을 효율적으로 수행할 수 있도록 하기 위해서 서비스를 구성하는 요소 기능들을 운용 환경의 특성을 파악하여 적절한 환경에 대응시키는 매칭(matching)과정과 각 기능들의 네트워킹의 유연성을 고려한 배포(placement)과정 및 서비스 기능들을 서비스 명세에 따라 연결시키는 접합(stitching)과정 및 유동적으로 변화하는 서비스에 대응하는 서비스 튜닝(service tuning) 등과 같은 서비스 합성 절차가 필요하다. 그 중에서도 본 문서에서는 기능들의 매칭, 배포와 접합을 통한 서비스 합성 절차에 집중하여 기술한다.

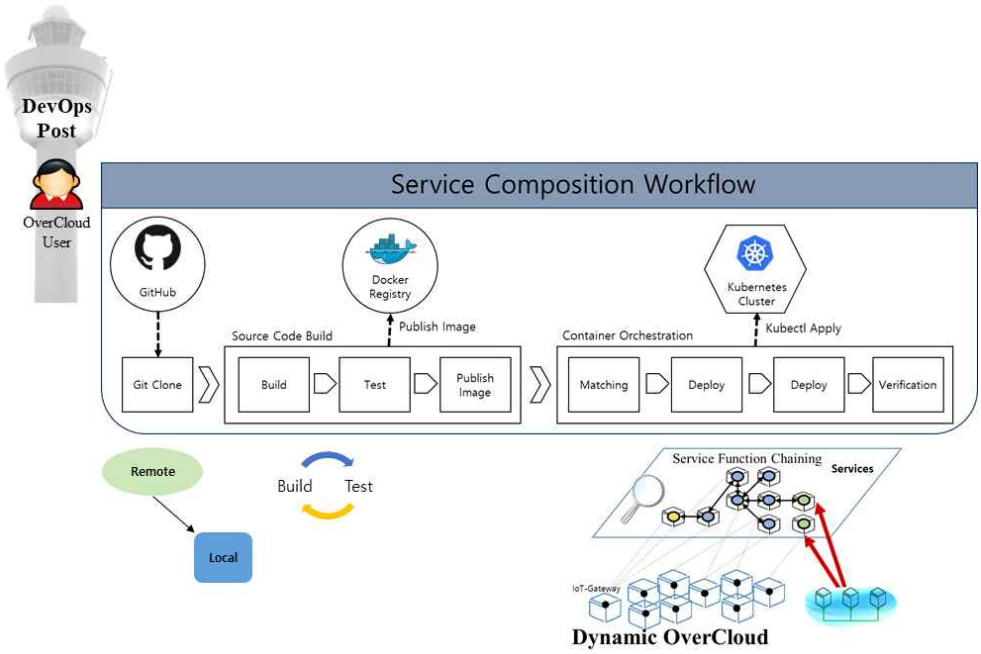

그림 4 IoT-Cloud SaaS 응용 Service Composition Workflow

o 본 논문에서 대상으로 하고 있는 IoT-Cloud SaaS 응용 서비스를 위해 목적하는 서비스 기능들이 대상 환경의 특성을 파악하고 적절하게 배치될 수 있도록 하는 매칭 절차는 매우 중요하다. 기능 매칭 절차에서는 적절한 기능을 선택해서 확보가 가능한 여유 자원의 유형/규모에 맞추고, 컨테이너 설치, 설정 유연성 등을 확인하게 된다.

o 엣지 컴퓨팅을 위해 활용하고 있는 IoT-Cloud Hub 소프트웨어는 IoT 디바이스 단과 가까운 곳에 위치하는 물리적인 컴포넌트인 IoT-Gateway에 배포되어야 한다. IoT-Cloud Hub 소프트웨어 기능들을 적소에 배치하기 위해서는 컨테이너 오케스트레이션을 위한 Cluster 내에서 사전에 각각의 컨테이너가 배치될 노드들에 대한 정보를 가지고 설정하여 배포하도록 한다.

o 서비스 기능 접합 절차에서는 배포할 기능들을 명세에 따라서 다른 기능들과 어떤 방식으로 연결할지 결정함으로써 서비스 합성을 마무리한다. 명세에 따라 서비스 접합이 된 기능들은 상호 연결되어 서로 데이터를 주고받는 것이 가능해지고 전체 기능들은 통합적인 서비스로 운용된다. 즉 SFC(Service Function Chaining)을 통해 기능들을 유연하게 연계하여 컨테이너 기반 서비스 합성을 위한 기능 접합이 서비스 기능들의 변동에 유연하게 대응하도록 지원할 수 있다.

3. 스마트 에너지 서비스 설계를 통한 IoT-Cloud SaaS 실증 방안

3.1. 서비스 개요

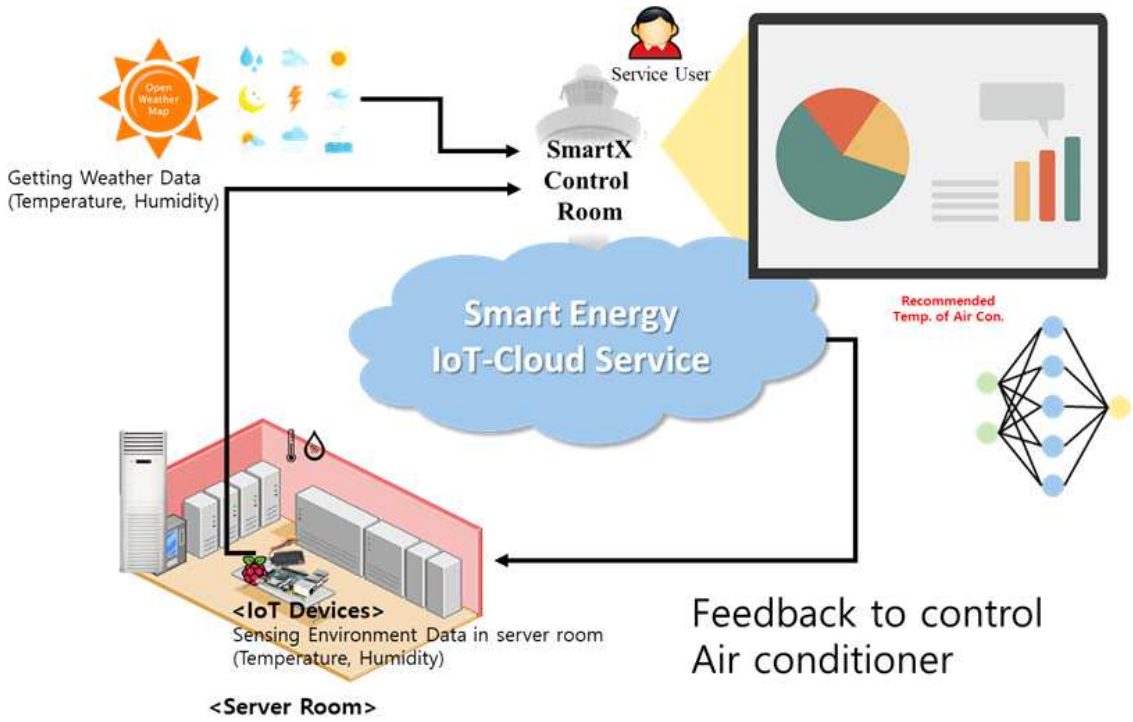

그림 5 스마트 에너지 서비스 시나리오

o 스마트 에너지 서비스는 데이터 센터 내부에 있는 서버실의 온도를 조절하는 에어컨의 희망 온도를 서버실 내부와 외부의 환경에 따라 조절하며 서비스를 위한 데이터들을 모니터링하여 에너지를 효율적으로 절약하여 운영될 수 있도록 하는 서비스이다. 예를 들어, 실내 공기의 온도가 상승했다면 이를 낮춰주기 위하여 에어컨의 온도를 보다 낮게 설정하여 서버들의 동작을 원활하게 유지하여줄 필요가 있지만 이미 서버실 내의 온도가 충분히 낮은 상태라면 에어컨의 온도를 낮추는 데에 에너지를 쓰지 않더라도 서버들의 동작에 무리가 없다. 이처럼 스마트 에너지 서비스를 통해 서버실 운영에 드는 에너지를 효율적으로 조절할 수 있다.

o 그림 5는 스마트 에너지 서비스의 시나리오를 나타낸다. 그림에서 볼 수 있는 것처럼 스마트 에너지 서비스는 IoT 디바이스의 센서를 통한 측정 기능, IoT-Gateway에서 IoT 디바이스를 통제하는 기능, 서비스 데이터 저장 기능, 모니터링 기능 그리고 머신러닝을 통한 분석 기능과 분석된 모델을 통해 실시간으로 들어오는 데이터를 만들어진 모델에 적용하여 에너지를 절약하여 운영될 수 있도록 하는 기능들로 이루어진다. 이와 같이 표준화된 기능들의 활용을 통해서 서버

실 내의 온습도 및 외부 날씨정보들을 파악하고 데이터를 수집하여 신속하고 안정적이며 확장성이 높은 서비스를 운영할 수 있다. 또한 수집된 데이터를 가시화를 통해 서버실의 현재 상황과 추천되는 서버실 에어컨의 온도 등을 모니터링할 수 있다. 이를 통해 효율적인 에너지의 사용과 안정적인 서비스 운영 또한 가능할 것으로 기대한다.

3.2. 서비스 상세 설명

o 스마트 에너지 서비스는 마이크로서비스 기반의 컨테이너 기술로 기능들을 알맞게 분리시켜서 독립적인 환경에서 개별 기능들이 문제없이 실행될 수 있고, 분리된 기능들을 일목요연하게 엮어내는 연결을 통해 하나의 효과적인 서비스로 만들고 안정적으로 운영하는 것을 목표로 하였다. 이를 위해 그림 6과 같은 서비스 기능들을 설계하고 구현하여 시험적으로 실증하였다. 각 기능들은 쿠버네티스 오케스트레이션을 기반으로 하여 Tower, IoT-Cloud Hub, Data Storage, IoT device에 이르는 4가지 노드별로 필요에 따라 다르게 배포된다.

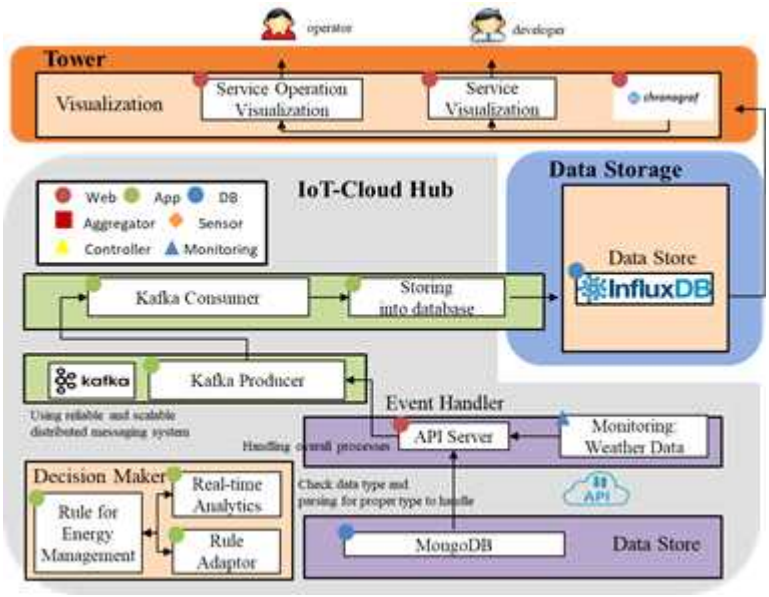

그림 6 스마트 에너지 서비스 기능 구성도

o 그림 6에서 보여지는 것처럼 스마트 에너지 서비스는 IoT 디바이스, 클라우드 내의 노드들과 그 사이를 중계하며 의사소통을 원활하게 하는 IoT-Gateway를 포함하는 서비스 환경을 기반으로 요소 기능들이 배치되어 수행된다. IoT 디바이스에서는 데이터를 측정하는 기능을 EdgeX Foundry의 프레임워크를 통해 구현하였고, IoT-Gateway에 배치되는 기능들은 IoT 디바이스로부터 전달되는 데이터를 받아서 임시 저장할 수 있는 mongoDB를 두고 실시간 날씨 정보와 데이터를 함께

가져와서 Kafka 메시징 시스템을 통해 클라우드에 있는 InfluxDB에 저장하도록 구현하였다. 또한 InfluxDB에 저장된 데이터를 chronograf에서 가시화를 통해 수집되는 데이터와 추천받는 데이터를 확인할 수 있도록 하였다. 위의 요소 기능들을 그림 6과 같이 정의하여 서비스 시나리오에 따라 원활하고 안정적인 서비스로 운용할 수 있도록 설계하고 시험적으로 구현하였다.

o IoT 디바이스 측에서는 에어컨의 온도를 효율적으로 제어하기 위해 Raspberry Pi를 활용하여 서버실 내부의 온도 및 습도 정보를 수집하였다. 먼저, 엣지 컴퓨팅 패러다임을 지향하는 서비스를 위해 EdgeX Foundry에서 제공하는 API를 활용하여 IoT 디바이스 프로필과 데이터 형식을 등록한다. EdgeX 서비스에 등록할 프로필을 기기의 확장성을 고려하여 작성하였다. 등록이 되면 온도와 습도를 EdgeX 서비스 측에 Restful API를 활용하여 10초마다 측정하여 보내줄 수 있도록 한다. 또한 IoT 디바이스에서 데이터를 송신할 때는 IoT-Cloud Hub의 API_server에 요청을 보내서 데이터를 중계해주는 Kafka에 전달할 수 있도록 한다.

```
import requests, json
import Adafruit_DHT as dht
import time
import requests, json
import Adafruit_DHT as dht
import time
import os
import sys
import subprocess
from time import localtime, strftime

class EdgeX:
    def __init__(self, ip='IoT-Cloud Hub IP address'):
        self.ip = ip
        self.port = 48080
        self.metaIp = ip
        self.metaPort = 48081
        self.headers = {'Content-Type': 'application/json', 'Access-Control-Allow-Origin':
  '*','Access-Control-Allow-Headers': '*', 'Access-Control-Allow-Methods' : 'POST, GET,
  OPTIONS'}

    def DataTemplate(self):
        deviceTemplate = '{"name":"device","formatting":"%s"}'
        tempTemplate = '{"name":"temperature","min":"-40","max":"140","type":"F","uomLab
  el":"degree cel","defaultValue":"0","formatting":"%s","labels":["temp","hvac"]}'
        humidTemplate = '{"name":"humidity","min":"0","max":"100","type":"F","uomLabel":
  "per","defaultValue":"0","formatting":"%s","labels":["humidity","hvac"]}'
        dateTemplate = '{"name":"time","formatting":"%s","labels":["time","YYYYMMDD H
  HMMSS"]}'
        url = 'http://%s:%d/api/v1/valuedescriptor' % (self.ip, self.port)
        response = requests.post(url, data=deviceTemplate, headers=self.headers)
        response = requests.post(url, data=tempTemplate, headers=self.headers)
        response = requests.post(url, data=humidTemplate, headers=self.headers)
        response = requests.post(url, data=dateTemplate, headers=self.headers)
```

그림 7 EdgeX Foundry 프레임워크 활용을 위한 데이터 형식 등록

```
def createDevice(self, deviceName):
        print('Creating addressable for ', deviceName)
        addresableData = '{"origin":1471806386919,"name":"%s","protocol":"HTTP","address":
    "","port":"161","path":"","publisher":"none","user":"none","password":"none","topic":"none"}'
    % (
        deviceName,)
        url = 'http://%s:%d/api/v1/addressable' % (self.metaIp, self.metaPort)
        response = requests.post(url, data=addresableData, headers=self.headers)
        if response.status_code != 200:
            print(response.status_code)
            print(response.content)
        else:
            print('OK')
        print('Creating addressable for service')
        serviceAddresableData = '{"origin":1471806386919,"name":"%s-address","protocol":"
    HTTP","address":"","port":"49989","path":"","publisher":"none","user":"none","password":"no
    ne","topic":"none"}'%(deviceName,)
        url = 'http://%s:%d/api/v1/addressable' % (self.metaIp, self.metaPort)
        response = requests.post(url, data=serviceAddresableData, headers=self.headers)
        if response.status_code != 200:
            print(response.status_code)
            print(response.content)
        else:
            print('OK')
        print('Creating service')
        serviceData = '{"origin":1471806386919,"name":"edgex-%s","description":"temperatur
    e service for rooms","lastConnected":0,"lastReported":0,"labels":["snmp","rtu","io"],"adminS
    tate":"unlocked","operatingState":"enabled","addressable":{"name":"%s-address"}}'%(device
    Name, deviceName,)
        url = 'http://%s:%d/api/v1/deviceservice' % (self.metaIp, self.metaPort)
        response = requests.post(url, data=serviceData, headers=self.headers)
        if response.status_code != 200:
            print(response.status_code)
            print(response.content)
        else:
            print('OK')

edgex = EdgeX()
edgex.DataTemplate()
edgex.createDevice("Device1")
```

그림 8 EdgeX Foundry 프레임워크 활용을 위한 디바이스 프로필 등록

```
while True:
    date = strftime("%y.%m.%d-%H:%M:%S", localtime())
    h, t = dht.read_retry(dht.DHT22, 4)
    if t <= 35:
        print 'Temp={0:0.1f}*C Humidity={1:0.1f}%'.format(t, h)

        url = "http://IoT-Cloud Hub IP address:48080/api/v1/event"

        payload = {"origin": 1471806386919, "device": "Device1",
                   "readings": [{"origin": 1471806386919, "name": "device", "value": 'Devi
ce1'},
                                {"origin": 1471806386919, "name": "time", "value": date},
                                {"origin": 1471806386919, "name": "temperature", "value
": t},
                                {"origin": 1471806386919, "name": "humidity", "value":
h}]}

        headers = {"Accept": "application/json", "Content-Type": "application/json"}

        response = requests.post(url, data=json.dumps(payload), headers=headers)

        print(response.text)
        print(response.headers)

        time.sleep(100)   ##delay time 10 seconds

        url = "http://IoT-Cloud Hub IP address:5000" ##flask에 api 요청
        response = requests.get(url, headers=headers)
```

그림 9 EdgeX Foundry 프레임워크를 활용한 데이터 측정 및 전송

o 등록된 디바이스에 의해 수집된 데이터들은 모두 IoT-Cloud Hub의 mongoDB 내에 임시로 저장되게 된다. API Server에서는 IoT 디바이스의 요청을 받아서 mongoDB에 접근한다. 이때 Pymongo를 활용하며, 데이터를 가져와서 Kafka broker에 데이터를 전달하도록 한다. 이 과정에서 Data 형식을 토픽 별로 정리될 수 있도록 Parsing을 함께 수행한다. 그리고 나서 Apache Kafka에 수집된 데이터들은 Real-time data sender라는 Kafka consumer를 통해 클라우드에 있는 InfluxDB에 전달된다. Kafka는 메시지를 파일 시스템에 저장함으로써 대용량의 데이터에 대응하여 빠른 처리가 가능하며, 분산 시스템으로 설계된 까닭에 견고하며 유연하게 활용할 수 있다. 이러한 구조적인 특성에서 오는 이점을 바탕으로 상대적으로 부족한 데이터 연산 및 처리 능력을 가진 IoT 기기들이 데이터를 전달할 때 생기는 과부하를 최소화할 수 있으며, 동시에 여러 디바이스로부터 다수의 데이터 메시지가 수집되더라도 빠르게 대응할 수 있도록 하였다.

o InfluxDB에 저장되는 데이터들은 CSV(Comma-seperated values) 파일형식으로 출력하여 Machine Learning을 위한 클러스터에 전달하면 기계 학습을 통해 모델을 생성한다. 그리고 생성된 모델에 기반하여 Inference server를 구축하여 서비스에 적용한다. Kafka consumer에서는 Kafka로부터 가져온 데이터를 InfluxDB에 전달할 때 Inference server에도 함께 전달한다. 이 때, 외부 습도, 외부 온도, 실내 습도, 실내 온도, 날씨 정보의 파라미터를 하나의 Set으로 총 여섯 Set의 데이터를 함께 Inference server에 전달한다. 그리고 출력되는 결과를 다시 InfluxDB에 전달하여 에어컨 온도 조정을 효율적으로 할 수 있는 Inference 값을 함께 저장한다.

```
import os
import subprocess
import sys
import pymongo
import requests
from pymongo import MongoClient
from flask import request
from flask import Flask
import msgpack
from kafka import KafkaProducer
from kafka.errors import KafkaError

SELECTOR_mongo = 'edgex-mongo'
SELECTOR_kafka = 'kafka-broker-1'
app = Flask(__name__)
producer = KafkaProducer(bootstrap_servers=['kafka-broker-1:9092'])
#topic = 'Device1'

@app.route('/')
def index():
    connection = pymongo.MongoClient(SELECTOR_mongo, 27017)
    # connection = pymongo.MongoClient("localhost", 27017)
    db = connection.coredata
    read_data = db.reading.find().limit(4)
    str1 = str()
    for data in read_data:
        str1 = str1 + "{}:{}, ".format(data['name'], data['value'])
    print(str1)
    str2 = str1.split(',')
    device_name = str2[0]
    str4 = str2[1]
    str5 = str2[2]
    str6 = str2[3]
```

```
    str4 = str4.split(' ')
    time_val = str4[1]
    str5 = str5.split(' ')
    temperature = str5[1]
    str6 = str6.split(' ')
    humidity = str6[1]
    device_name2 = device_name.split(':')
    device_name = device_name2[1]
    producer.send(device_name, str1)
    db.reading.remove({})
    return 'connection is good'

if __name__ == '__main__':
   # app.debug = True
    app.run(host='0.0.0.0')
```

그림 10 IoT 디바이스에서 수집된 데이터를 Kafka에 전달하기 위한 flask_api_server.py 스크립트

o Kafka broker까지 데이터를 전달하였다면 최종 데이터 저장소인 클라우드의 InfluxDB 까지 데이터를 송신하여야 한다. 이를 위해 Real time data sender라는 Kafka consumer 컴포넌트를 활용한다. 이 때, 디바이스들과 현재 날씨 정보를 동기화 시켜서 함께 전달하며 날씨 정보 활용에는 오픈소스 API를 제공하는 OWM(Open Weather Map)을 활용한다. 이 때 광주지역의 날씨를 표본으로 하며 온도는 섭씨, 습도는 상대습도를 기준으로 외부 날씨 정보를 수집한다. 날씨 상태 정보는 정도에 따라 Thunderstorm, Drizzle 등 9가지로 분류하며 데이터 분석을 위해 수치화 하였다.

```
import threading, logging, time
import multiprocessing
import msgpack
from kafka import KafkaProducer
from kafka import TopicPartition
from kafka import KafkaConsumer
from kafka.errors import KafkaError
import requests, json
import time
import os
import sys
import subprocess
import urllib, urllib2
from time import localtime, strftime
from pyowm import OWM

cmd ="curl -XPOST 'influxdb:8086/query' --data-urlencode 'q=CREATE DATABASE 'Sen
   sordata''"
subprocess.call([cmd], shell=True)
timeout = 100
actual_data=[]
consumer = KafkaConsumer('Device1', bootstrap_servers=['kafka-broker-1:9092'])
partitions = consumer.poll(timeout)
while partitions == None or len(partitions) == 0:
          consumer = KafkaConsumer('Device1', bootstrap_servers=['kafka-broker-1:9092'])
          message = next(consumer)
          str1 = message.value
          str2 = str1.split(',')
          device_name = str2[0]
          str4 = str2[1]
          str5 = str2[2]
          str6 = str2[3]
          str4 = str4.split(' ')
          time_val = str4[1]
          str5 = str5.split(' ')
          temperature = str5[1]
          str6 = str6.split(' ')
          humidity = str6[1]
```

```
device_name2 = device_name.split(':')
        device_name = device_name2[1]
        time_val2 = time_val.partition(':')
        time_val = time_val2[2]
        temperature2 = temperature.split(':')
        temperature = temperature2[1]
        humidity2 = humidity.split(':')
        humidity = humidity2[1]
        #variables = "dev1"
        Aircon_temp = '18'
        API_key = '5ae0b0ddfba53bb1acc0ec416e931177'  ## information for weather co
  nfigure
        owm = OWM(API_key)
        obs = owm.weather_at_place('Gwangju')
        obs = owm.weather_at_coords(35.15972, 126.85306)
        w = obs.get_weather()
        weather_stat = w.get_status()
        ext_temperature = w.get_temperature(unit='celsius')['temp']
        ext_temperature = "{0:0.2f}".format(ext_temperature)
        ext_humidity = w.get_humidity()
        ext_humidity = "{0:0.2f}".format(ext_humidity)
         if weather_stat=='Thunderstorm':
                weather_stat = 1
         if weather_stat=='Drizzle':
                weather_stat = 2
         if weather_stat=='Rain':
                weather_stat = 3
         if weather_stat=='Snow':
                weather_stat = 4
         if weather_stat=='Atmosphere':
                weather_stat = 5
         if weather_stat=='Clear':
                weather_stat = 6
         if weather_stat=='Clouds':
                weather_stat = 7
         if weather_stat=='Mist':
                weather_stat = 8
         if weather_stat=='Haze':
                weather_stat = 9
```

```
        cmd = "curl -XPOST 'influxdb:8086/write?db=Sensordata' --data-binary 'dev1,loc
ation=Gwangju temperature=%s,humidity=%s,Aircon_temp=%s,ext_temperature=%s,ext_
humidity=%s,weather_stat=%s'" % (temperature, humidity, Aircon_temp, ext_temperat
ure, ext_humidity, weather_stat)
        subprocess.call([cmd], shell=True)
        int_ext_humid=int(float(ext_humidity)) #형변환
        int_ext_temp=int(float(ext_temperature))
        int_humid=int(float(humidity))
        actual_data[i] = '{0},{1},{2},{3},{4}'.format(int_ext_humid, int_ext_temp, int_humid,
'24', weather_stat)
        i = i+1
        if i == 6:
                print("%s, %s") % ('----------', actual_data[0])
                print("%s, %s") % ('----------', actual_data[1])
                print("%s, %s") % ('----------', actual_data[2])
                print("%s, %s") % ('----------', actual_data[3])
                print("%s, %s") % ('----------', actual_data[4])
                print("%s, %s") % ('----------', actual_data[5])
                data_0 = actual_data[0]
                data_1 = actual_data[1]
                data_2 = actual_data[2]
                data_3 = actual_data[3]
                data_4 = actual_data[4]
                data_5 = actual_data[5]
                cmd = "curl -d '{\"inputs\": [[[%s],[%s],[%s],[%s],[%s],[%s]]]}' -X POST i
nference-server-service:9000/v1/models/service:predict" % (data_0, data_1, data_2, data
_3, data_4, data_5)
                print(cmd)
                result = subprocess.check_output([cmd], shell=True)
                print(result)
                result = json.loads(result)
                data_output =result["outputs"][0]
                print(data_output)
                recommend_data = int(round(data_output))
                print(recommend_data)

                cmd = "curl -XPOST 'influxdb:8086/write?db=Sensordata' --data-binary
'dev1,location=Gwangju recommend_temperature=%s'" % (str(recommend_data))
                subprocess.call([cmd], shell=True)
                i = 0
```

그림 11 Real time data sender 기능을 위한 스크립트 명세

o 컨테이너 오케스트레이션을 통한 스마트 에너지 서비스를 위해서는 kubernetes에서 활용할 수 있는 yaml 파일로 서비스 기능들에 대해 명세가 되어야 한다. 이를 위해 위 기능들의 스크립트를 실행할 수 있는 도커 기반의 컨테이너 이미지들이 만들어 져야 하며 각각의 기능을 담는 컨테이너 이미지를 만들기 위한 Dockerfile들의 명세는 아래와 같다.

```
#!/bin/bash
FROM ubuntu:16.04
MAINTAINER shlee <lshyeung@gmail.com>

RUN apt-get update
RUN apt-get install git -y
RUN git clone https://github.com/SmartX-Team/SmartX-MicroBox /tmp/SmartX-MicroBox
RUN apt-get install -y libcurl3 openssl curl
RUN apt-get install -y python2.7 python-pip
RUN apt-get install -y python3-pip
RUN pip install --upgrade pip
RUN pip install requests
RUN pip install pymongo
RUN pip install flask
RUN pip install kafka-python
RUN pip install queuelib
RUN pip install influxdb
#RUN pip install pandas
RUN pip install msgpack
RUN pip install pyowm

CMD ["/usr/bin/python", "/tmp/SmartX-MicroBox/application_functionality/Smart-Energy-service/IoT-Cloud-Hub_functions/API_server/flask_api_server_ver2.py"]
```

그림 12 API server 도커 이미지 빌드를 위한 Dockerfile 스크립트 명세

```
#!/bin/bash
FROM ubuntu:16.04
MAINTAINER shlee <lshyeung@gmail.com>

RUN apt-get update
RUN apt-get install git -y
RUN git clone https://github.com/SmartX-Team/SmartX-MicroBox /tmp/SmartX-MicroB
   ox
RUN apt-get install -y libcurl3 openssl curl
RUN apt-get install -y python2.7 python-pip
RUN apt-get install -y python3-pip
RUN pip install --upgrade pip
RUN pip install requests
RUN pip install pymongo
RUN pip install flask
RUN pip install kafka-python
RUN pip install queuelib
RUN pip install influxdb
#RUN pip install pandas
RUN pip install msgpack
RUN pip install pyowm

CMD ["/usr/bin/python", "/tmp/SmartX-MicroBox/application_functionality/Smart-Energ
   y-service/IoT-Cloud-Hub_functions/consumer_inference(kube)_v3.py"]
```

그림 13 Kafka consumer 도커 이미지 빌드를 위한 Dockerfile 스크립트 명세

o 위에서 명세한 Dockerfile을 활용하여 도커 키반의 컨테이너 이미지를 만들고 컨테이너 오케스트레이션을 위해 kubernetes에서 기능들을 생성하고 배포할 수 있도록 yaml 파일을 작성한다. 스마트 에너지 서비스에서는 관련이 있는 기능들로 묶어서 6개의 yaml 파일을 구분하여 작성하고 Smart_Energy.sh 라는 실행파일을 통해 모든 기능들을 순차적으로 생성하고 배포한다.

```
---
apiVersion: v1
kind: Service
metadata:
  name: edgex-mongo
  labels:
    app: edgex-mongo
spec:
  type: NodePort
  ports:
  - name: "edgex-mongo"
    port: 27017
    nodePort: 32000
  selector:
    app: edgex-mongo
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: edgex-mongo
spec:
  template:
    metadata:
      labels:
        app: edgex-mongo
    spec:
      containers:
      - name: edgex-mongo
        image: edgexfoundry/docker-edgex-mongo
        ports:
        - containerPort: 27017
      nodeSelector:
        functiontype: edgex
```

```
---
apiVersion: v1
kind: Service
metadata:
  name: edgex-core-consul
  labels:
    app: edgex-core-consul
spec:
  type: NodePort
  ports:
  - name: "8400"
    port: 8400
    targetPort: 8400
    nodePort: 32222
  - name: "8500"
    port: 8500
    targetPort: 8500
    nodePort: 32223
  - name: "8600"
    port: 8600
    targetPort: 8600
    nodePort: 32005
  selector:
    app: edgex-core-consul
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: edgex-core-consul
spec:
  template:
    metadata:
      labels:
        app: edgex-core-consul
    spec:
      containers:
      - name: edgex-core-consul
        image: edgexfoundry/docker-core-consul:latest
        ports:
        - containerPort: 8400
        - containerPort: 8500
        - containerPort: 8600
      nodeSelector:
        functiontype: edgex
```

```
---
apiVersion: v1
kind: Service
metadata:
  name: edgex-core-command
  labels:
    app: edgex-core-command
spec:
  type: NodePort
  ports:
  - name: "edgex-core-command"
    port: 48082
    nodePort: 32001
  selector:
    app: edgex-core-command
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: edgex-core-command
spec:
  template:
    metadata:
      labels:
        app: edgex-core-command
    spec:
      containers:
      - name: edgex-core-command
        image: edgexfoundry/docker-core-command:0.2.1
        ports:
        - containerPort: 48082
      nodeSelector:
        functiontype: edgex
```

```
---
apiVersion: v1
kind: Service
metadata:
  name: edgex-support-logging
  labels:
    app: edgex-support-logging
spec:
  type: NodePort
  ports:
  - name: "edgex-support-logging"
    port: 48061
    targetPort: 48061
    nodePort: 32003
  selector:
    app: edgex-support-logging
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: edgex-support-logging
spec:
  template:
    metadata:
      labels:
        app: edgex-support-logging
    spec:
      containers:
      - name: edgex-support-logging
        image: edgexfoundry/docker-support-logging:0.2.1
        ports:
        - containerPort: 48061
      nodeSelector:
        functiontype: edgex
```

```
---
apiVersion: v1
kind: Service
metadata:
  name: edgex-core-data
  labels:
    app: edgex-core-data
spec:
  type: NodePort
  ports:
  - name: "48080"
    port: 48080
    targetPort: 48080
    nodePort: 31091
  - name: "5563"
    port: 5563
    targetPort: 5563
    nodePort: 32009
  selector:
    app: edgex-core-data
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: edgex-core-data
spec:
  template:
    metadata:
      labels:
        app: edgex-core-data
    spec:
      containers:
      - name: edgex-core-data
        image: edgexfoundry/docker-core-data:0.2.1
        ports:
        - containerPort: 48080
        - containerPort: 5563
      nodeSelector:
        functiontype: edgex
```

```
---
apiVersion: v1
kind: Service
metadata:
  name: edgex-core-metadata
  labels:
    app: edgex-core-metadata
spec:
  type: NodePort
  ports:
  - name: "48081"
    port: 48081
    targetPort: 48081
    nodePort: 32007
  selector:
    app: edgex-core-metadata
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: edgex-core-metadata
spec:
  template:
    metadata:
      labels:
        app: edgex-core-metadata
    spec:
      containers:
      - name: edgex-core-metadata
        image: edgexfoundry/docker-core-metadata:0.2.1
        ports:
        - containerPort: 48081
      nodeSelector:
        functiontype: edgex
```

그림 14 EdgeX Foundry 프레임워크를 활용한 기능 생성 및 배포를 위한 EdgeX.yaml 파일 스크립트 명세

```
---
apiVersion: v1
kind: Service
metadata:
  name: api-server
  labels:
    app: api-server
spec:
  type: NodePort
  ports:
  - name: "api-server"
    port: 5000
    nodePort: 32014
  selector:
    app: api-server
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: api-server
spec:
  template:
    metadata:
      labels:
        app: api-server
    spec:
      containers:
      - name: api-server
        image: lshyeung/api-server
        ports:
        - containerPort: 5000
```

그림 15 API server 기능 생성 및 배포를 위한 api_server.yaml 파일 스크립트 명세

```
---
apiVersion: v1
kind: Service
metadata:
  name: kafka-broker-1
  labels:
    app: kafka-broker
spec:
  ports:
  - name: "kafka"
    port: 9092
  selector:
    app: kafka-broker
    brokerID: "1"
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: kafka-broker-1
spec:
  template:
    metadata:
      labels:
        app: kafka-broker
        brokerID: "1"
    spec:
      containers:
      - name: kafka-broker
        image: wurstmeister/kafka:latest
        ports:
        - containerPort: 9092
        env:
        - name: KAFKA_ADVERTISED_HOST_NAME
          value: "kafka-broker-1"
        - name: KAFKA_ZOOKEEPER_CONNECT
          value: "kafka-zookeeper:2181"
        - name: KAFKA_BROKER_ID
          value: "1"
        - name: KAFKA_CREATE_TOPICS
          value: "Device1:1:1"
```

그림 16 Kafka broker 기능 생성 및 배포를 위한 kafka.yaml 파일 스크립트 명세

```
---
apiVersion: v1
kind: Service
metadata:
  name: kafka-zookeeper
  labels:
    app: kafka-zookeeper
spec:
  ports:
  - name: "zookeeper"
    port: 2181
  selector:
    app: kafka-zookeeper
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: kafka-zookeeper
spec:
  template:
    metadata:
      labels:
        app: kafka-zookeeper
    spec:
      containers:
      - name: kafka-zookeeper
        image: wurstmeister/zookeeper:latest
        ports:
        - containerPort: 2181
```

그림 17 Kafka zookeeper 기능 생성 및 배포를 위한 zookeeper.yaml 파일 스크립트 명세

```
---
apiVersion: v1
kind: Service
metadata:
  name: consumer-d1-inference
  labels:
    app: consumer-d1-inference
spec:
  type: NodePort
  ports:
  - name: "consumer-d1-inference"
    port: 3334
    nodePort: 32112
  selector:
    app: consumer-d1-inference
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: consumer-d1-inference
spec:
  template:
    metadata:
      labels:
        app: consumer-d1-inference
    spec:
      containers:
      - name: consumer-d1-inference
        image: lshyeung/consumer_d1_inference
        ports:
        - containerPort: 3334
```

그림 18 consumer 기능 생성 및 배포를 위한 consumer_d1_inference.yaml 파일 스크립트 명세

```
---
apiVersion: v1
kind: Service
metadata:
  name: influxdb
  labels:
    app: influxdb
spec:
  type: NodePort
  ports:
  - name: "influxdb"
    port: 8086
    nodePort: 32015
  selector:
    app: influxdb
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: influxdb
spec:
  template:
    metadata:
      labels:
        app: influxdb
    spec:
      containers:
      - name: influxdb
        image: influxdb
        ports:
        - containerPort: 8086
```

```
---
apiVersion: v1
kind: Service
metadata:
  name: chronograf
  labels:
    app: chronograf
spec:
  type: NodePort
  ports:
  - name: "chronograf"
    port: 8888
    nodePort: 32016
  selector:
    app: chronograf
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: chronograf
spec:
  template:
    metadata:
      labels:
        app: chronograf
    spec:
      containers:
      - name: chronograf
        image: chronograf
        ports:
        - containerPort: 8888
```

그림 19 InfluxDB와 chronograf 기능 생성 및 배포를 위한 influx_chro.yaml 파일 스크립트 명세

```
#!/bin/bash



kubectl apply -f zookeeper.yaml
kubectl apply -f kafka.yaml
kubectl apply -f influx_chro.yaml
kubectl apply -f api_server.yaml
kubectl apply -f consumer_device1.yaml
kubectl apply -f EdgeX2.yaml
```

그림 20 스마트 에너지 서비스의 기능 생성 및 배포 실행을 위한 Smart_Energy.sh 파일 스크립트 명세

4. 컨테이너 오케스트레이션을 통한 스마트 에너지 서비스 기능 배포 및 합성 실증

4.1. 서비스 합성 개요

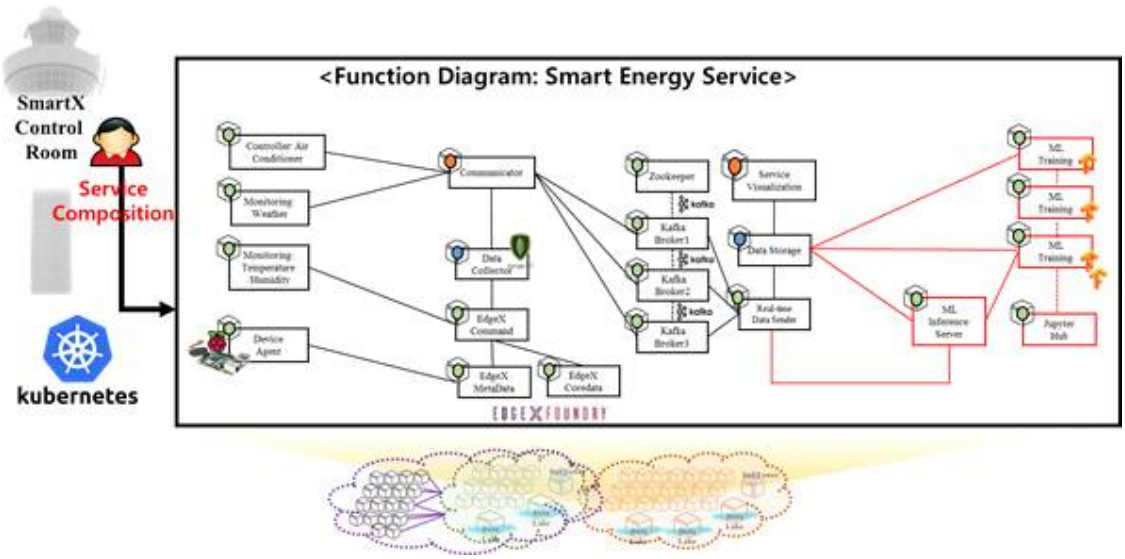

그림 21 스마트 에너지 서비스 Function Diagram

o 오버클라우드 상에서 IoT-Cloud SaaS 응용 호환성을 실증하기 위한 스마트 에너지 서비스는 오버클라우드를 활용하여 생성되는 Kubernetes Cluster를 바탕으로 구성된 환경에 기반하여 합성된다. 그림 21과 같이 생성되고 연결되는 모든 기능들을 Kubernetes 컨테이너 오케스트레이션을 통해 기능들을 적합한 곳에 배포하여 서비스를 구성하고 실증한다.

o 오버클라우드의 구성이 완료된 후, 오버클라우드 상에 생성된 Kubernetes Cluster를 확인하고 이것들을 바탕으로 스마트 에너지 서비스에 대한 합성을 수행한다. 작성해둔 Smart_Energy.sh 실행파일을 실행하면 스마트 에너지 서비스의 모든 기능이 미리 명세한 yaml파일을 기반으로 하여 생성되어 배포되고 이것들을 지속적으로 확인할 수 있다. 또한 서비스에서 제공하는 가시화(visualization) 도구인 chronograf를 활용하여 서비스 데이터를 확인하여 서버실 내부 및 외부 날씨 정보에 대한 상황을 파악할 수 있다.

4.2. 서비스 합성 실증

o 스마트 에너지 서비스 수행을 위한 모든 소스 코드 및 파일은 http://github.com/SmartX-Team/Smart-Energy-Service에서 확인 가능하다. 스마트 에너지 서비스 합성의 시작은 정의된 서비스 실행 파일을 바탕으로 하며, 이를 통해 사전에 작성된 명세를 바탕으로 배포된 각 요소 기능들에 대한 설정을 적용하여 서비스를 수행하고 그 결과를 확인한다. 아래 그림 22과 같이 스마트 에너지 서비스의 요소 기능들의 생성 및 배포되는 절차를 확인하고, 그림 23과 같이 동작 상태에 대해서 모니터링할 수 있다.

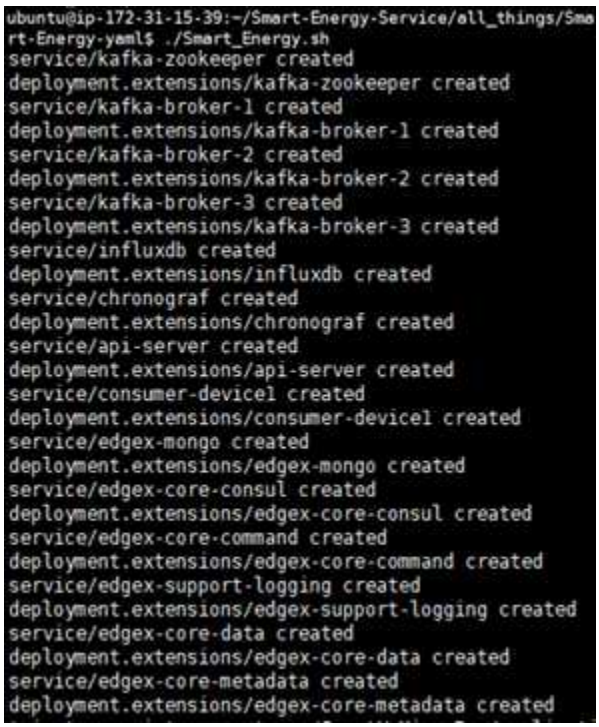

그림 22 Kubernetes 기반의 스마트 에너지 서비스 요소 기능 생성 및 배포

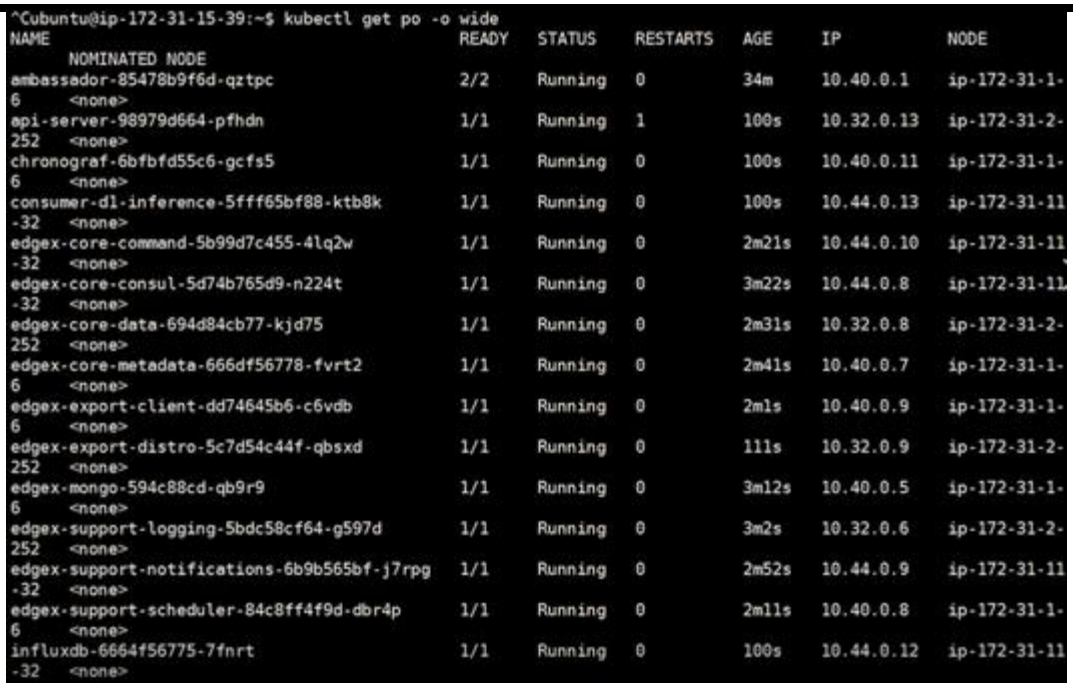

그림 23 Kubernetes 기반의 스마트 에너지 서비스 요소 기능 동작 상태

o 그림 24는 스마트 에너지 서비스의 IoT 디바이스 측의 수집 및 전달 기능의 실행 결과이다. 절차는 오버클라우드 상의 IoT-Cloud Hub의 EdgeX 서비스에 어떤 데이터 형식을 등록할 것인지 정의한 후, 디바이스의 프로필을 등록한다. 그리고 디바이스 서비스를 등록하여 연계한 후에 데이터를 10초 간격으로 측정하여 전달한다.

그림 24 스마트 에너지 서비스 IoT 디바이스의 기능 실행 결과

o 스마트 에너지 서비스를 통한 서비스 데이터 축적 결과는 오버클라우드 상의 최종 데이터 저장소인 InfluxDB에서 확인할 수 있다. 그림 25는 자체적으로 확인한 InfluxDB 상의 데이터이다. 데이터 형식은 현재 에어컨 온도, 외부 습도, 외부 온도, 실내 습도, 실내 온도, 날씨 상태, 추천 온도 등과 같은 데이터 메트릭(Data Metric)으로 저장된다.

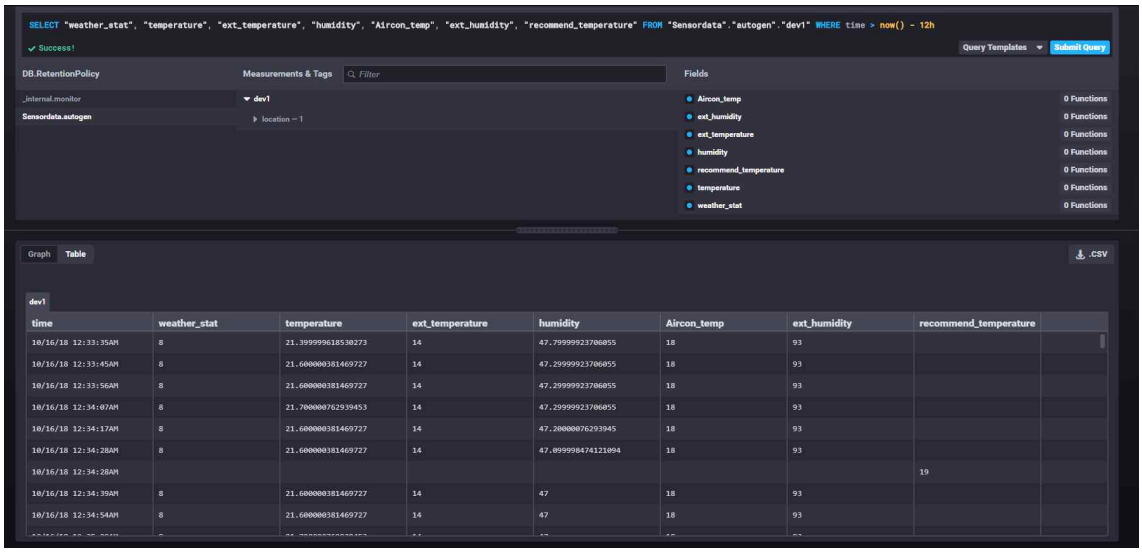

그림 25 스마트 에너지 서비스를 통한 서비스 데이터 축적 결과

o 그림 26은 스마트 에너지 서비스의 가시화 결과이다. 사용자 인터페이스에서 실내 온도와 실외 온도를 하나의 그래프로 좌측에 나타내고 우측에는 실내 습도와 실외 습도의 그래프를 배치하여 한눈에 실시간 온습도 모니터링에 용이하게 하였다. 또한 상세 데이터 축적 결과를 볼 수 있는 데이터베이스 정보를 함께 나타내주고, 가장 중요한 정보인 현재 에어컨 설정 온도, 날씨 정보, 추천 설정 온도 값을 아래에 색깔 별로 구분하여 확인하도록 하였다.

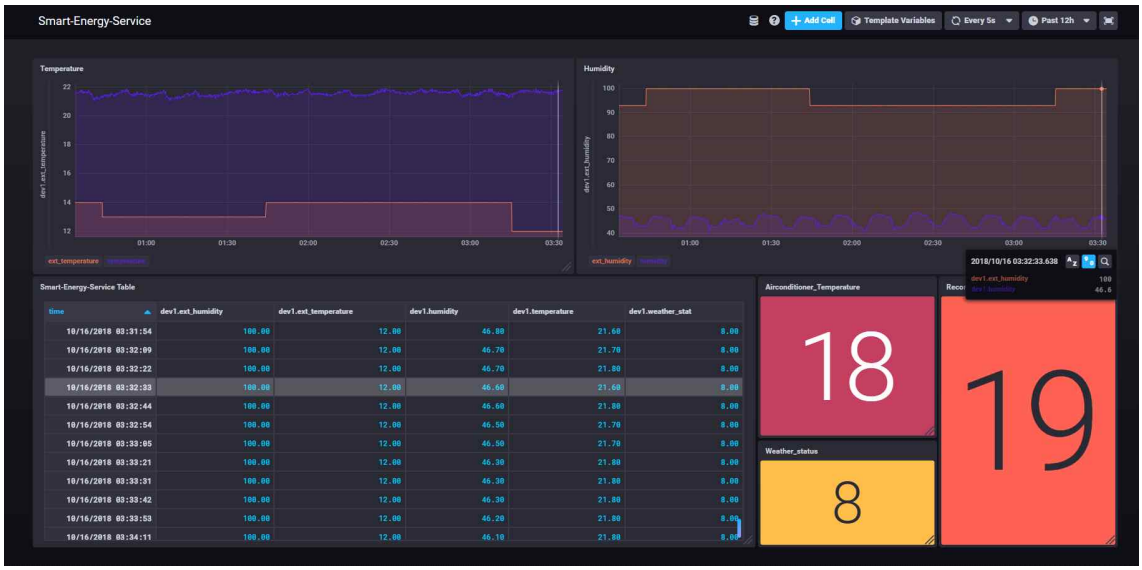

그림 26 스마트 에너지 서비스 가시화 결과

o 향후 연구에서는 서비스에서 분리된 기능들을 각각의 필요에 맞는 환경을 파악하여 기능들을 리소스 별로 알맞게 할당하여 자동으로 어떤 환경에서도 서비스가 운영될 수 있도록 하는 리소스 인지 서비스에 대한 연구를 진행하여 서비스를 보완할 계획이다. 이를 통해 통상적인 컨테이너 오케스트레이션이 제공하는 컨테이너 배포 및 운영보다 효율적인 서비스로 개선하고자 한다.

References

[1] EdgeX Foundry, https://www.edgexfoundry.org/

[2] Docker, https://www.docker.com/

[3] Smart Energy IoT-Cloud Service, http://github.com/SmartX-Team/Smart-Energy-Service

[4] S. Nastic, S. Sehic, D. Le, H. Truong, and S. Dustdar, "Provisioning software-defined iot cloud systems," in Proc. International Conference on Future Internet of Things and Cloud (FiCloud 2014), 2014, pp. 288-295.

[5] F. Khodadadi, R. Calheiros, and R. Buyya, "A data-centric framework for development and deployment of internet of things applications in clouds," in Proc. IEEE 10th International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP 2015), 2015, pp. 1-6.

[6] J. Shuja, K. Bilal, S. Madani, M. Othman, R. Ranjan, P. Balaji, and S. Khan, "Survey of techniques and architectures for designing energy-efficient data centers," 2014.

[7] Z. Sheng, H. Wang, C. Yin, X. Hu, S. Yang, and V. Leung, "Lightweight management of resource-constrained sensor devices in internet of things," IEEE Internet of Things Journal, vol. 2, no. 5, pp. 402-411, 2015.

SaaS OverCloud 기술 문서

작성기관: 광주과학기술원
작성년월: 2019/1