Ghent University

Cloud Applications and Mobile (E736010)

Lab 3: Microservices

Professor:
Prof. dr. ir. Sofie Van Hoecke

Assistants:
Ing. Stef Pletinck
Dr. Jennifer B. Sartor

2024–2025

Table of contents

1 Introduction

In order to provide a good overview of the bigger picture, we first show the complete architecture (Figure 1). During lab sessions 1 and 2, the source code was running almost entirely on the embedded devices (RPi and ESP32). From the current lab session on (with the exception of Lab 4), a lot of overhead will be taken away from these constrained devices by removing their need to compute and visualize the data they are capturing or have collected. We will move to microservices running in the cloud.

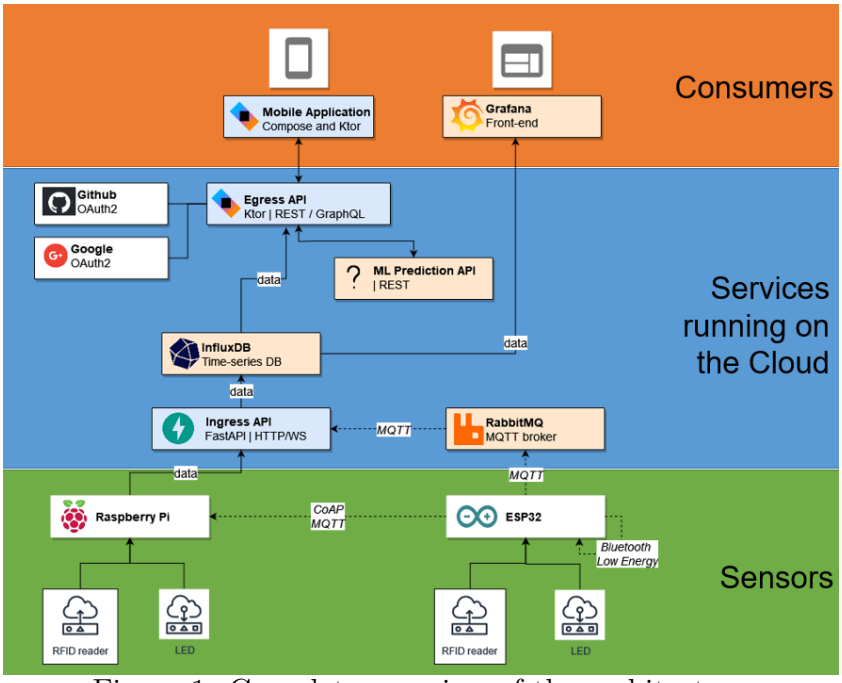

Figure 1: Complete overview of the architecture

The components that will be used in the current lab session become clear by zooming in on the architectural overview. Various technologies and databases will be used over the course of this lab session as illustrated in Figure 2. Notice that as we move up from the Raspberry Pi and ESP32, the larger boxes that are outlined represent microservices that we will create and launch in the cloud in this lab session. The links between different components and their purpose within the complete architecture will become clear as we deploy each one of them step by step. You will code up the Ingress API box for this lab, and the orange-colored boxes are shared services that run on the cloud and have been set up for you. For example, the InfluxDB box is a database that stores our sensor data, and the Grafana box is a visualization tool that can visualize our sensor data.

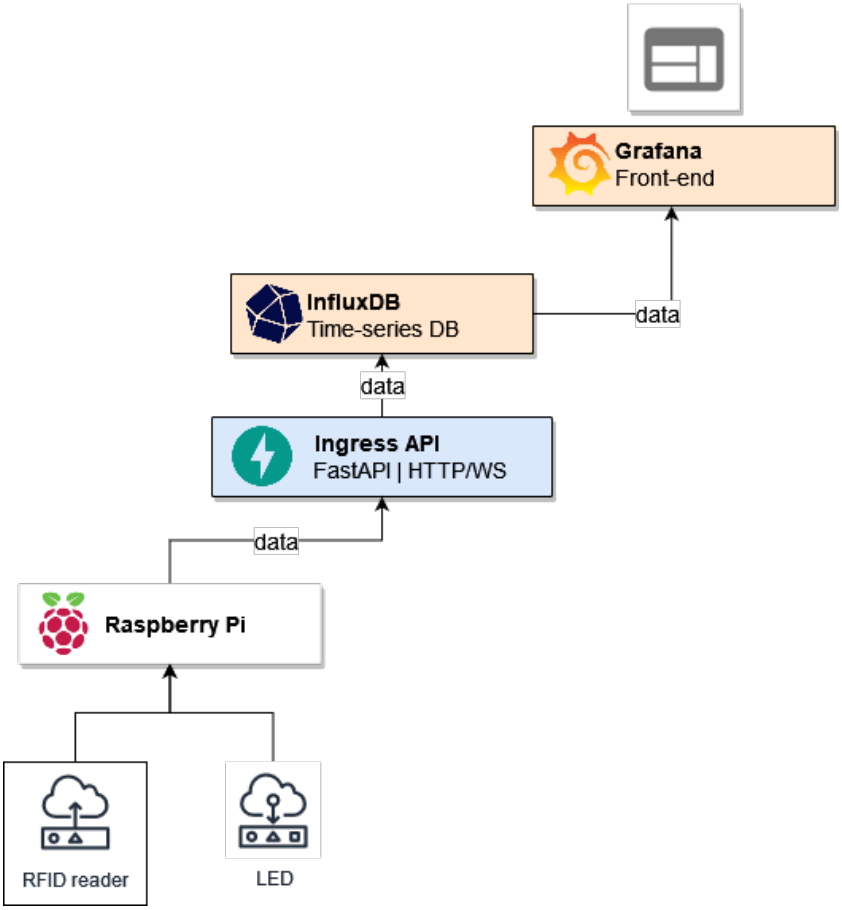

Figure 2: Architecture for the current lab session

2 Goals

1. Implement a REST API using FastAPI and InfluxDB
2. Hands-on experience with the microservice architecture concept using Docker and Kubernetes in the cloud
3. Persistence and visualization of time-series (sensor) data with InfluxDB and Grafana

3 Technology overview

3.1 InfluxDB

InfluxDB is an open-source time series database developed by the company InfluxData. It is used for storage and retrieval of time series data in fields such as operations monitoring, application metrics, Internet of Things sensor data, and real-time analytics.

InfluxDB provides an SQL-like language with built-in time-centric functions for querying a data structure composed of measurements, series, and points. Each point consists of several key-value pairs called the fieldset and a timestamp. When grouped together by a set of key-value pairs called the tagset, these define a series. Points are indexed by their time and tagset.

Finally, series are grouped together by a string identifier to form a measurement.

3.2 Grafana

Grafana is a leading open-source platform for analytics that allows one to query metrics from various data sources and offers various types of graphs for visualization. A data source is any backend storage for time-series data. The following data sources are officially supported: Graphite, **InfluxDB**, OpenTSDB, Prometheus, Elasticsearch and CloudWatch. Using Grafana, data can be brought together in a dashboard and shared within and across teams. It is also very extendable and offers hundreds of dashboards and plugins created by the community.

4 Preparation

4.1 Installation

4.1.1 Docker Desktop

The concept of microservices and the microservice architecture is highly influenced by the popularity of containerization technologies like Docker. To quickly deploy services regardless of the operating system that has been installed on the machine, the services are deployed and run within a Docker container. The installation guide provides all the instructions required to install Docker on your machine. More information about Docker and its workflow can be found on the official documentation pages.

Note: For Windows users, Docker Desktop is only available for Windows Pro, Enterprise and Education editions. For any other edition, use Docker Desktop for Windows.

4.1.2 Kubernetes and Helm

The microservice cloud environment used throughout the following lab sessions is a Kubernetes cluster. To interact with this environment from your local machine, you are required to download and install the Kubernetes Command Line Tool (kubectl). The installation guide can be found here.

To ease the deployment of our applications onto this Kubernetes cluster, we will use Helm. This is a package manager for Kubernetes which allows us to easily deploy and maintain our microservices. Install Helm v3. The installation guide can be found here.

Note: For Windows users, download the Helm executable from Github.

Windows users: Once you've downloaded **kubectl.exe** and **helm.exe**, place these executables in a folder of your choice and append this folder to your path environment variable. You can do this from the command line using the commands below or follow this tutorial.

```
# Using Command Prompt
```

```
set PATH=%PATH%;C:\your\path\here\
# Using Powershell
$env:Path+=";C:\your\path\here\"
```

Finally, test the successful installation by executing the following commands:

```
kubectl --help
helm --help
```

4.2 Accessing the Kubernetes cluster

Before we're able to deploy the microservices created throughout the next lab sessions in the cloud environment, we should first configure our access to it. You should have received a configuration file similar to the one shown in Listing 1. It describes the Kubernetes cluster used for these lab sessions and the service account created for you. Your service account is restricted to your own personal *namespace*, e.g., "cl-moens-pieter". As this is already defined within the configuration file, you won't have to worry about namespaces throughout the lab sessions.

$HOME/.kube/config

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://cloudandmobile.ilabt.imec.be:6443
  name: cloudandmobile
contexts:
- context:
    cluster: cloudandmobile
    user: pieter
    namespace: cl-moens-pieter
  name: pieter@cloudandmobile
current-context: pieter@cloudandmobile
kind: Config
preferences: {}
users:
- name: pieter
  user:
    token: DATA+OMITTED
```

Listing 1: Basic microservice - Kubeconfig

Rename this file **config** and copy it to a hidden directory in your home folder that you have to create (unless you've already used Kubernetes before): **$HOME/.kube/**. After that, it should be automatically picked up by the Kubernetes command line tool, *kubectl*. If this is not the case, you can specify the configuration file as an environment variable:

```
# Linux
export KUBECONFIG=$HOME/.kube/config
# Windows
set KUBECONFIG=%HOMEDRIVE%%HOMEPATH%\.kube\config
```

You should now already be able to connect to the Kubernetes cluster using the following command:

```
kubectl get pod
```

which should return "No resources found." as you currently have no pods running within your namespace.

4.3 Accessing Grafana dashboard

Within the cluster, there is a Grafana service already deployed which will be shared across students. Each student has their own environment in which dashboards can be created freely. Before continuing, double check your access to this dashboard by browsing to http://grafana.monitoring.cloudandmobile.ilabt.imec.be and entering the credentials provided to you.

5 Tutorials

5.1 Interacting with the cloud environment

This tutorial includes an introduction to Docker, Kubernetes and Helm. To show the benefits of each of these technologies and how they interact with one another, a simple example will be deployed within the cloud environment.

5.1.1 Creating a basic microservice

A microservice is no more than an application that can be accessed by other services. To achieve this, we can deploy our FastAPI application from Lab2 to the cloud, which will expose itself on port 8000. Instead of running that code locally, we will run it on a server in our Kubernetes cluster. To do that, you will need the zip file from Ufora called `filesForStudents-Lab3-microserviceFromLab2.zip` which contains a Dockerfile and chart folder (needed for helm) that you have to move to the main folder of your Lab2 code.

The cloud will use the **requirements.txt** file to install the correct libraries and dependencies for our project.

To deploy this application on a Kubernetes cluster, there are a few more steps required.

5.1.2 Virtualization using Docker

The first step is to package the application as a Docker container image. The image created in this tutorial is based on python 3.11. In Listing 2, you can see the Dockerfile that was given to

you that uses python 3.11. The additional lines of the Dockerfile are quite straightforward as they (i) install the requirements specified in the **requirements.txt** file, (ii) copy the required files into the container, and (iii) finally run the python application.

Dockerfile

```
FROM python:3.11
WORKDIR /code
COPY ./requirements.txt /code/requirements.txt
RUN pip install --no-cache-dir --upgrade -r /code/requirements.txt
COPY ./*.py /code/
ADD static /code/static
ADD templates /code/templates
ENTRYPOINT ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "80",
"--proxy-headers", "--workers", "4", "--forwarded-allow-ips", "*"]
CMD []
```

Listing 2: Basic FastAPI microservice - Dockerfile

Now, we have to host the code for this container image in a so-called container repository, so our cloud can find it and deploy it. Docker Hub is the official container repository hosted by Docker. Go ahead and create a free user account.

Using the **Dockerfile** in Listing 2 and a couple of simple Docker commands, we can build and push our container image to this container repository. Run these commands within the directory of the Dockerfile:

```
docker login

docker build --tag <docker-username>/helloworld:latest .
docker push <docker-username>/helloworld:latest
```

Note that the version tag "latest" can be changed to any specific version, e.g., "0.0.1".

If these commands have been executed successfully, you should now find the container image publicly accessible through your Docker Hub account.

If you change your code and want to deploy it again, do not forget to rerun the build and push commands above.

5.1.3 Deploying resources using Helm

Out-of-the-box, each resource created in Kubernetes is defined by a YAML file. This is the case for all resources types, e.g., Deployments, Services, Ingress, Secrets. To prevent all users from having to create these YAML descriptions, Helm has introduced *Charts*. In essence, these are templates containing the required YAML descriptions to deploy a given application. You can generate a basic example using the following command:

```
helm create example
```

```
example
├── templates
├── .helmignore
├── Chart.yaml
└── values.yaml
```

This creates a chart directory named "example" with the following structure:

The **templates** folder contains these previously mentioned YAML files for the required resources. Within these templates, variables can be used to customize the YAML files before deployment. These variables and their values are defined in the **values.yaml** file. This way, you don't have to write YAML that's common across many services every single time, instead just filling in the variables in templates.

The Helm chart allows you to create a Deployment, Service, Ingress and a PersistentVolume resource. You should find a chart directory included in the zip file you downloaded from Ufora, and copy it to the directory where your Lab2 code lives. An example `values.yaml` file is shown in Listing 3.

The first important block of the `values.yaml` file is the specification of the container image. This is the same image created in Section 5.1.2, i.e., your-docker-username/hello-world:latest. Secondly, a Service resource is created. We specify the containerPort variable which specifies the port on which our application is exposed. Finally, the Ingress resource is created to ensure availability through our domain name. This is not to be confused with the ingress API that we build in this lab. Ingress refers here to making sure our services are visible and accessible outside the cluster.

!Important: The convention throughout the lab session for the Ingress hostname will be:

```
<your-service>.<your-ugent-username>.cloudandmobile.ilabt.imec.be
```

You can now proceed to deploy your Lab2 application code on the Kubernetes cloud by executing the following command in the directory where the **chart** folder is located (normally within the folder where `main.py` is):

```
helm install "helloworld" chart
```

The Chart includes a description of all the Kubernetes resources required when deploying our application. Using the above command, Helm injects the values for the variables described in the **values.yaml** file into the chart's templates and deploys them on the Kubernetes cluster. Note that you can specify any **values.yaml** file you might have if it includes the required variables.

After a few seconds, your application should now be accessible through the specified hostname, e.g.,

```
helloworld.pimoens.cloudandmobile.ilabt.imec.be
```

values.yaml

```yaml
replicaCount: 1

image:
  repository: stefpletinck/helloworld
  pullPolicy: Always
  # Overrides the image tag whose default is the chart appVersion.
  tag: "latest"

imagePullSecrets: []
nameOverride: ""
fullnameOverride: ""

serviceAccount:
  create: false
  annotations: {}
  name: ""

podAnnotations: {}

podSecurityContext: {}
  # fsGroup: 2000

securityContext: {}

service:
  type: ClusterIP
  port: 80

ingress:
  enabled: true
  className: ""
  annotations: {}
  hosts:
    - host: hello-world.stpletin.cloudandmobile.ilabt.imec.be
      paths:
        - path: /
          pathType: ImplementationSpecific
  tls: []

resources: {}

autoscaling:
  enabled: false
  minReplicas: 1
  maxReplicas: 100
  targetCPUUtilizationPercentage: 80
  # targetMemoryUtilizationPercentage: 80

nodeSelector: {}

tolerations: []

affinity: {}
```

Listing 3: Basic FastAPI microservice - values.yaml

However, something went wrong, and we will help you debug it in person in the lab session. If

you are working at home, you will have to debug it yourself. Depending on the problem, follow the steps below to fix it and re-deploy the code until it is working.

Hint: Look carefully at the `values.yaml` file.

For an overview of how Docker, Kubernetes, and Helm work together, look at Figure 3.

5.1.4 Debugging your code

If your code is deployed to the cloud but is having some error, you could debug the problem using this sequence of commands:

```
$ kubectl get pod
NAME                                READY   STATUS    RESTARTS   AGE
hello-standard-66c8cdccdf-xbp8w     1/1     Running   0          7d2h

$ kubectl logs -f helloworld-standard-66c8cdccdf-xbp8w
```

The `kubectl logs` command will display the output of your code or the stacktrace if an error has occurred.

Re-deploying your code after changes

We have configured the Deployment to pull the latest version of your Docker image every time the pod starts. You also know that a Deployment will always re-create a pod if it notices it is missing. **Thus, you can update your application by deleting the pod and letting Kubernetes re-create it from the code in your DockerHub repository.** So, first make sure your code is up-to-date in the DockerHub repository before you let Docker re-create your pod.

This is what that might look like:

```
$ kubectl get pod
NAME                                READY   STATUS    RESTARTS   AGE
hello-standard-66c8cdccdf-xbp8w     1/1     Running   0          7d2h

$ kubectl delete pod helloworld-standard-66c8cdccdf-xbp8w
```

Another option is to ask Kubernetes to restart all pods in your Deployment one by one. If you have several pods, it'll wait until the first one has become ready again before continuing to restart the next pod; this way you can run updates without having your application go offline. Here is what that might look like:

```
$ kubectl get deployment
NAME                    READY   UP-TO-DATE   AVAILABLE   AGE
```

```
helloworld-standard     1/1     1            1           2h7m

$ kubectl rollout restart deployment helloworld-standard
```

Re-deploying your code after yaml changes

If, for whatever reason, **the name of your image or any of the values in `values.yaml` changes**, after making those changes, you can run an upgrade command:

```
helm upgrade "helloworld" chart
```

Useful commands for inspecting and debugging your container

You can explore your deployed microservice using the following commands:

```
# helm
helm ls --all

# kubectl
kubectl get pod
kubectl get deployment
kubectl get service
kubectl get ingress

# debugging
kubectl describe pod <pod-name>
kubectl logs -f <pod-name>

# to uninstall the microservice
helm uninstall <release-name>
```

5.2 REST API powered by FastAPI

During this lab session, a web service, named the Ingress API, will be created and deployed that will communicate with other services within and outside of the Kubernetes cluster. The most elegant way of realizing this communication is by having each service expose an Application Programmable Interface (API). The most popular and widely used technology for APIs is REST. It is a stateless, robust protocol that utilizes HTTP and therefore supports all the HTTP request methods (GET, POST, PUT, DELETE, OPTIONS..).

This tutorial will cover how to easily and quickly create a RESTful API using FastAPI. In this section, we will go over the code that we gave you as starter code to implement ingress in the cloud. You will have to extend this code to complete the tasks below.

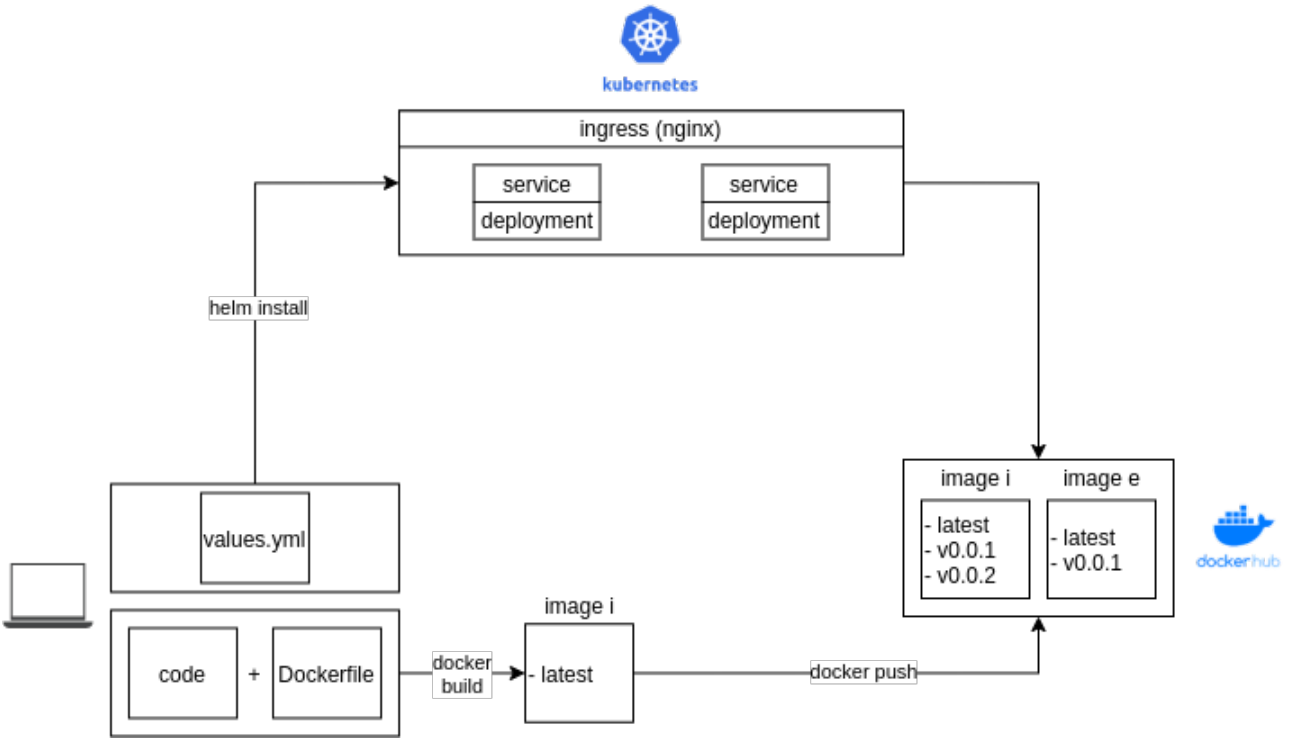

Figure 3: Overview of interaction with different cloud components

5.2.1 Application structure

The FastAPI application created during this lab session will become larger over the span of the semester. In order to maintain scalability, a little change in structure compared to the previous lab session is introduced as shown in Figure 4.

```
ingress-api
├── config.py
├── models.py
├── main.py
├── helm
├── Dockerfile
├── compose.yml
└── requirements.py
```

Figure 4: Application structure

You can see the configuration details in `config.py`, listed in Listing 4, where we define a class `Settings` that inherits from Pydantic's `BaseSettings`. Configuration of a FastAPI app can be implemented in several ways; here we opt to store the configuration constants in a class that inherits from BaseSettings, because then the automatic validation system of FastAPI will replace the defaults with environment variables if they exist.

We have a model of a sensor data point defined in `models.py`. This model is the data structure for the RPi to use as it collects sensor data. The class `RfidDataPoint` inherits from a base class of Pydantic and has fields `timestamp`, `rfid_id`, `count`, and `sensor_name`. Each time a new RFID tag is scanned, the RPi will update an internal data structure of the active RFID tags, which it will use to store a `count` of the number of active RFID tags, or active people. The first time an RFID tag is scanned, this represents a person entering and the count is increased. If that RFID tag is scanned again, the person is leaving and the count is

decreased. The `sensor_name` is the name of who sent the sensor data to the database, an identifier, such as `rpi_sensors_rfid`. **Make sure the time on your RPi is correctly set**, as this is required to generate an accurate timestamp for InfluxDB. Every time a new RFID tag is scanned by your RFID reader, you will send both the scanned `rfid_id` and the updated `count` to the InfluxDB running in the cloud, with an accurate `timestamp` and `sensor_name`.

ingress-api/config.py

```
from pydantic_settings import BaseSettings

class Settings(BaseSettings):
    bucket: str = 'replace-me'
    influx_org: str = 'cloud2023'
    log_level: str = 'INFO'
    influx_token: str
    influx_url: str
```

ingress-api/models.py

```
from pydantic import BaseModel

class RfidDataPoint(BaseModel):
    timestamp: int
    rfid_id: str
    count: int
    sensor_name: str
```

ingress-api/main.py

```
import logging
from fastapi import FastAPI, HTTPException
from influxdb_client import InfluxDBClient
from influxdb_client.client.write_api import SYNCHRONOUS

from config import Settings
from models import RfidDataPoint

app = FastAPI()
settings = Settings()
influx = InfluxDBClient(url=settings.influx_url, token=settings.influx_token,
org=settings.influx_org)

@app.get("/", description="Example route, also used for healthchecks")
async def root():
    return {"message": "Hello World"}
```

Listing 4: Showing files `config.py`, `models.py`, and `main.py`

Finally, we have our `main.py` file. We import the necessary libraries from FastAPI and InfluxDB. We then import our configuration class and our model. Then we instantiate our app and our `Settings` class, and create an `InfluxDBClient`, using the details specified in `config.py`. We then define one endpoint for "/" that just returns the message "Hello World".

When you run your `ingress-api` code, don't forget to specify your InfluxDB credentials

from your credentials file in the `config.py` file **if you run your code locally**. If you run your code in the cloud, you specify your InfluxDB credentials in your `values.yaml` file.

5.3 Ingress API - InfluxDB

This third and final tutorial will explain how to combine the concepts covered in the previous tutorials in order to create and link together the Ingress API with the InfluxDB service which is running in the cloud environment, as shown in Figure 2.

Ingress refers to unsolicited traffic sent from an address in the public internet to a private network. In this case, it's our sensor data being sent to our API.

5.3.1 InfluxDB

InfluxDB 2.7 is a platform purpose-built to collect, store, process and visualize time series data. Time series data is a sequence of data points indexed in time order. Data points typically consist of successive measurements made from the same source and are used to track changes over time. InfluxDB provides a useful getting started tutorial with information on the query language and how to write data to the database.

The InfluxDB data model organizes time series data into buckets and measurements. A bucket can contain multiple measurements. Measurements contain multiple tags and fields.

A bucket is a named location where time series data is stored. A bucket can contain multiple measurements. A measurement is a logical grouping for time series data. All points in a given measurement should have the same tags. A measurement contains multiple tags and fields. Tags are key-value pairs with values that differ, but do not change often. Tags are meant for storing metadata for each point; for example, something to identify the source of the data like host, location, station, etc. Fields are key-value pairs with values that change over time–for example: temperature, pressure, stock price, etc. Timestamp is of course the timestamp associated with the data. When stored on disk and queried, all data is ordered by time.

For example, consider a use case where you collect data from sensors in your home. Each sensor collects temperature and humidity. In this case, we would have a measurement be set to "home", and a tag with a key set to "room", which would specify the room where the sensor collected the data in the value, such as "Kitchen". Then the fields' keys would be "temp" and "hum", for example, and the value corresponding to the keys would hold the sensor data.

In InfluxDB, a point is a single data record identified by its measurement, tag keys, tag values, field key, and timestamp. A series is a group of points with the same measurement, tag keys, and tag values.

!Important: InfluxDB supports up to nanosecond precision. If the precision of the timestamp is not in nanoseconds, you must specify the precision when writing the data to InfluxDB. By default our ingress will keep track of timestamps in milliseconds, so this precision must be adjusted when writing to InfluxDB.

To also help with making other endpoints, we have extended `models.py` with a few more models, see Listing 5. The models are similar to our `RfidDataPoint` model defined above,

but these models are used for receiving data from the database after executing a query. There is one class to represent a `TagEntry`, or a scanned RFID tag, which we can get back after querying the database. One class represents a `CountEntry` for when we query the database for its count value(s). Finally, if we want to acquire all recent data stored in the database, there is a class called `RecentData` that combines the previous two classes.

main.py

```
class TagEntry(BaseModel):
    timestamp: int
    rfid_id: str
    sensor_name: str

class CountEntry(BaseModel):
    timestamp: int
    count: int
    sensor_name: str

class RecentData(BaseModel):
    rfid_ids: List[TagEntry]
    counts: List[CountEntry]

    def merge(self, other: Self) -> Self:
        self.rfid_ids.extend(other.rfid_ids)
        self.counts.extend(other.counts)

        return self
```

Listing 5: Models of data stored in InfluxDB

Here you can find a good tutorial on how to program in Python with InfluxDB, which explains both how to write data to InfluxDB and how to query the data after it is in the database, both of which you will need to complete Task 1. Here we provide a small overview of `Flux`, which is InfluxDB's query language.

When querying InfluxDB with Flux, there are three primary functions you use:

- `from()`: Queries data from an InfluxDB bucket.
- `range()`: Filters data based on time bounds. Flux requires "bounded" queries, or queries limited to a specific time range.
- `filter()`: Filters data based on column values. Each row is represented by r and each column is represented by a property of r. You can apply multiple subsequent filters.

Flux uses the pipe-forward operator (`|>`) to pipe the output of one function as the input of the next function. The following example Flux query returns the 'hum' and 'temp' fields stored in the 'home' measurement with timestamps between 2022-01-01T08:00:00Z and 2022-01-01T20:00:01Z.

As the InfluxDB service is already running in the cloud, you have to deploy your Ingress API in the cloud using Helm. The complete **values.yaml** file for our Ingress API is provided in the source code and the most important snippets are highlighted in Listing 7. The image, service and ingress sections are similar to the ones previously covered in 5.1.3. In addition to

```
from(bucket: "get-started")
    |> range(start: 2022-01-01T08:00:00Z, stop: 2022-01-01T20:00:01Z)
    |> filter(fn: (r) => r._measurement == "home")
    |> filter(fn: (r) => r._field == "hum" or r._field == "temp")
```

Listing 6: Example Flux query for InfluxDB

these, a small section is added to set environment variables to specify the correct settings or configuration for talking with InfluxDB.

Before deploying this, you should create a new Influx token using the web interface. The one included in your credentials file is read-only, and we want our ingress to write data. To do this, go to http://influx.database.cloudandmobile.ilabt.imec.be/signin and follow the instructions as explained in the Influx documentation. A custom access token that has read and write permissions to your bucket should be enough.

The helm installation is similar, but this time the chart is contained in a directory called helm, so the command would become `helm install ingress-api helm`

ingress-api/helm/values.yaml

```
replicaCount: 1

image:
  repository: stefpletinck/ingress-api
  pullPolicy: Always
  # Overrides the image tag whose default is the chart appVersion.
  tag: "latest"

env:
  - name: 'BUCKET'
    value: 'stpletin'
  - name: 'INFLUX_ORG'
    value: 'cloud2324'
  - name: 'INFLUX_TOKEN'
    value: 'abc'
  - name: 'INFLUX_URL'
    value: 'http://example.org'

service:
  type: ClusterIP
  port: 80

ingress:
  enabled: true
  className: "nginx"
  annotations: {}
  hosts:
    - host: ingress-api.stpletin.cloudandmobile.ilabt.imec.be
      paths:
        - path: /
          pathType: Prefix
  tls: []
```

Listing 7: Ingress API - Helm values

When you've pushed the ingress code to the cloud, you can check out the Swagger documenta-

tion of the API located at

`http://ingress-api.<username>.cloudandmobile.ilabt.imec.be/docs`

After you have completed Task1 and written some data to the database, it is possible to then explore this data using the InfluxDB user interface.

The sensor data, e.g. `count`, can be queried and visualized, such as outlined in this tutorial. You can either select a bucket, a time range, a measurement, and particular fields or tags through InfluxDB's interface, or write a simple Flux query. You will then see a visualization as shown in Figure 5.

Figure 5: InfluxDB user interface

If something goes wrong and you send incorrect data to the InfluxDB, particularly if you sent data with the wrong type, the only way to fix this is to delete the entire bucket. The instructions on how to do that are to be found on this InfluxDB webpage on how to delete a bucket. Since you have to delete the whole bucket, we only recommend doing this in the initial stages when testing and not later when you have useful data in the database.

6 Tasks

6.1 Task 1: Extending the ingress API

Extend the ingress API by creating a few new endpoints. You will both write to the database, and query the database for what has already been stored. The following endpoints should be created **and implemented**:

- **/data/** - POST - Send a new sensor measurement to the database, including `sensor_name`, timestamp, `rfid_id`, and `count`, which is the RPi's count of active RFIDs.

- **/data/** - GET - List all the data stored in the database within a given timespan, e.g. the last hour.
- **/count/** - GET - Query the database for its latest `count` of active users.

For the POST of new data, you should have two different measurements, one for sending the `count`, for which the measurement should be called `people`. The second measurement should be called `raw_ids`, which sends the latest scanned `rfid_id`. The key for the `tags` field should be `source`, and the value will contain the `sensor_name`. The key for the `fields` field should be `value`, and will point to the raw data that is being sent, either the `count` or `rfid_id`. For the details of what to send, see Section 5.2.1 above which explains the details of the `RfidDataPoint`.

The endpoint that gets all data should query the database for both `rfid_ids` and `counts` within the last given amount of time, e.g. the last hour, which will be specified in a parameter. The method should return an instance of a class defined in `models.py` which specifies the format of `RecentData`.

Finally, the endpoint that gets the most recent count should return just a number, the latest `count` stored in the database. This could be used to reset the RPi's `count` variable after it is restarted, for example.

After implementing these new endpoints, your Ingress API should look like that in Figure 6, although the `GET /current/` endpoint is optional and is explained below in Task4. Make sure you return the data in the proper format.

Figure 6: Task 1 - Result

6.2 Task 2: Collecting sensor data

Create a Python application to read scans of RFID tags (see previous lab sessions) and post the data to your ingress API running in the cloud. Each time a new RFID tag is scanned, the

RPi will have to update its count of active RFID tags and send both the current `count` of people, as well as the new `rfid_id` to ingress, in the right format. Deploy the project on the Raspberry Pi.

Send the data formatted according to the documented model:

```
class RfidDataPoint(BaseModel):
    timestamp: int
    rfid_id: str
    count: int
    sensor_name: str
```

Hint: There is no need to create a FastAPI application. A simple Python project will do.

In addition, you can revise this Python project to generate random sensor data for when you don't want to connect the RFID reader to the RPi. This script may be run on your laptop in future lab sessions.

6.3 Task 3: Monitoring sensor data

Load http://grafana.monitoring.cloudandmobile.ilabt.imec.be in your browser to visit the Grafana user interface. You can login with the username and password as specified in the credentials file. Add the InfluxDB time-series database as a data source in Grafana. Here are some guidelines on how to use InfluxDB in Grafana, or InfluxDB's page on how to get started visualizing with Grafana. Note that you can either use the internal Kubernetes DNS or the browser. The DNS name for the InfluxDB service is **http://influx.database.cloudandmobile.ilabt.imec.be/**

Create a Grafana dashboard to visualize the sensor values and count from the Raspberry Pi. The dashboard should look similar to the result shown in Figure 7.

Figure 7: Task 3 - Result

6.4 Optional Task 4: Query for active RFID IDs

Do not start with this task unless you have done the previous tasks! Solving this last task correctly is meant for students that want to distinguish themselves.

- **/current/** - GET - Query the database for the active `rfid_ids`, or those only scanned an odd number of times.

Write a new endpoint that queries InfluxDB to only get the list of RFID IDs of active IDs, or only those that have been scanned an odd number of times. The endpoint should return a list of strings. Note that we expect the logic of finding the RFIDs that have been scanned an odd number of times to be in the Flux query and not in Python code.

7 Questions

Answer the following (open) questions in your lab report:

- Which steps are required to add a new container (e.g. an egress API that exposes the time-series data from InfluxDB) to this architecture?

8 Summary

A microservice-based architecture improves the modularity and scalability of an application. This lab session provides an introduction to microservices and the technologies that drive this architectural evolution: Docker containerization for ease-of-deployment and REST APIs for inter-service communication. With this lab we also set up our Ingress API that can receive sensor data from an outside source, and InfluxDB, a time-series database that receives that data for processing and visualization. We run all of the microservices of our IoT architecture in the cloud.

9 Material to submit

Give the lab report a structure that mimics the structure of this document. If you had to do anything noteworthy in the preparation or tutorial parts of this lab, they should be included in the report. For example, how you solved the errors when deploying your Lab2 code to the cluster for the first time should be included, along with a screenshot showing you got it deployed and working in the end. In your lab report, elaborate on how you solved the tasks along with an explanation of the code that you wrote. Include **screenshots** in your report or **videos** in your zip to show that you completed each task successfully. Don't forget to answer the question above in your report. Only the .pdf extension is allowed for the report. Archive the report together with the source code for the exercises, and name this archive **Lab3_FamilyName_FirstName.zip**. Hand everything in using Ufora.