INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

HYDERABAD

IOT Based Application Platform

Design Document

IAS - Spring 2023

Group - 05

1) Overview

With growing IoT use, respective device data is also growing at a similar rate. Each sensor sends potential data which can be used to make an important decision. The difficulty arises for a developer to keep track of this sensor data and use this data as input in its application. Managing algorithms, binding data, and analyzing the output are some big tasks a developer has to take care of. The aim of this platform is to provide the user with an environment that allows them to create applications which can connect to / use location-sensitive IoT devices with ease. In addition to this, it also provides the capability to dynamically run user-defined algorithms which make use of such devices. This IoT Application platform directly integrates with the IoT devices and is capable of performing remote actions on these devices.

List of Teams Module:

1. Deployment Manager
2. Sensor Manager
3. Application Controller, Scheduler
4. Node Manager, Load Balancer
5. Monitoring and Fault Tolerance, Initializer

2) Use Cases & Requirements

a) Use Cases:

i) **Farm management System App** : To increase the productivity of agricultural and farming processes in order to improve yields and cost-effectiveness with sensor data collection by sensors like soil moisture, temperature sensor, ph sensor, water level sensor. We will use sensor data collection for measurements to make accurate decisions in future. It can be used as a reference for members of the agricultural industry to improve and develop the use of IoT to enhance agricultural production efficiencies. The sensor types are air-condition-sensor, for measuring the temperature of the soil, and soil-moisture-sensor, for measuring the humidity of the sensor and turning off or on the sprinkler based on the threshold.

ii) **Smart fire detection and Sprinkler System** : The smart fire detection system uses various sensors such as smoke detectors, heat detectors, and flame detectors to detect the presence of fire. These sensors are connected to a central control panel that can identify the location of the fire and alert the building occupants and the fire department.The sprinkler system is designed to automatically release water or other extinguishing agents to extinguish the fire once it has been detected. The sprinkler system is typically composed of a network of pipes, sprinkler heads, and control valves.In a smart fire detection and sprinkler system, the sprinkler system can be activated selectively based on the location of the fire, thanks to the information provided by the smart fire detection system. This allows for more efficient use of water and other resources, as well as reducing water damage to unaffected areas of the building.

b) **Requirements:**
(Very detailed documentation of requirements. For the group documentation it will be about overall platform while in the Team documentation it will be more focused about specific components)

3) Test cases

3.1 Test cases

a) Sensor Manager

Test Cases	Input	Output	Expected Result
Registration			
config file is parsed correctly and all mandatory meta fields are present	config file	Success Message	Sensor type schema is created
unable to parse config file	config file	Error Message	Send error code 500 unable to parse json file
config file is parsed correctly and mandatory meta fields are absent	config file	Error Message	Send error code 500 unable to parse json file
Device type is not present in repository	config file	Error Message	Send error code 403 Unable to Allocate Device
Location is not present	config file	Error Message	Send error code 403 Unable to Allocate Device
Sensor Data			
Kafka Stream available	Sensor data	Success Message	Sensor data is saved to DB and an instance is published on kafka stream
Kafka Stream unavailable	Sensor data	Error Message	Save message to logs : sensor id :broker not found

b) Load Balancer
   a. On requesting a node to deploy a service load balancer should return the node to deploy the service.

Input - Deployer requests a node to deploy the service and all nodes provided by node manager are at its full capacity.
Output - Request the node manager to create a new node and provide its details.

b. Load balancer finds that service s1 running on node n1 and container c1 is overloaded. So it sends details of the service to the deployer to create one new instance of that service running.
Input: LB finds c1 overloaded.
Output: LB sends details of s1 to deployer.

c) Node Manager

a. When Load Balancer requests the details of the active nodes, Node Manager sends the details of active nodes to Load Balancer.
Input - Deployer requests a node to deploy the service and all nodes provided by node manager are at its full capacity.
Output - Request the node manager to create a new node and provide its details
b. On request of new node creation from Load Balancer, Node Manager creates a new node.
Input - Load Balancer requests to create a new node.
Output - Node manager creates the new node and sends its details to load balancer.

d) Deployment Manager

a. Input: The scheduler requests the deployment manager to deploy an application on an available node.
Output: The deployment manager, before deployment checks if the application and the algorithm that is to be deployed is a valid one. If valid, it creates a docker image and uploads it to the Azure container repository.
b. Input: Load Balancer provides the node name and address of the best node.
Output: Deployer passes on the request to additional sub-modules for the purpose of running the file. The image is then containerized and runs on the best node provided by load balancer.
c. Input: Node manager should be informed about the selected node for deployment.
Output: Send the selected node information to the node manager.
d. Input: Node manager sends acknowledgement.
Output: Deploy application on the instance of the selected node.
e. Input: The scheduler's request to initiate the deployment of a service is in an incorrect format.
Output: Deployer sends error message to scheduler.
f. Input: Deployment manager fails to connect with load balancer.
Output: Deployer waits for sometime and resends connection requests.

e) App Controller **-**

a. When a user arrives on the platform it'll be first authenticated and then access will be provided of its designated role.
   Input: User Credentials
   Output: If the user is genuine, redirect it to the home page of the platform.
b. When a user submits an application, the zip file App Controller will unzip the file and validate the files and their structure against the platform standards.
   Input: App.Zip File
   Output: Boolean True/False
c. If the files are valid then workflow info files are then used to generate the workflow code and then store the code files in the repository.
   Input: Workflow info file
   Output: Workflow code
d. All the user requests are targeted at the application controller(Action Manager) on which it'll inform the scheduler to schedule the requested service.
   Input: API request stating the service
   Output: Acknowledgement
   When the platform configurer provides the scheduling information.

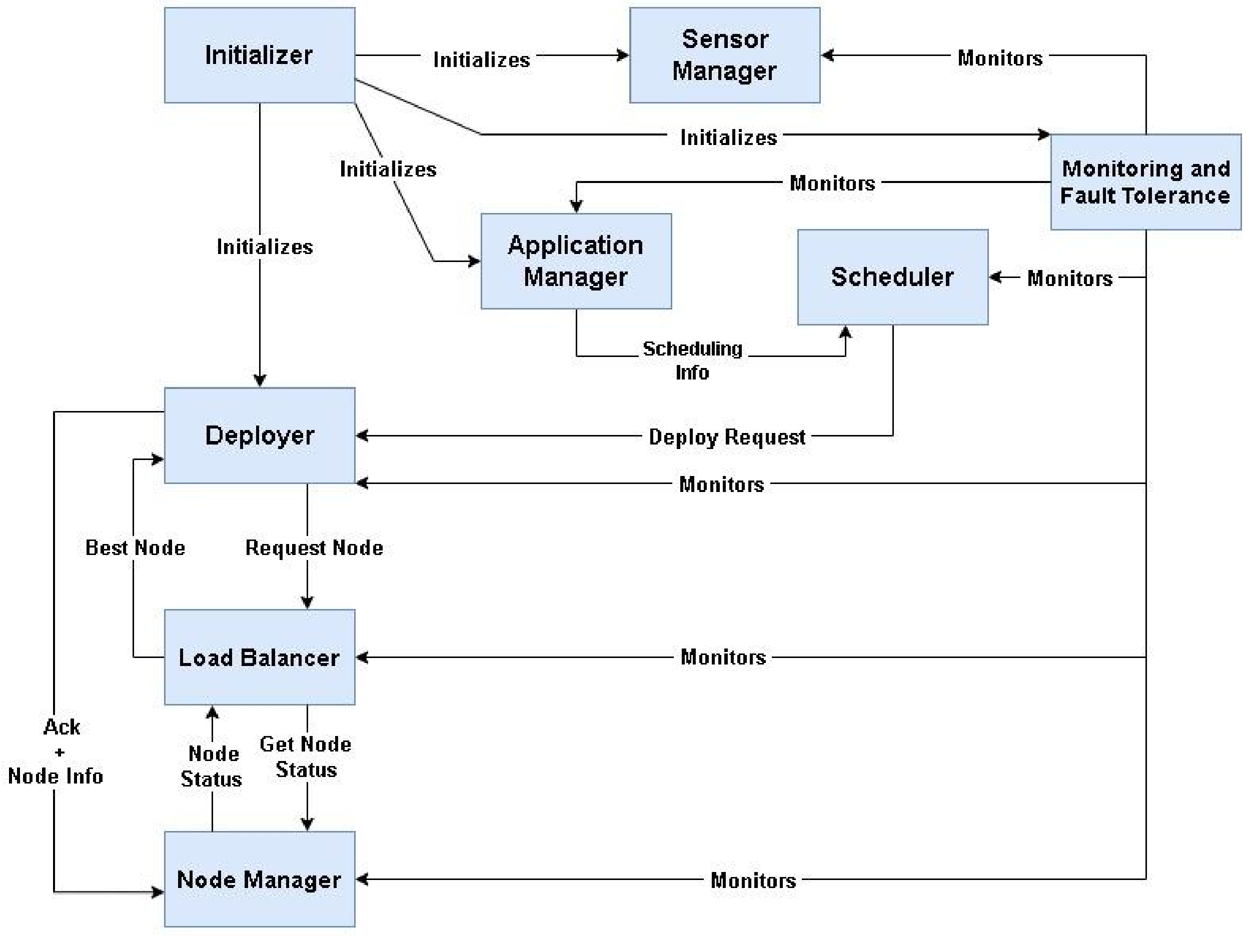
f) Scheduler
   a. When the scheduler receives the request to schedule a job it'll put it on a priority queue based on its priority(time). It'll have the application id and service information which it'll provide to the deployer on its schedule time.
      Input: API request to schedule a job
      Output: Entry on the queue for the job

3.2 Overall project test cases

(relevant to the module) - Integration testing - How & what to test, after integrating with other modules - This should be based on overall use-cases listed in Group requests.

4) Solution design considerations

4.1 Interactions between components



4.2 Environment to be used

- Linux OS

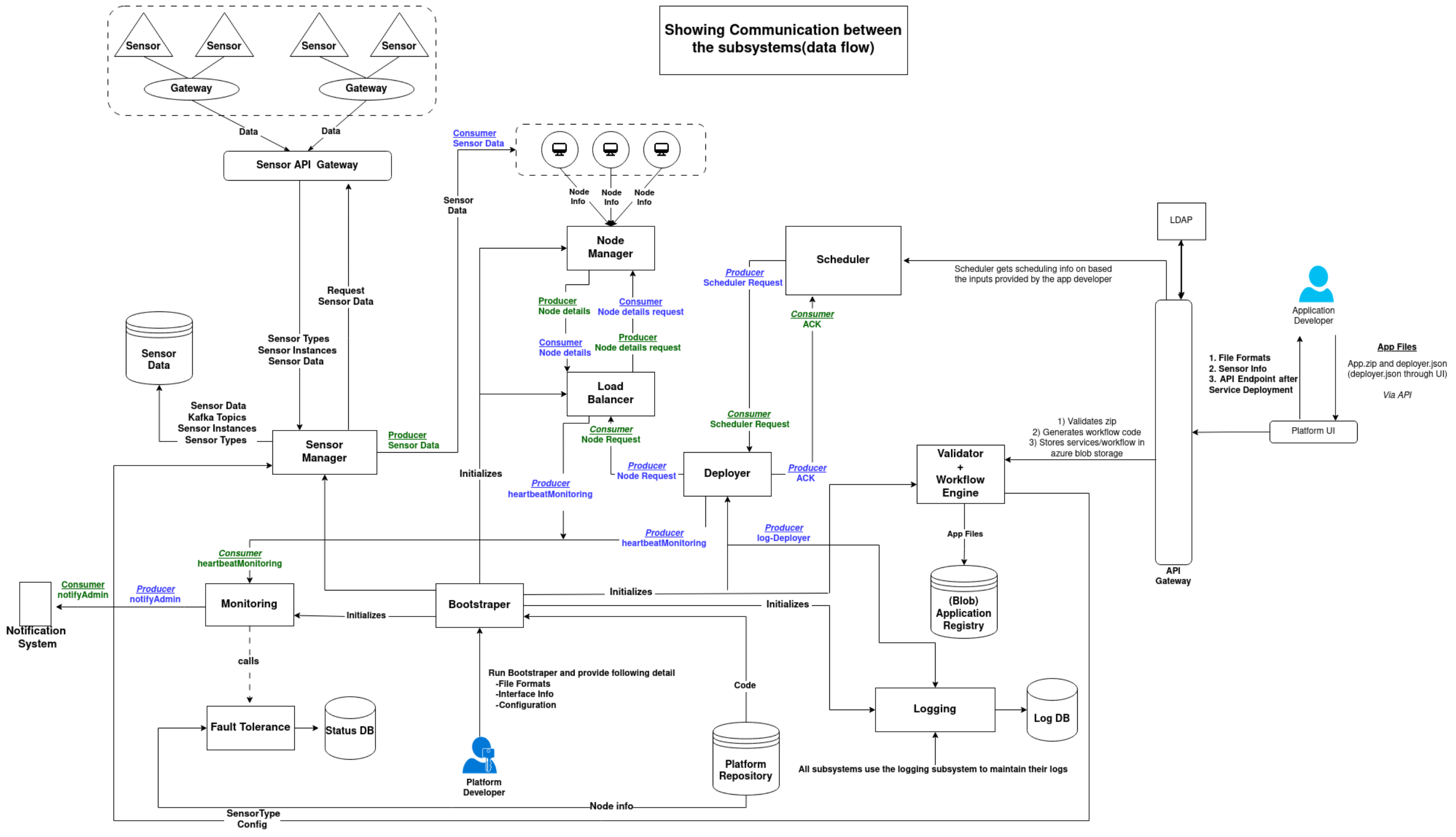
4.3 Technologies to be used

1. Python - Overall Development
   - Large developer community.
   - Extensive libraries.
   - User Friendly data structures
   - Portability
2. Apache Kafka - for communication
   - Real-time streaming data pipelines
   - Scalable
   - Publish-Subscribe Model
3. MySQL/ MongoDB/ Azure / Azure Container Repository / Blob Storage
4. Docker - for platform independence and portability
   - Isolated environment
   - Mobility
   - Easy collaboration
5. Flask
6. React JS (Platform UI)
7. HTML, CSS (Sample App)

4.4 Overall system flow & interactions

- Application developer provides the application package to the platform, it is saved in the application repository.
- Configurer registers the sensors to the platform.
- The end user will then use the application and send requests(after being authenticated) for different functionalities that he/she wants to use.
- The request will be parsed by the application controller that determines the application that needs to be deployed.
- The scheduler then determines when the application will be deployed.
- The deployer then fetches the application code and config files from the application repository.
- The load balancer then determines the best available node where our application can be deployed.
- The deployer then creates the environment and sets up the dependencies as per the config files, and deployes/launches the application on the node in a container.
- After the application is deployed successfully, the end user can uses app services to communicate with sensors and perform analysis and visualization on sensor data

4.5 Communication Diagram


Showing Communication between the subsystems(data flow)
Sensor
Sensor
Sensor
Sensor
Gateway
Gateway
Data
Data
Sensor API Gateway
Consumer Sensor Data
Sensor Data
Node Info
Node Info
Node Info
Node Manager
Request Sensor Data
Sensor Types Sensor Instances Sensor Data
Sensor Data
Sensor Data Kafka Topics Sensor Instances Sensor Types
Sensor Manager
Producer Sensor Data
Producer Node details
Consumer Node details request
Consumer Node details
Producer Node details request
Load Balancer
Consumer Node Request
Producer heartbeatMonitoring
Initializes
Producer Node Request
Deployer
Scheduler
Producer Scheduler Request
Consumer ACK
Consumer Scheduler Request
Producer ACK
Scheduler gets scheduling info on based the inputs provided by the app developer
LDAP
Application Developer
App Files
App.zip and deployer.json (deployer.json through UI)
Via API
1. File Formats
2. Sensor Info
3. API Endpoint after Service Deployment
Platform UI
1) Validates zip
2) Generates workflow code
3) Stores services/workflow in azure blob storage
API Gateway
Validator + Workflow Engine
App Files
(Blob) Application Registry
Producer heartbeatMonitoring
Producer log-Deployer
Consumer heartbeatMonitoring
Consumer notifyAdmin
Producer notifyAdmin
Notification System
Monitoring
Initializes
Bootstraper
Initializes
Initializes
calls
Fault Tolerance
Status DB
Run Bootstraper and provide following detail
-File Formats
-Interface Info
-Configuration
Platform Developer
Code
Platform Repository
Logging
Log DB
All subsystems use the logging subsystem to maintain their logs
Node info
SensorType Config

4.6 Interaction with Sensors

- The incoming data from sensors is sent to KAFKA which is streamed to the Nodes as needed. The data of a sensor is bound to the application ID and service ID.

4.8 Approach for communication & connectivity

All the one way communication between sub components will be carried out using Kafka and two way communication between sub components will be carried out using API calls.

4.9 Registry & repository

- Platform Repository
- Application Repository
- Auth Database
- Sensor Data Storage
  - Sensor Repository

4.10 Server & service lifecycle

- **Server Lifecycle -** The responsibility of the Server Life Cycle Manager is to oversee and manage every aspect of a server's functioning throughout its life cycle.
- **Service Lifecycle -** This interacts with the server lifecycle manager and server lifecycle provides the node address to it. Then the service is deployed at the designated node address.

4.11 Scheduler

Initial Application Deployment:

- When the Application Controller delivers the request with the scheduling information, the scheduler retrieves the application zip from the application registry (application id, priority) based on the scheduling information.
- Uses a FCFS scheduling which takes application id.
- Sends the first element, along with all of its metadata, from the FCFS queue to the Deployment Manager.

End User Request:

- Scheduler receives requests for scheduling new end user requests and its priority.
- Uses a priority-based scheduling which takes requests and its priority.
- Sends the first element from the priority queue to the Deployment Manager.

4.12 Load Balancing

This module is responsible for providing the best node to the deployer to deploy a service and also checks for container utilization. It asks for node details from the node manager and selects the best node(one with the lowest node).

4.13 Interactions between modules

1. **Initializer and Other Modules**

   Sensor Manager, Application Manager and Deployer are initialized using this module.
2. **Application Manager and Scheduler**

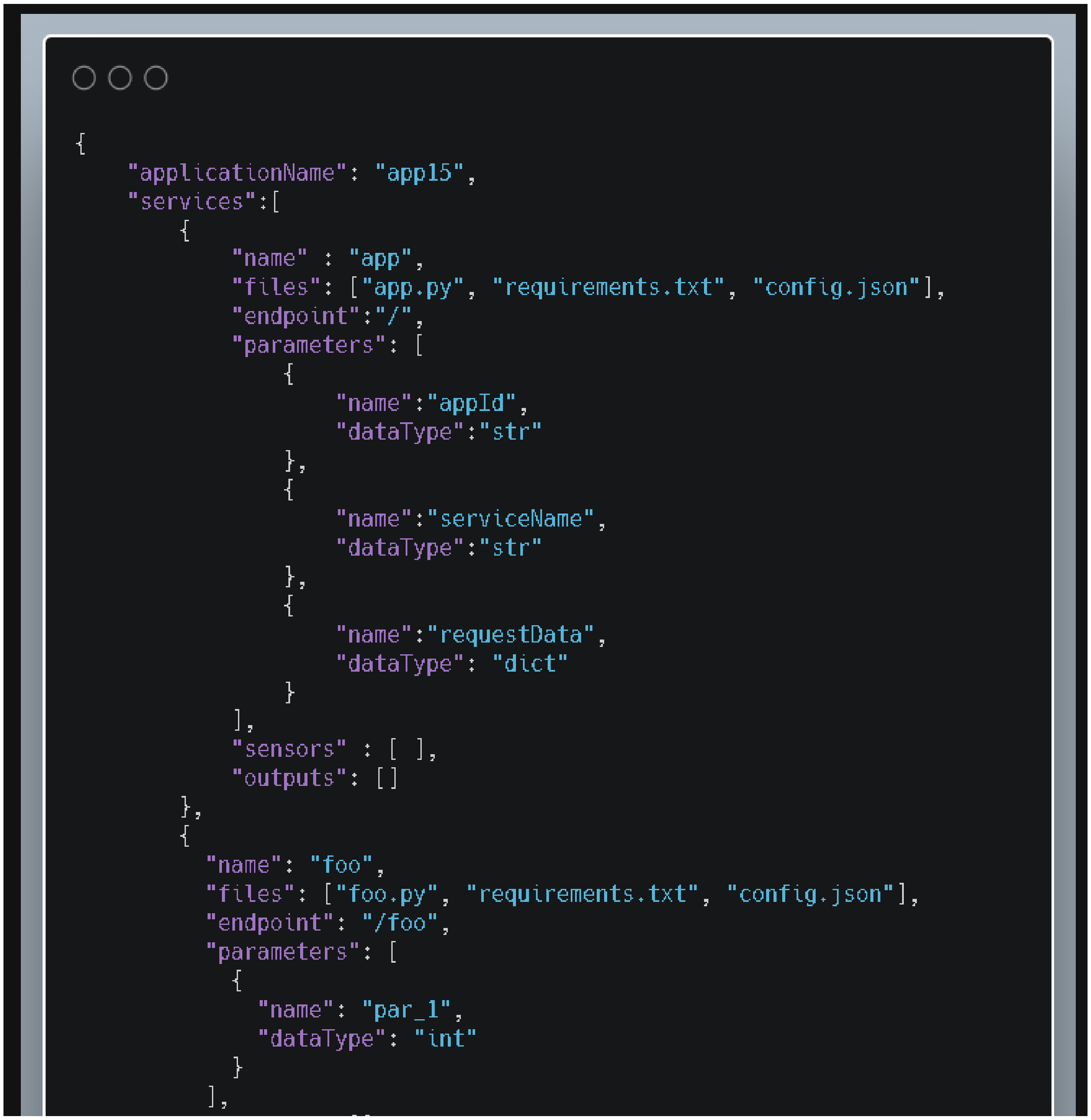
   Application manager sends information to the scheduler. This information includes application details and scheduling details. Using these scheduling details, the scheduler will perform actions for that application.
3. **Scheduler and Deployer**

Scheduler will reach the Deployer as per schedule. It will provide information about the application.

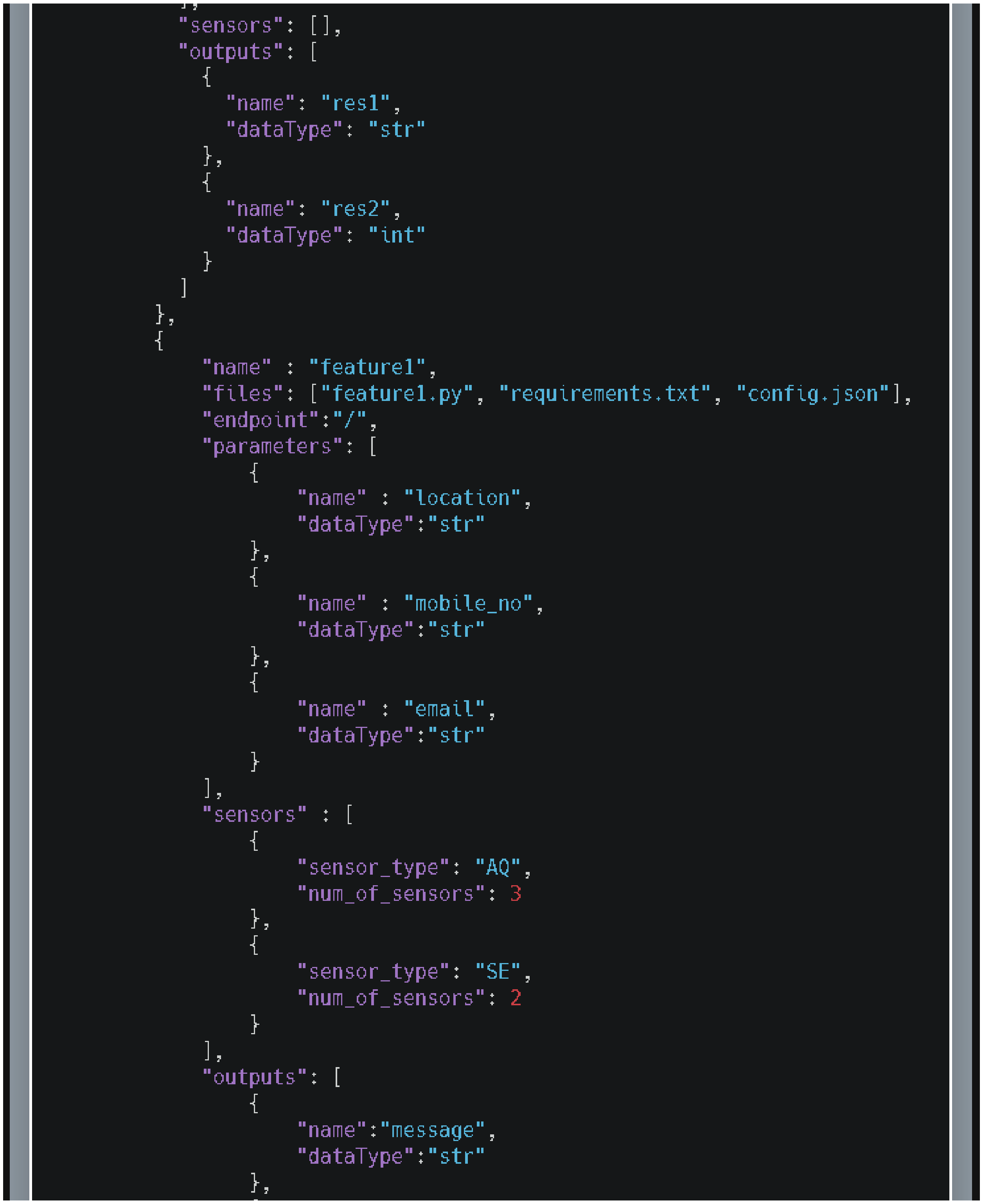
4. **Load Balancer and Deployer**
Deployer will request a load balancer for a node in order to run a service. Load Balancer will give details of the best node using the Load Balancing Algorithm.
5. **Node Manager and Load Balancer**
Load Balancer contacts Node Manager for details of Nodes like number of services executing on nodes, type of services etc. Node Manager will provide these details to Load Balancer.
6. **Monitoring & Fault Tolerance and Other Modules**
Monitoring module checks that other modules are working properly or not. If the module is down, it will take actions accordingly. It checks for application manager, scheduler, deployer, load balancer, node manager.

4.14 Wire and file formats

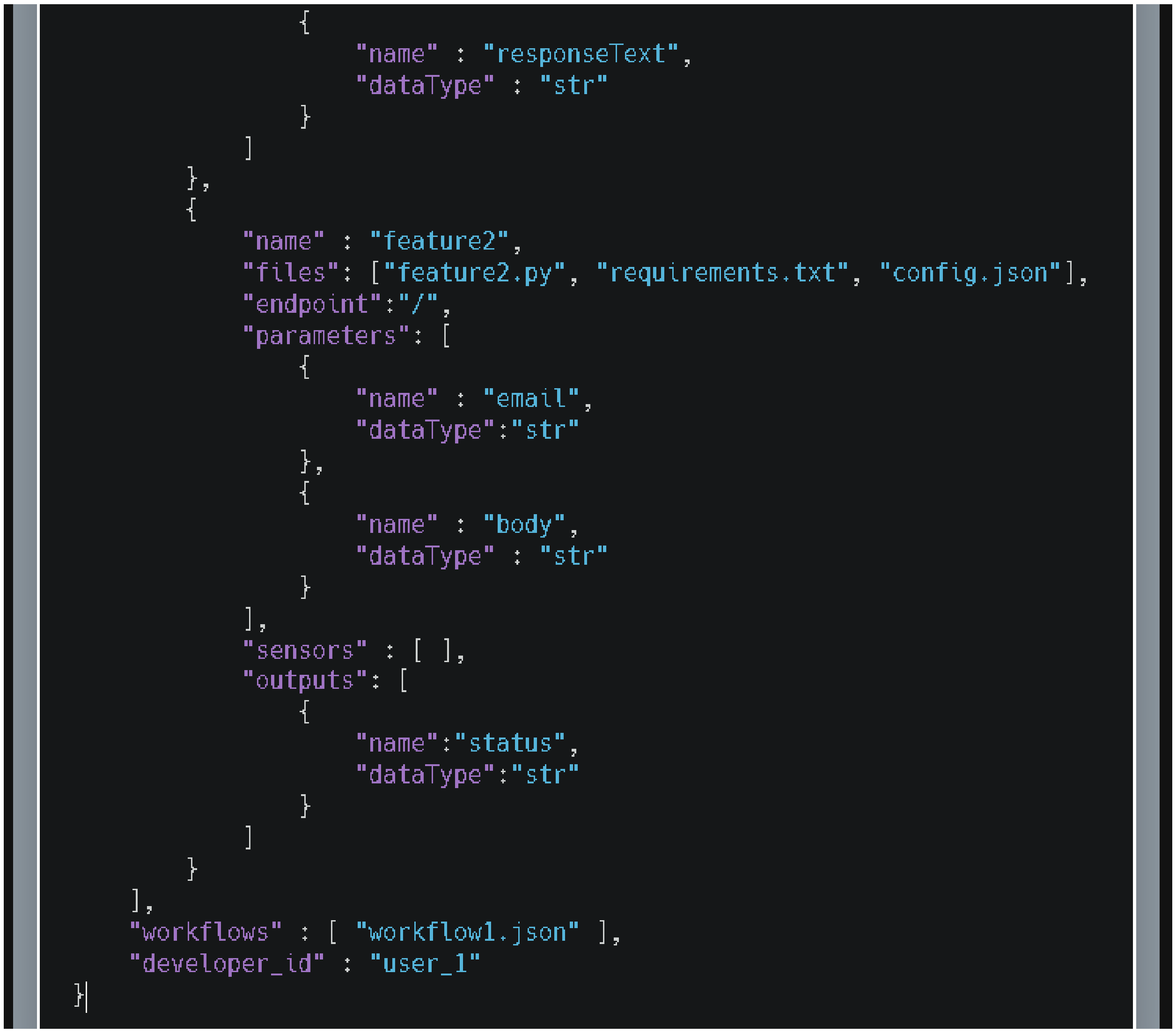
1. appConfig.json



```
{
    "applicationName": "app15",
    "services":[
        {
            "name" : "app",
            "files": ["app.py", "requirements.txt", "config.json"],
            "endpoint":"/",
            "parameters": [
                {
                    "name":"appId",
                    "dataType":"str"
                },
                {
                    "name":"serviceName",
                    "dataType":"str"
                },
                {
                    "name":"requestData",
                    "dataType": "dict"
                }
            ],
            "sensors" : [ ],
            "outputs": []
        },
        {
          "name": "foo",
          "files": ["foo.py", "requirements.txt", "config.json"],
          "endpoint": "/foo",
          "parameters": [
            {
              "name": "par_1",
              "dataType": "int"
            }
          ],
```

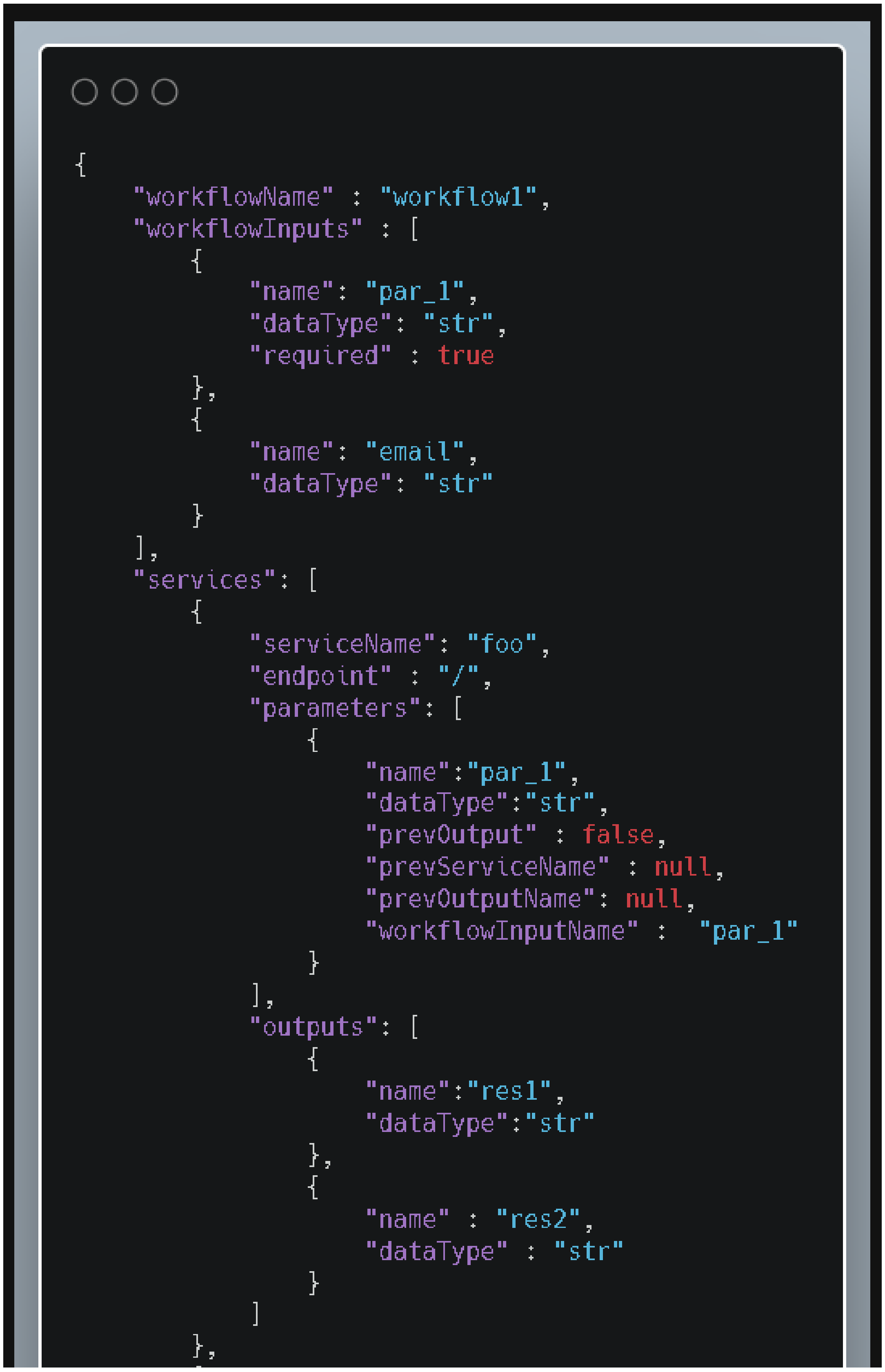


```
            "sensors": [],
            "outputs": [
              {
                "name": "res1",
                "dataType": "str"
              },
              {
                "name": "res2",
                "dataType": "int"
              }
            ]
          },
          {
              "name" : "feature1",
              "files": ["feature1.py", "requirements.txt", "config.json"],
              "endpoint":"/",
              "parameters": [
                  {
                      "name" : "location",
                      "dataType":"str"
                  },
                  {
                      "name" : "mobile_no",
                      "dataType":"str"
                  },
                  {
                      "name" : "email",
                      "dataType":"str"
                  }
              ],
              "sensors" : [
                  {
                      "sensor_type": "AQ",
                      "num_of_sensors": 3
                  },
                  {
                      "sensor_type": "SE",
                      "num_of_sensors": 2
                  }
              ],
              "outputs": [
                  {
                      "name":"message",
                      "dataType":"str"
                  },
```

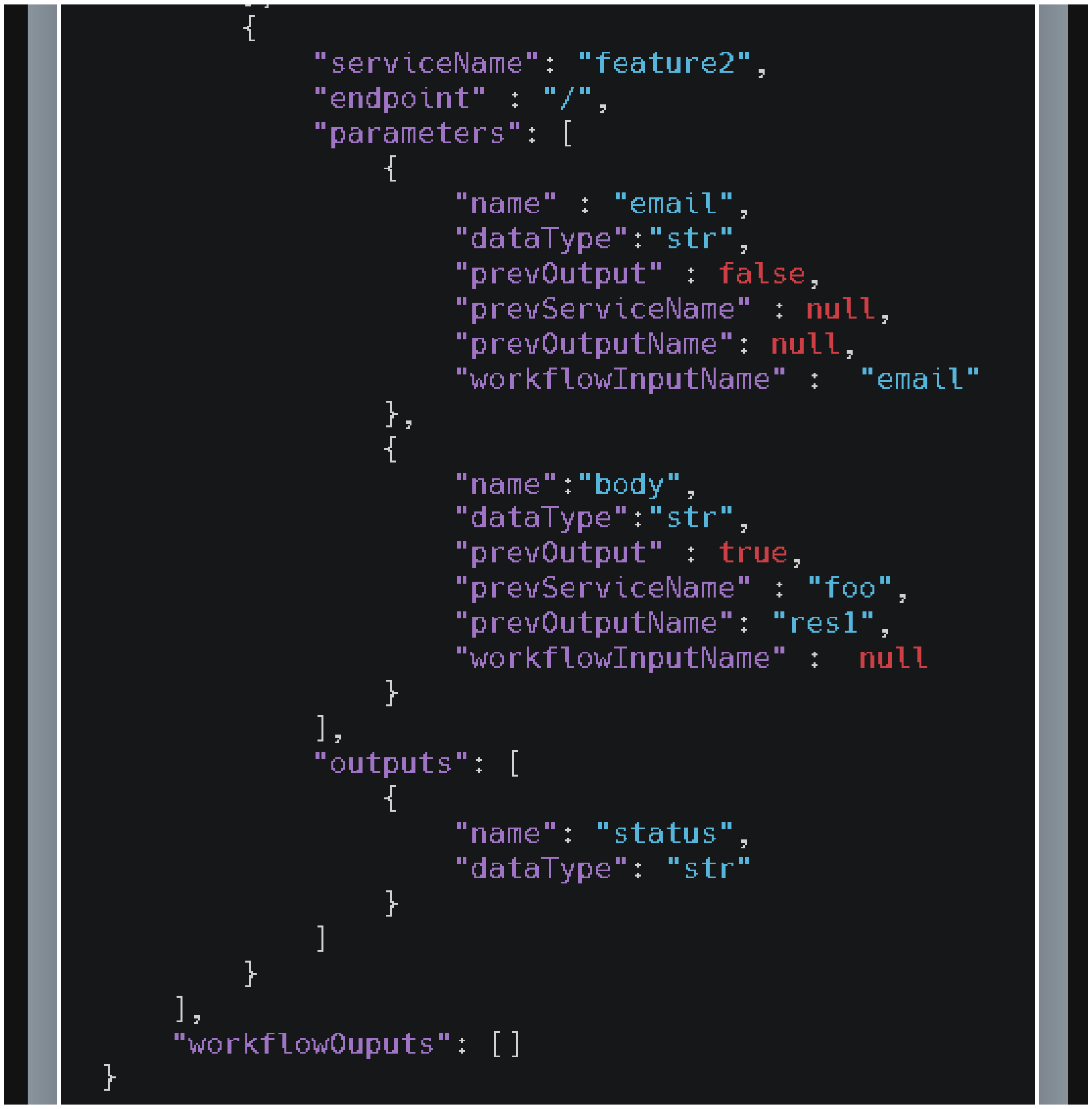


```
                    {
                        "name" : "responseText",
                        "dataType" : "str"
                    }
                ]
            },
            {
                "name" : "feature2",
                "files": ["feature2.py", "requirements.txt", "config.json"],
                "endpoint":"/",
                "parameters": [
                    {
                        "name" : "email",
                        "dataType":"str"
                    },
                    {
                        "name" : "body",
                        "dataType" : "str"
                    }
                ],
                "sensors" : [ ],
                "outputs": [
                    {
                        "name":"status",
                        "dataType":"str"
                    }
                ]
            }
        ],
        "workflows" : [ "workflow1.json" ],
        "developer_id" : "user_1"
}
```

2. Workflow.json: eg: workflow1.json

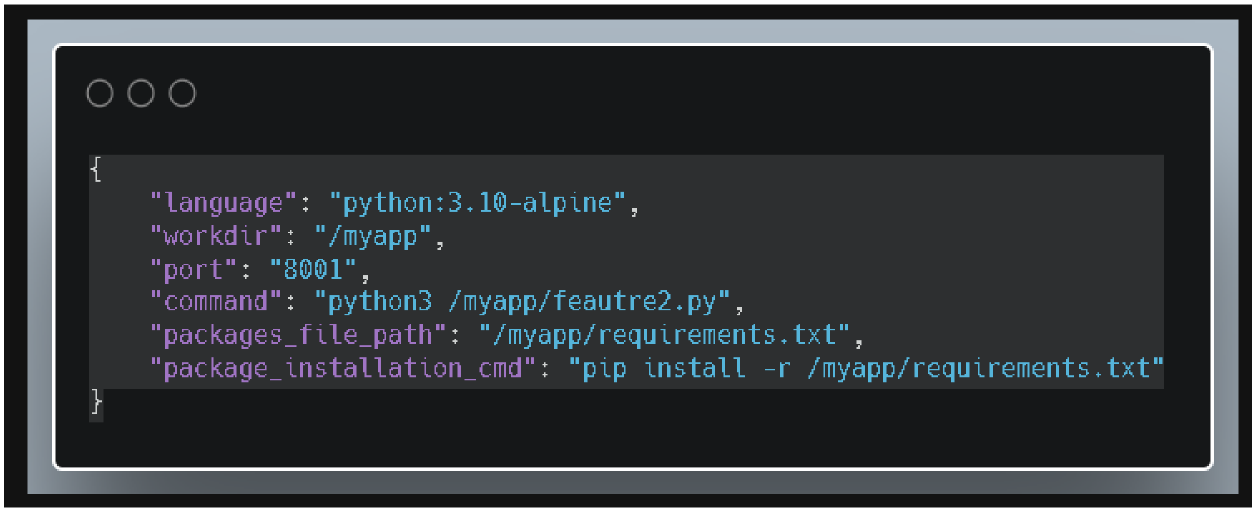


```
{
    "workflowName" : "workflow1",
    "workflowInputs" : [
        {
            "name": "par_1",
            "dataType": "str",
            "required" : true
        },
        {
            "name": "email",
            "dataType": "str"
        }
    ],
    "services": [
        {
            "serviceName": "foo",
            "endpoint" : "/",
            "parameters": [
                {
                    "name":"par_1",
                    "dataType":"str",
                    "prevOutput" : false,
                    "prevServiceName" : null,
                    "prevOutputName": null,
                    "workflowInputName" :  "par_1"
                }
            ],
            "outputs": [
                {
                    "name":"res1",
                    "dataType":"str"
                },
                {
                    "name" : "res2",
                    "dataType" : "str"
                }
            ]
        },
```



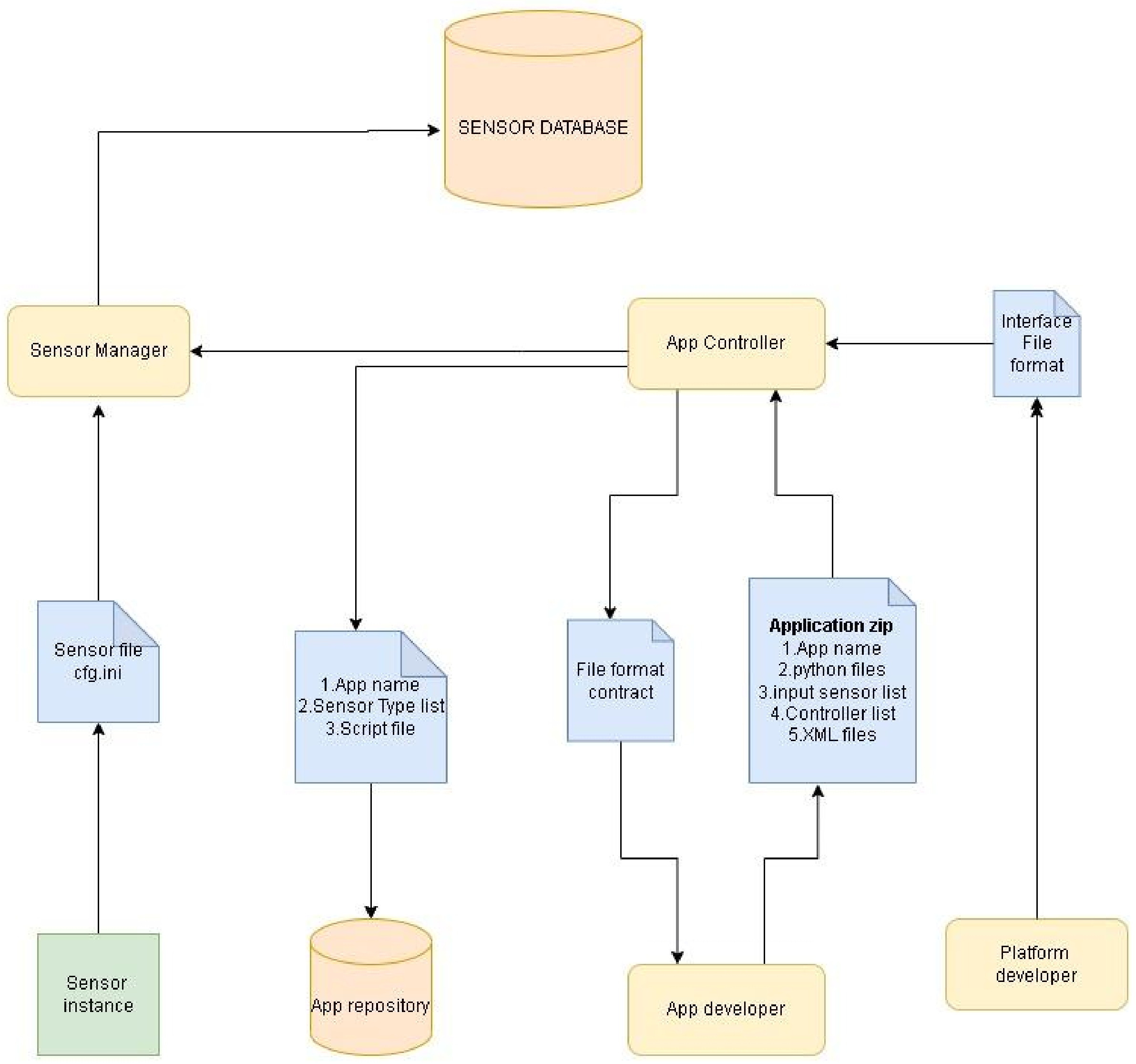
```
            {
                "serviceName": "feature2",
                "endpoint" : "/",
                "parameters": [
                    {
                        "name" : "email",
                        "dataType":"str",
                        "prevOutput" : false,
                        "prevServiceName" : null,
                        "prevOutputName": null,
                        "workflowInputName" :  "email"
                    },
                    {
                        "name":"body",
                        "dataType":"str",
                        "prevOutput" : true,
                        "prevServiceName" : "foo",
                        "prevOutputName": "res1",
                        "workflowInputName" :  null
                    }
                ],
                "outputs": [
                    {
                        "name": "status",
                        "dataType": "str"
                    }
                ]
            }
        ],
        "workflowOuputs": []
    }
```

3. **Config.json - present in each service**

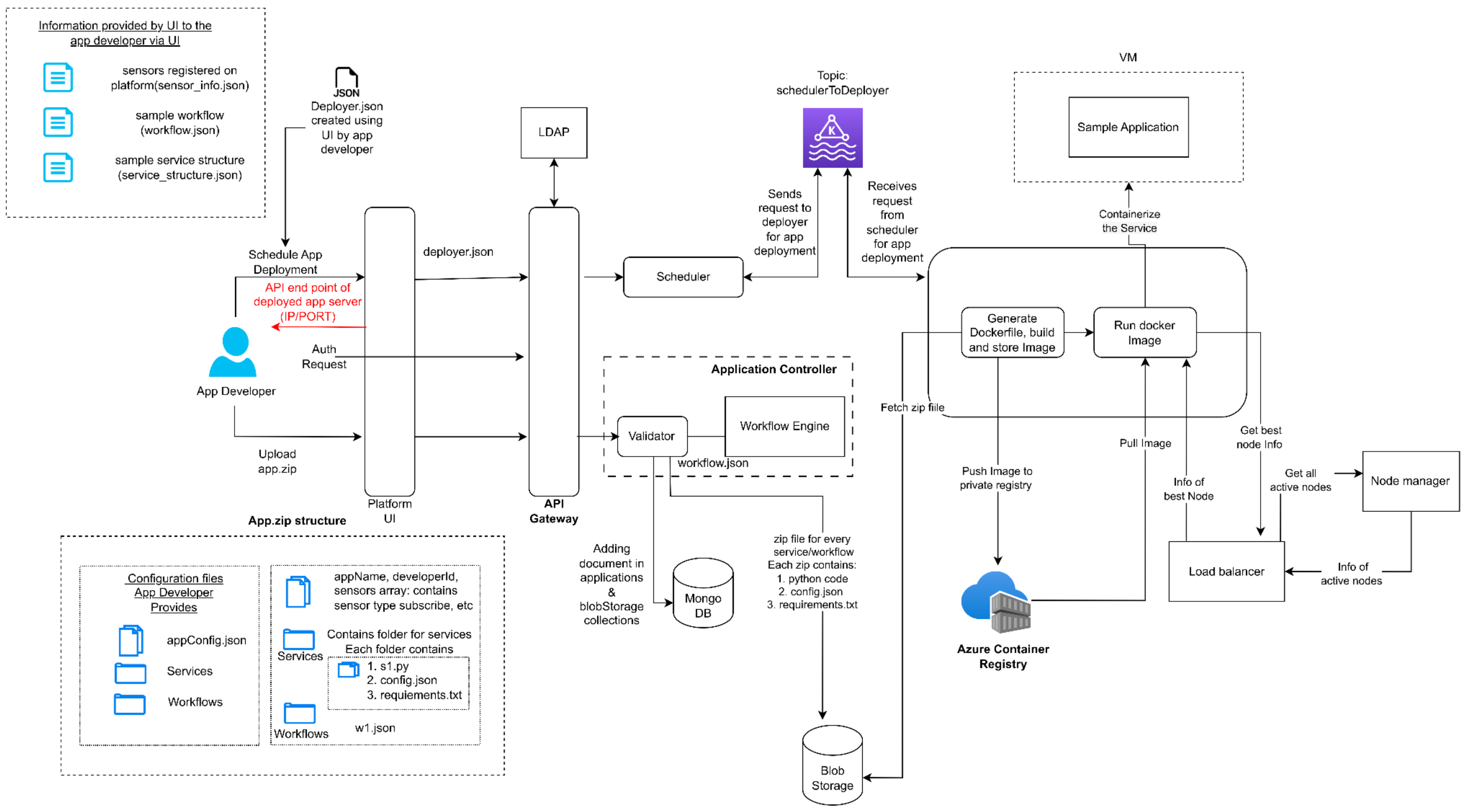


```
{
    "language": "python:3.10-alpine",
    "workdir": "/myapp",
    "port": "8001",
    "command": "python3 /myapp/feautre2.py",
    "packages_file_path": "/myapp/requirements.txt",
    "package_installation_cmd": "pip install -r /myapp/requirements.txt"
}
```

4.15 Artifact Diagram



5. Application Model


Information provided by UI to the app developer via UI
sensors registered on platform(sensor_info.json)
sample workflow (workflow.json)
sample service structure (service_structure.json)
JSON
Deployer.json created using UI by app developer
LDAP
Topic: schedulerToDeployer
VM
Sample Application
Sends request to deployer for app deployment
Receives request from scheduler for app deployment
Containerize the Service
Schedule App Deployment
API end point of deployed app server (IP/PORT)
deployer.json
Scheduler
Generate Dockerfile, build and store Image
Run docker Image
Auth Request
App Developer
Application Controller
Validator
Workflow Engine
workflow.json
Fetch zip file
Get best node Info
Pull Image
Push Image to private registry
Info of best Node
Get all active nodes
Node manager
Upload app.zip
Platform UI
API Gateway
App.zip structure
Adding document in applications & blobStorage collections
Mongo DB
zip file for every service/workflow Each zip contains: 1. python code 2. config.json 3. requirements.txt
Load balancer
Info of active nodes
Azure Container Registry
Configuration files App Developer Provides
appConfig.json
Services
Workflows
appName, developerId, sensors array: contains sensor type subscribe, etc
Contains folder for services Each folder contains
1. s1.py
2. config.json
3. requiements.txt
Services
Workflows
w1.json
Blob Storage

6. Key Data structures

1. **Queue :** A queue can be used to allocate nodes.
2. **Min Heap** : In job scheduling.
3. **Map** : To map code with application ID.
4. **Dictionary:** Contains the status of activeness of each node.

7. Interactions & Interfaces APIs

- Interaction Between Platform and Application Developer :- Platform developer is responsible for providing interface data to the application developer. Interface data will provide information to the application developer how they'll package the application data. Application developers will then provide details of the application like config info,scripts and workflow related information.

8. Persistence

- Sensor Data Storage
  - Sensor repository is used to store the sensor related data. So that if it is required by some service at a later time then it can be obtained.
- Application Repository
  - All the application related data like config files and scripts is stored in the application repository.
  - Application controller will validate the files provided by the application developer and if they are valid then those are stored in the application repository.
  - Deployer will use the application repository to obtain the code files at the time of deploying an application.

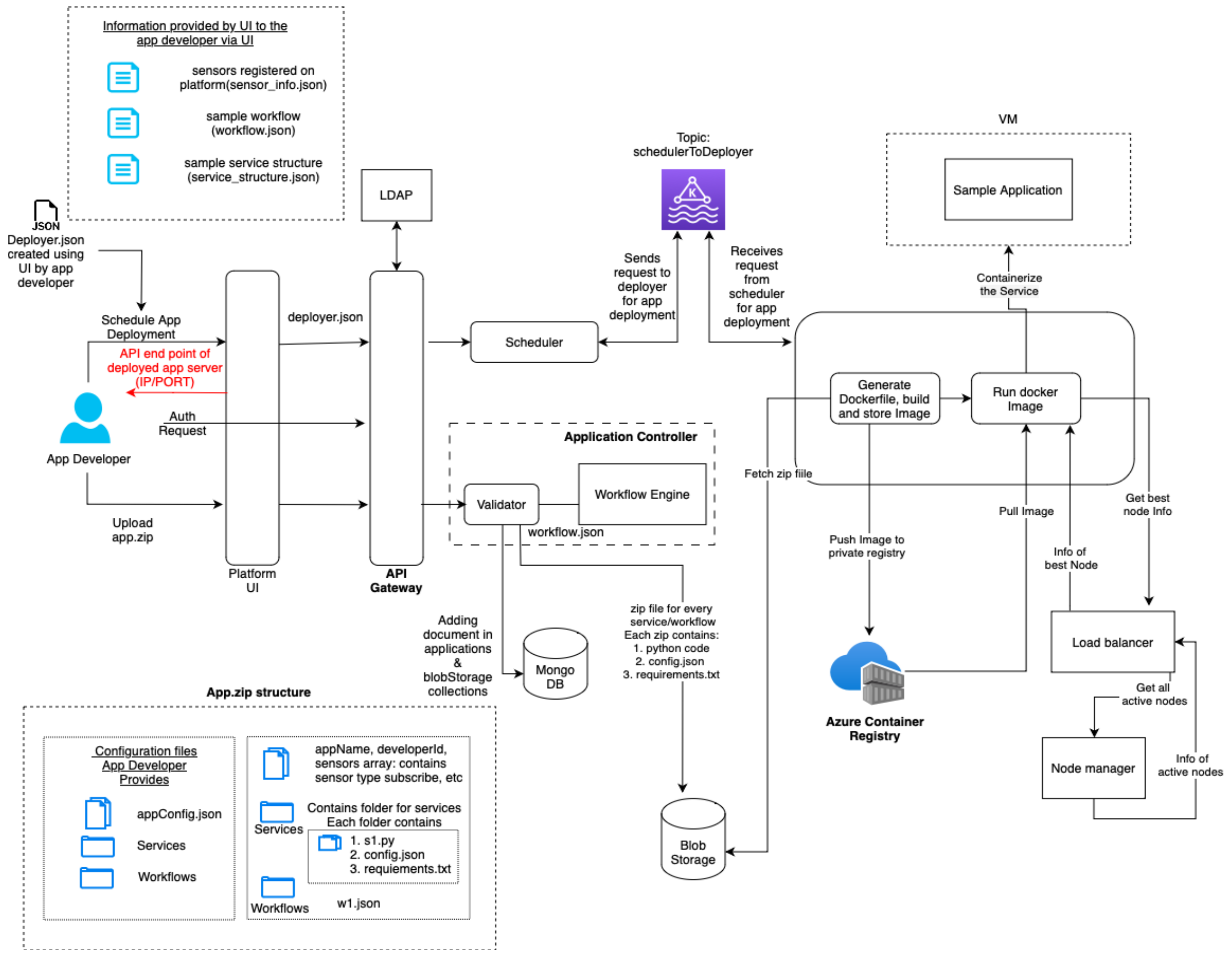
- Platform Repository
  - All the platform related data like config files and scripts for starting different subsystems is stored in the platform repository.
  - Will be useful when any subsystem goes down. The platform initializer will use it to make the subsystem go live again.

- Node DB
  - It stores the status of active nodes.
  - It will be useful when the Load balancer asks for node details.
- In case a node crashes, the platform brings it back to its previous saved state on some other available node.
- In case any platform or application service crashes then it will be made alive by monitoring and fault tolerance service.

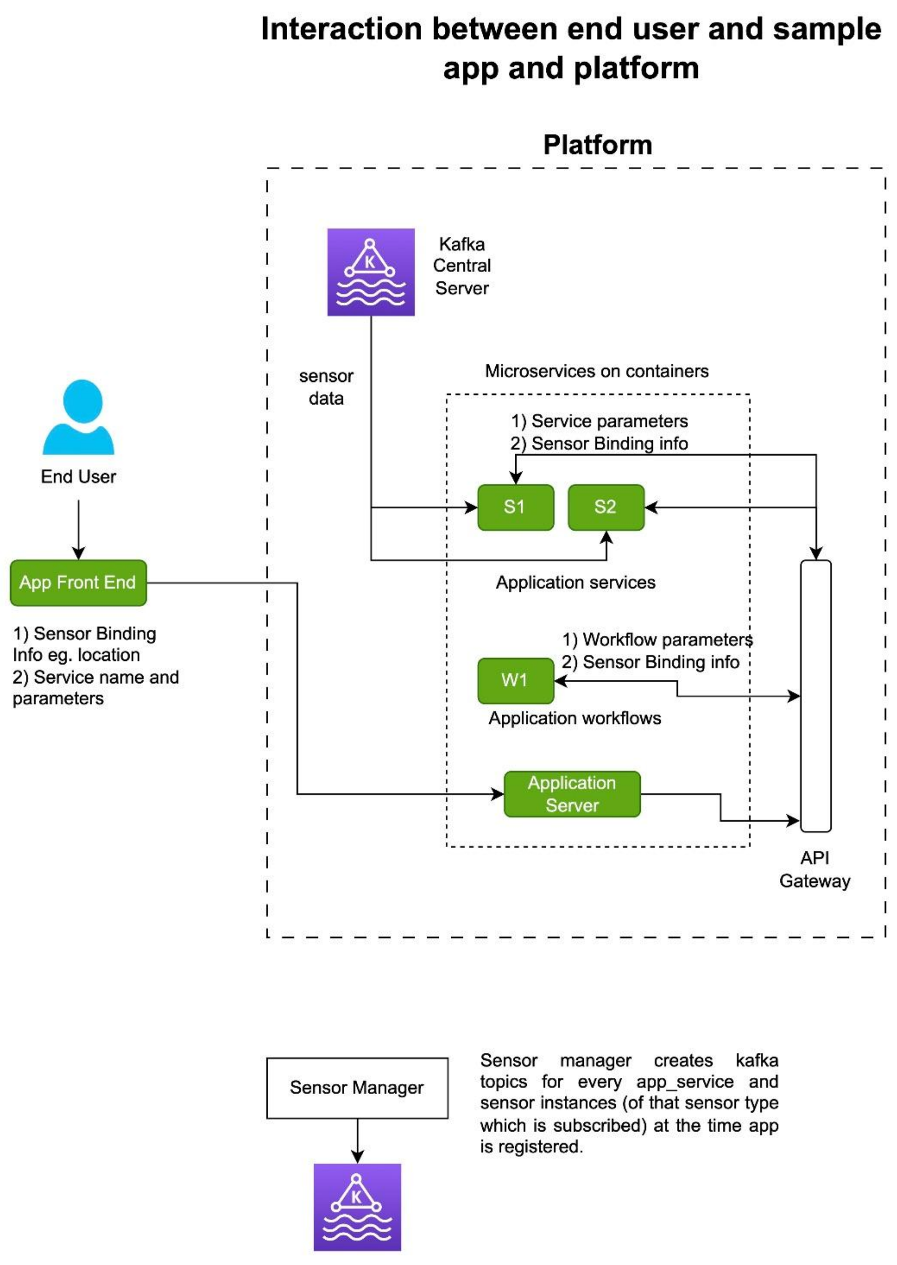
9. The modules

Internal design overview (additional details like above, for each model)
Team 1 - Deployer, Boostraper, LDAP, Platform UI
Team 2 - Application Controller(Validator & Workflow Manager), Scheduler, Kafka Central
Team 3 - Node Manager and Load Balancer
Team 4 - Sensor Manager
Team5 - Logging, Monitoring & Fault Tolerance

Application Model


Information provided by UI to the app developer via UI
sensors registered on platform(sensor_info.json)
sample workflow (workflow.json)
sample service structure (service_structure.json)
JSON
Deployer.json created using UI by app developer
Schedule App Deployment
API end point of deployed app server (IP/PORT)
Auth Request
App Developer
Upload app.zip
Platform UI
deployer.json
LDAP
API Gateway
Scheduler
Topic: schedulerToDeployer
Sends request to deployer for app deployment
Receives request from scheduler for app deployment
VM
Sample Application
Containerize the Service
Generate Dockerfile, build and store Image
Run docker Image
Application Controller
Validator
Workflow Engine
workflow.json
Fetch zip fille
Pull Image
Get best node Info
Push Image to private registry
Info of best Node
Adding document in applications & blobStorage collections
Mongo DB
zip file for every service/workflow Each zip contains: 1. python code 2. config.json 3. requirements.txt
Azure Container Registry
Load balancer
Get all active nodes
Node manager
Info of active nodes
Blob Storage
App.zip structure
Configuration files App Developer Provides
appConfig.json
Services
Workflows
appName, developerId, sensors array: contains sensor type subscribe, etc
Contains folder for services Each folder contains
1. s1.py
2. config.json
3. requiements.txt
w1.json

Package Model:



Sensor manager creates kafka topics for every app_service and sensor instances (of that sensor type which is subscribed) at the time app is registered.

Team 1 :-

Deployer

The **Deployment subsystem** is a critical component of application deployment, responsible for managing the process of deploying an application from development to production environments. Its primary role is to automate the process of moving the application code, configurations, and other necessary files from the development environment to the production environment.

List of services:

- Listen to the scheduler to get app service deployment requests.
- Send a request to the Load balancer to select the best node to deploy the application.
- Query the application code and config files from the application repository.
- Make a Docker image of the application code files and push it to a docker private repository.
- Download the docker image and run it on a container in the specified node.
- Send deployment status(success/failure) to the scheduler.
- Stop/kill the application service container on request of the scheduler

Bootstraper

Initializes all other subsystems and sets up the Kafka central server, Logger and LDAP central server.

Platform UI

The platform UI consists of two interfaces, one for Administrator and the other for App developer. It consists of a signup and login page. User verification and authentication is done via LDAP server.

App developer UI view consists of the following tabs:

- Dashboard tab: It contains a map view that shows the location of the sensors that are being used in the app developers applications. It also shows the count for the number of applications registered, deployed and sensors registered by the app developer.
- Upload Application tab: This view will be used by the developer to upload their app zip files for application registration on the platform. Also, it shows a table view of all his previous uploaded applications along with their list of services and registered sensors.
- Schedule Application tab: This interface is used by the app developer to view the status of their apps and schedule their applications as well. Developer has the functionality to deploy his applications now, later or periodically.

Admin UI view will consist of the following tabs:

- Logs tab: From here the admin will be able to view the logs for each of the platform subsystem's instances that are up and running on the nodes.

- Sensor Registration tab: This view will enable the admin to register new sensors to the platform that can be used by the app developer to bind to their applications. He can also view the already registered sensors list from this view.

LDAP

LDAP is used in as enterprise networks for implementing centralized authentication and authorization systems. When bootstrapper is run, it initialized the LDAP server.
An automated script is run which sets the domain name as **"ias.group5"** and organization name is set as **IIITH**.
When the LDAP services starts, 2 groups are created, one of admin and other for the developers. Admin's ID is made and gets add to **Admin** group. When any Application developer signup it's ID is created and is added to group named **AppDeveloper**.

Block Diagram:

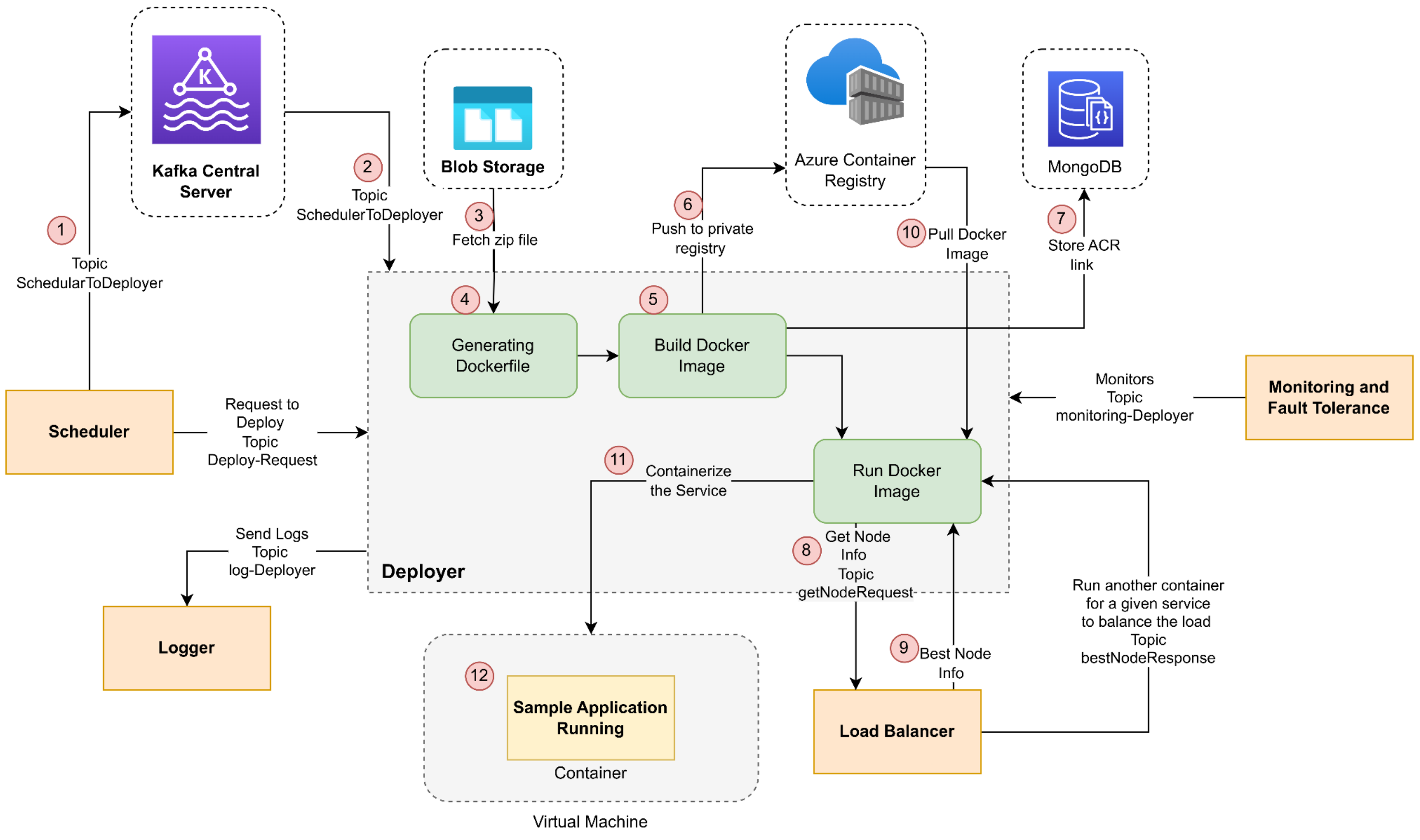

Kafka Central Server
Blob Storage
Azure Container Registry
MongoDB
1
Topic SchedularToDeployer
2
Topic SchedulerToDeployer
3
Fetch zip file
6
Push to private registry
10
Pull Docker Image
7
Store ACR link
4
Generating Dockerfile
5
Build Docker Image
Scheduler
Request to Deploy Topic Deploy-Request
Monitors Topic monitoring-Deployer
Monitoring and Fault Tolerance
11
Containerize the Service
Run Docker Image
Send Logs Topic log-Deployer
Deployer
8
Get Node Info Topic getNodeRequest
Run another container for a given service to balance the load Topic bestNodeResponse
Logger
12
Sample Application Running
Container
Virtual Machine
9
Best Node Info
Load Balancer

Figure: Deployer Block Diagram

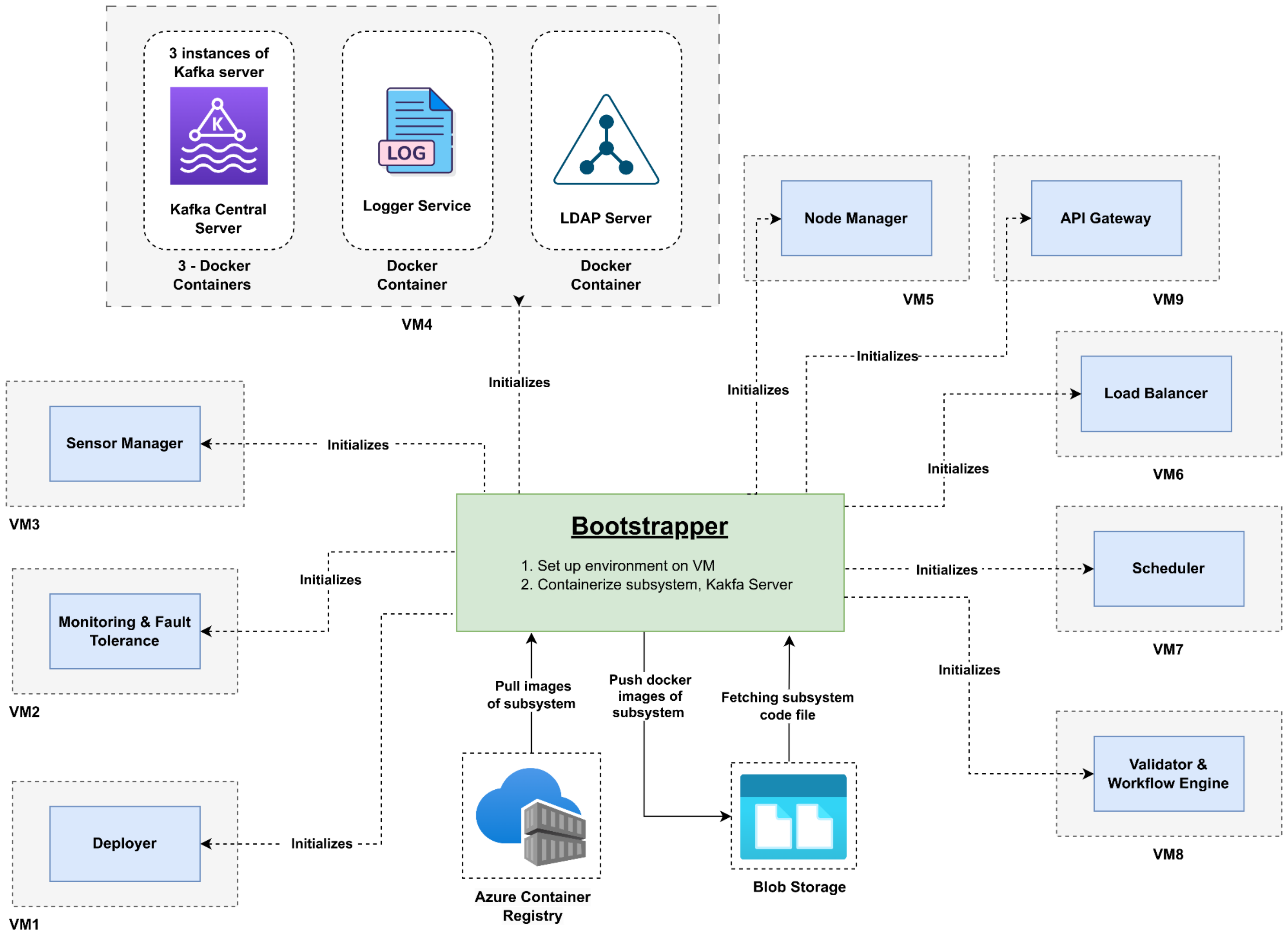


Figure : Block Diagram of Bootstrapper

Deployment Service Components:

- Deployer: The deployer will create an instance on the node selected by the load balancer and deploy the application there using the config and code files of the application sent by the scheduler.
- Database of Deployer: This database will have the code and config files needed by the platform initializer to make the deployer live and running.

- Database Handler: This handler will be used by the platform initializer to query the deployer code and config files.
- Application Repository Handler: This handler will be used by the deployer to query the application code and config files from the application repository.

Interactions of Deployer with other subsystems:

- Scheduler sends the Deployer the scheduling information about the application that needs to be deployed.
- The deployer then requests the load balancer to select an available node required for deployment.
- The deployer then queries the application repository to get the necessary code and config files of the application.
- After getting the available node information, we send it to the node manager.
- Deployer then creates an instance on the node and deploys the application and its services in that environment.
- Deployer also sends heartbeat signals to the Monitoring subsystem regularly.
- The initializer subsystem uses the deployer DB to make the deployer live and running.

Team 2 :-

1. Functional Overview - Validator,Workflow Manager, Scheduler and Kafka Central Server

Validator

- Takes appconfig.json, workflow.json and script files information in zip format, validates them against the rule base and stores them into the application repository(Azure Blob).
- Validator is also responsible for adding all the necessary libraries.

Workflow manager

- Is a module of application controller which will be responsible for generating the workflows from the workflow.json file.

Scheduler

- Schedules the user request for application deployment by maintaining a **priority queue** and **cron jobs**. It then selects the appropriate request according to the start time.
- Once the app is deployed

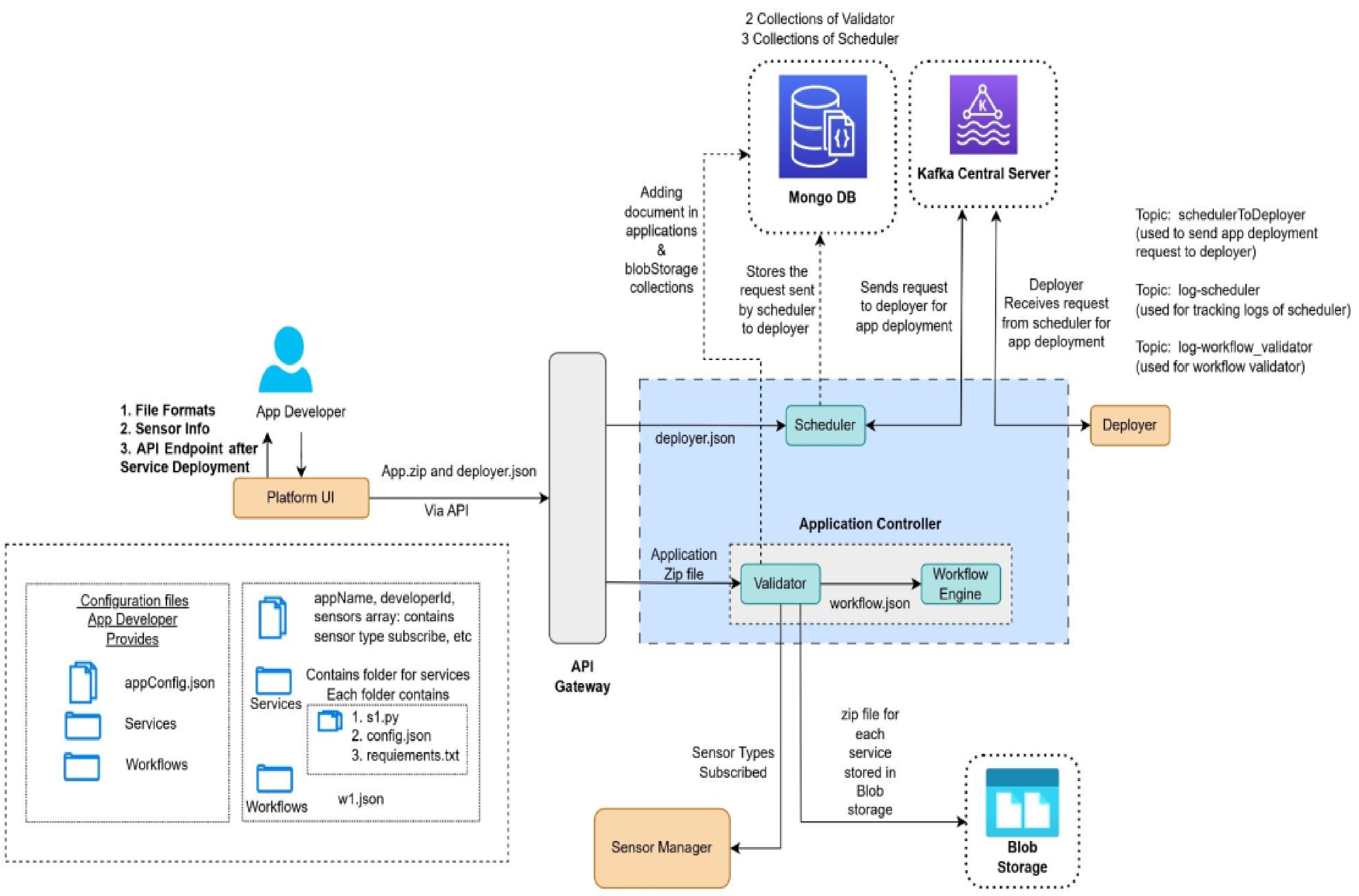
Kafka Central

- It is the backbone communication system of the entire platform which will be used for communication between all the subsystems.

2. List of services:

a. **Validation of App files**: Validates structure of configuration zips submitted by application developer. The structure should be the same as we provided by the Platform Developer in User Interface.
b. **Scheduling**: When the Application Controller delivers the request with the scheduling information, the scheduler retrieves the application zip from the application registry (application id, priority) based on the scheduling information.
c. **Application Repository**: Application Controller stores all the application files after validation and processing (Workflow Manager) in Application Repository database
d. **Workflow Management:** Workflow manager is a module of application controller which will be responsible for generating the workflows from the workflow.json file.

3. Block Diagram:


2 Collections of Validator
3 Collections of Scheduler
Mongo DB
Kafka Central Server
Adding document in applications & blobStorage collections
Stores the request sent by scheduler to deployer
Sends request to deployer for app deployment
Deployer Receives request from scheduler for app deployment
Topic: schedulerToDeployer (used to send app deployment request to deployer)
Topic: log-scheduler (used for tracking logs of scheduler)
Topic: log-workflow_validator (used for workflow validator)
App Developer
1. File Formats
2. Sensor Info
3. API Endpoint after Service Deployment
Platform UI
App.zip and deployer.json
Via API
API Gateway
deployer.json
Scheduler
Deployer
Application Controller
Application Zip file
Validator
workflow.json
Workflow Engine
Configuration files App Developer Provides
appConfig.json
Services
Workflows
appName, developerId, sensors array: contains sensor type subscribe, etc
Contains folder for services Each folder contains
1. s1.py
2. config.json
3. requiements.txt
w1.json
Sensor Types Subscribed
zip file for each service stored in Blob storage
Sensor Manager
Blob Storage

Interactions of Validator:

1. **Interaction with Sensor Manager:** Validator provides the sensor information to the Sensor manager (sensor id, sensor type, geolocation etc) upon sensor registration.
2. **Interaction with UI:** It receives app.zip from the validator via API Gateway and validates it. If some error occurs while validating it returns a response for the same.
3. **Interaction with workflow manager:** It sends all the workflow jsons to the workflow manager for the creation of script files for the workflow.
4. Interaction with monitoring and fault tolerance.

Interaction of workflow Manager:

1. **Interaction with validator:** It receives information from the validator for workflows.json.

Interactions of Scheduler with other subsystems:

1. **Interaction with Deployer:** Scheduler will select the job with the highest priority and send a request to the deployer to deploy the same on a node. Once the job is deployed it receives an acknowledgement from the deployer for the same.
2. **Interaction with Monitoring and Fault Tolerance:** Monitoring and Fault Tolerance interacts with the Scheduler and sends heartbeat messages to perform monitoring at regular intervals.

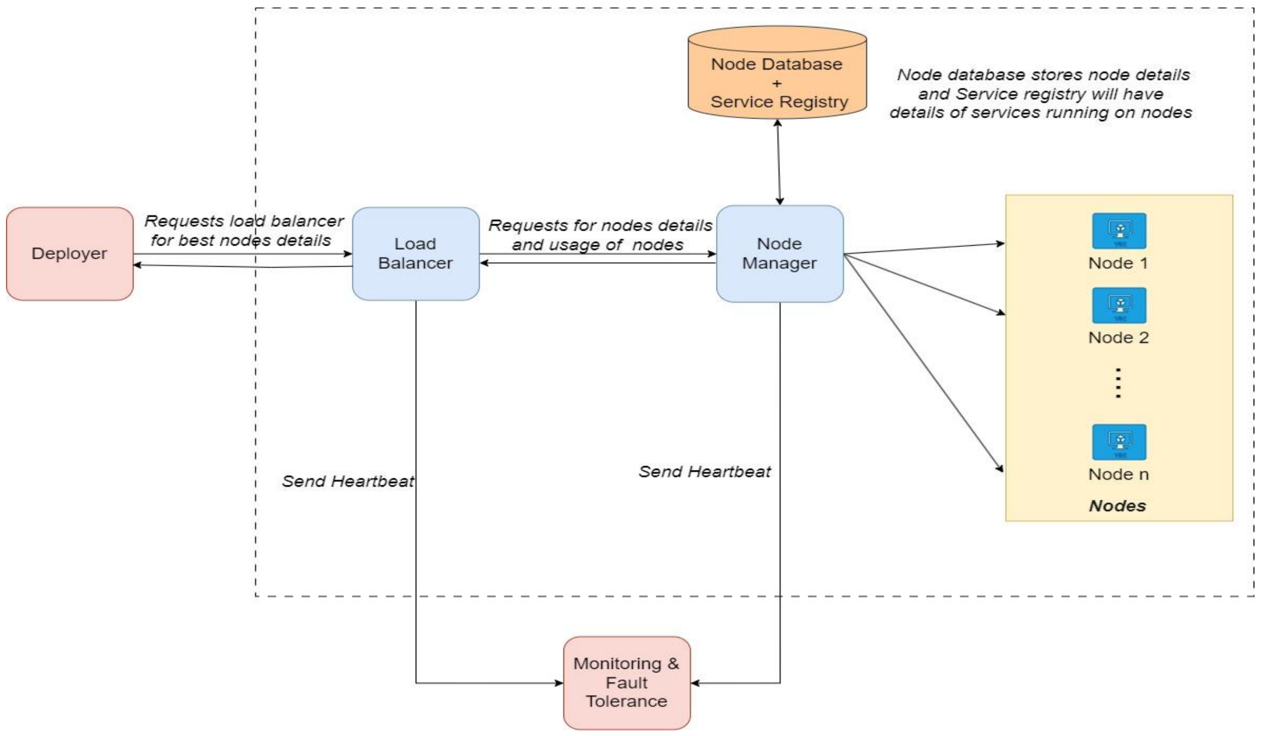
Team 3 :-

Node Manager - This module contains information about nodes running. It stores the information about all nodes in nodes DB and when load balancer asks for a node details to deploy a service it returns the details of active nodes. It also creates a new node on the request of the load balancer.

- **Submodules**
  - Nodes - Nodes are the entities where execution of an application instance or a platform component takes place. This may be a physical server or a virtual machine. The nodes may also communicate with the sensor manager for sending/receiving data to/from sensor(s).
  - Node Manager - Node manager manages the data about active nodes and provides the details load balancer whenever needed.
  - Node DB - Stores the data provided by node manager and provides the data to node manager when asked for it.
- **Functional Requirements**
  - Providing the details of active nodes to load balancer.
  - Creating a node if required.
  - Maintaining the node registry.
  - Shutdown the node if no service is running on the node.
- **Interaction between sub-modules**
  - **Node DB - Node Manager :** Node Manager stores the data about nodes in Node DB and retrieves the data from Node DB whenever needed.
  - **Node - Node Manager :** Nodes send their status to the node manager.
- **Interaction with other modules**
  - Load Balancer - It provides information about active nodes to the load balancer and creates a new node if the load balancer asks for it.
  - Monitoring and Fault tolerance - It sends heartbeats to monitoring and fault tolerance to tell that it is alive and working.

Load Balancer - This module is responsible for providing the best node to the deployer to deploy a service. It asks for node details from the node manager and selects the best node(one with the lowest node).

- **Functional Requirements**
    - Select the best node amongst the node details provided by the node manager.
    - If all nodes provided are at its full capacity then ask the node manager to create a new node and provide its details.
    - Provide the selected node to the deployer to deploy a service.
    - It will continuously check the health of containers, if it finds an overloaded container it should inform the deployer to create a new instance of that service.


Node Database
+
Service Registry
Node database stores node details and Service registry will have details of services running on nodes
Deployer
Requests load balancer for best nodes details
Load Balancer
Requests for nodes details and usage of nodes
Node Manager
Node 1
Node 2
Node n
Nodes
Send Heartbeat
Send Heartbeat
Monitoring & Fault Tolerance

<u>Node Manager and Load Balancer Diagram</u>

- **Interaction with other modules**
    - Node manager - Request the node manager to get the details of active nodes.
    - Deployer - Accept the request from the deployer to deploy a service and provide a node to the deployer to deploy the service.
    - Deployer - Load balancer sends the details of service when it finds an overload on that container(service).
    - Monitoring and Fault tolerance - It sends the heartbeat signal to monitoring and fault tolerance.

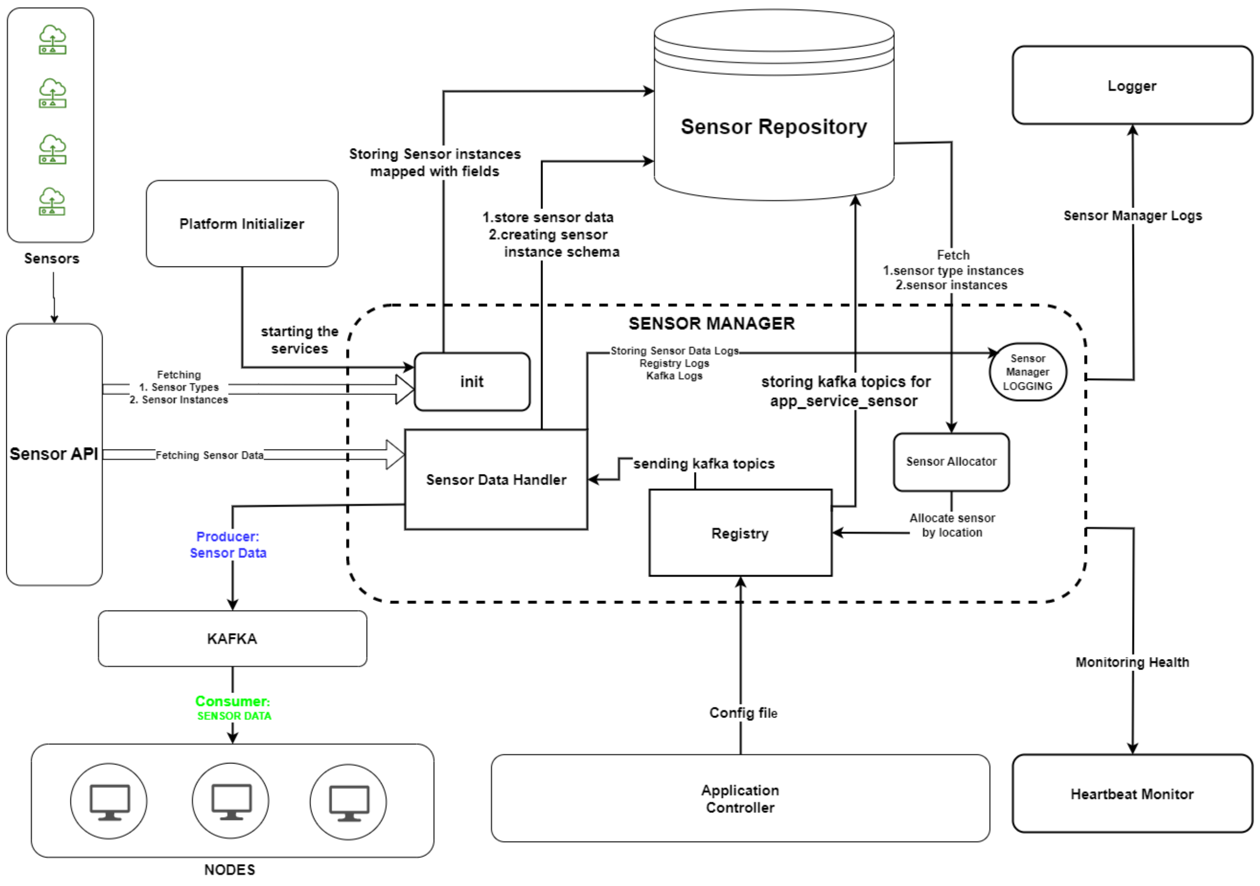
Team 4 :-

Sensor Manager: This module is designed to retrieve sensor data from OneM2M API and subsequently transmit the data to applications upon request through Kafka. Additionally, the module facilitates sensor registration for the platform by binding sensors to application services. This functionality enables the transmission of data to the application requests.

Low Level Design:

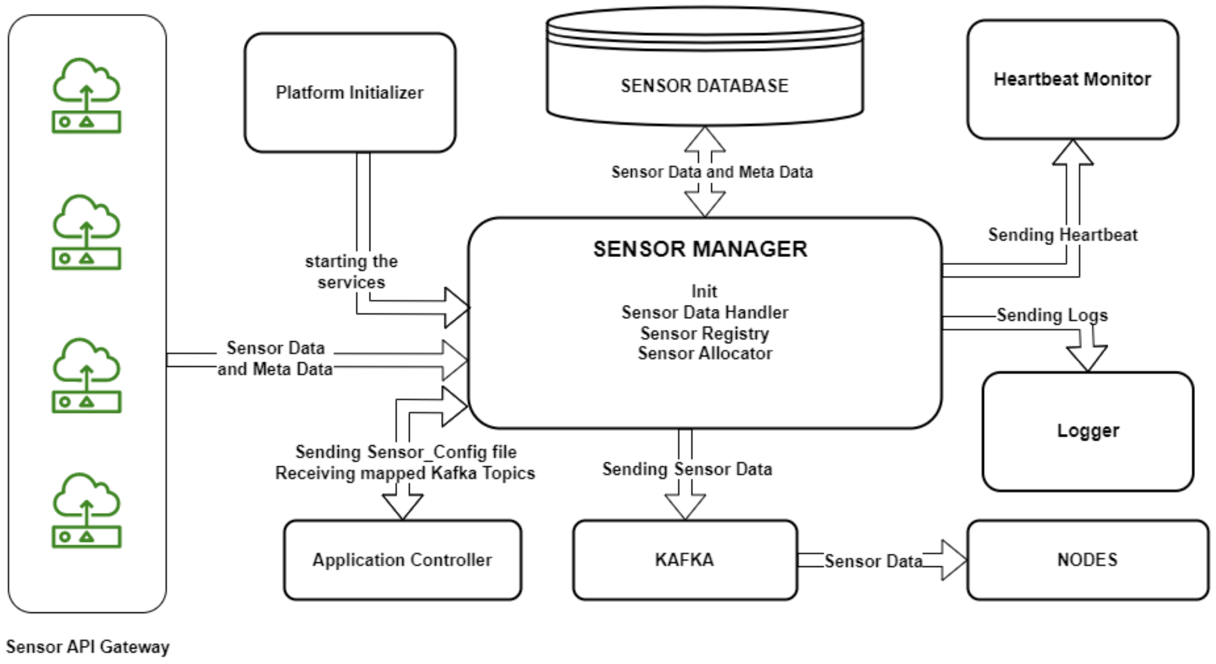
- Lifecycle:
    - Sensor Manager is started by the platform initializer module.
    - Application controller sends config files for sensors.
    - Streams sensor data using Kafka.
- Submodules:
    - Sensor Manager
        - Registers the sensors on the platform
        - Stores the instances of sensors in the sensor repository
    - Sensor metadata store
        - Databases for storing the metadata of registered sensors respectively
    - Kafka stream
        - Stream the sensor data
    - Allocator
        - Allocate sensor to application and service filtered by type and location
    - Logging
        - Log each event of sensor manager

- Interactions between submodules:



- Interactions with other modules
    - Interacts with Application Controller:
        - Allows sensor registration.
        - Allows sensor instance requests.
    - Interacts with Platform Initializer: The platform initializer gets the sensor manager service up and running.
    - Interacts with Heartbeat Monitor: Gives acknowledgement to the heartbeat monitor about its status (running/not running).

- API Endpoints:
    - registry: registers sensors as received in config file



Team 5 :-

Logging: This module makes use of kafka to extract logs and working of the platform. It is one of the most important module and is responsible for storing logs in the logging database. It stores the information under 3 categories based on the type of log detected.

Monitoring and Fault Tolerance: Monitoring and Fault Tolerance subsystem monitors each instance of every subsystem and keeps a record of active and inactive subsystems. If the module is down, it will take action accordingly. It checks for the application controller, scheduler, deployer, load balancer, node manager and other subsystems. It is also called the heartbeat of the platform due to its regular checkup and monitoring.

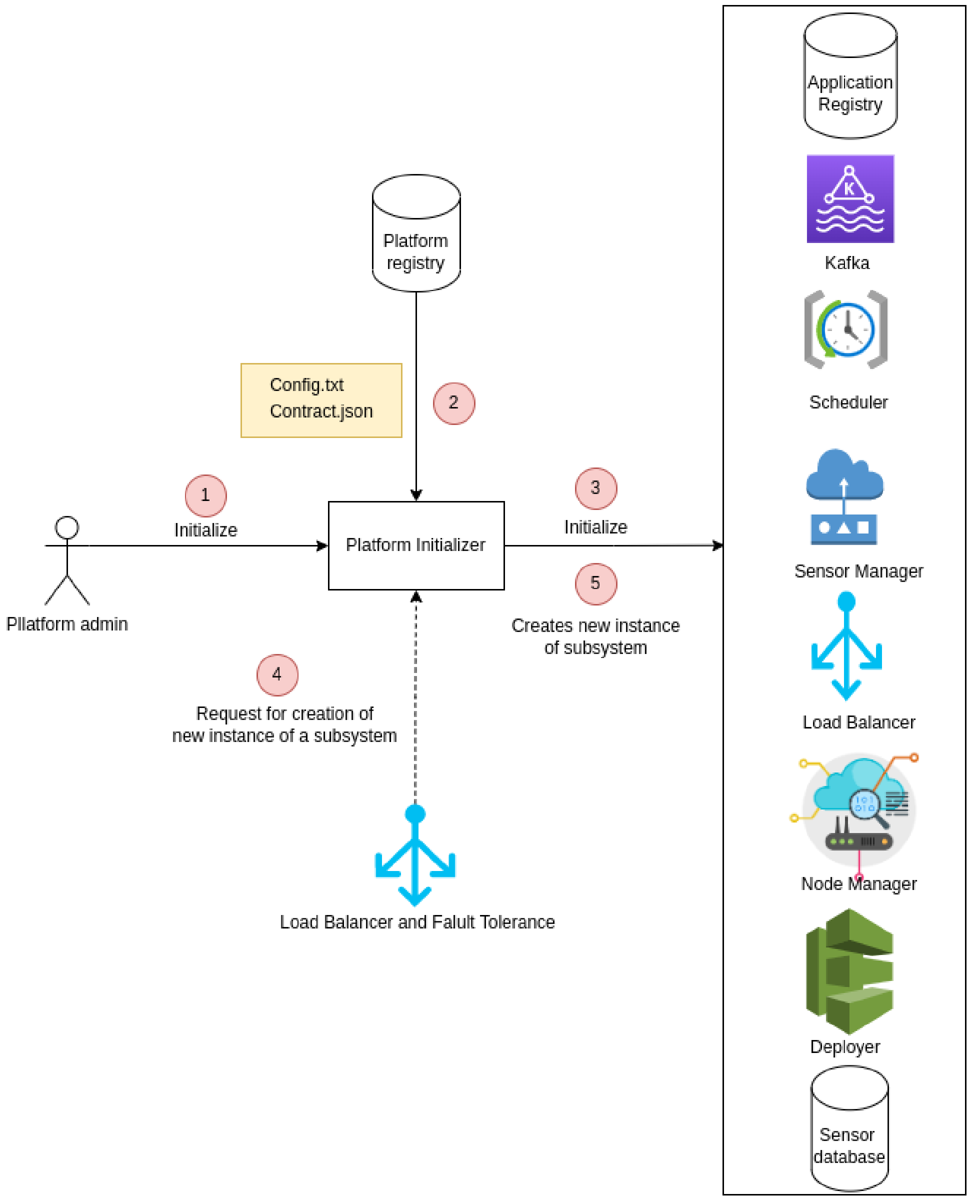
Low Level Design:

- Lifecycle:
  - Platform Initializer:
    - This subsystem is executed as the first module by the platform admin.
    - The platform initializer then extracts config and contract files from the 'platform registry'.
    - The source code of each subsystem is then procured from the platform registry.
    - The initialization of each sub system takes place.
  - Monitoring and Fault Tolerance:
    - This subsystem is initialized by the Platform Initializer.
    - It continuously receives status messages from each subsystem.
    - It keeps track of active and dead instances of a subsystem.

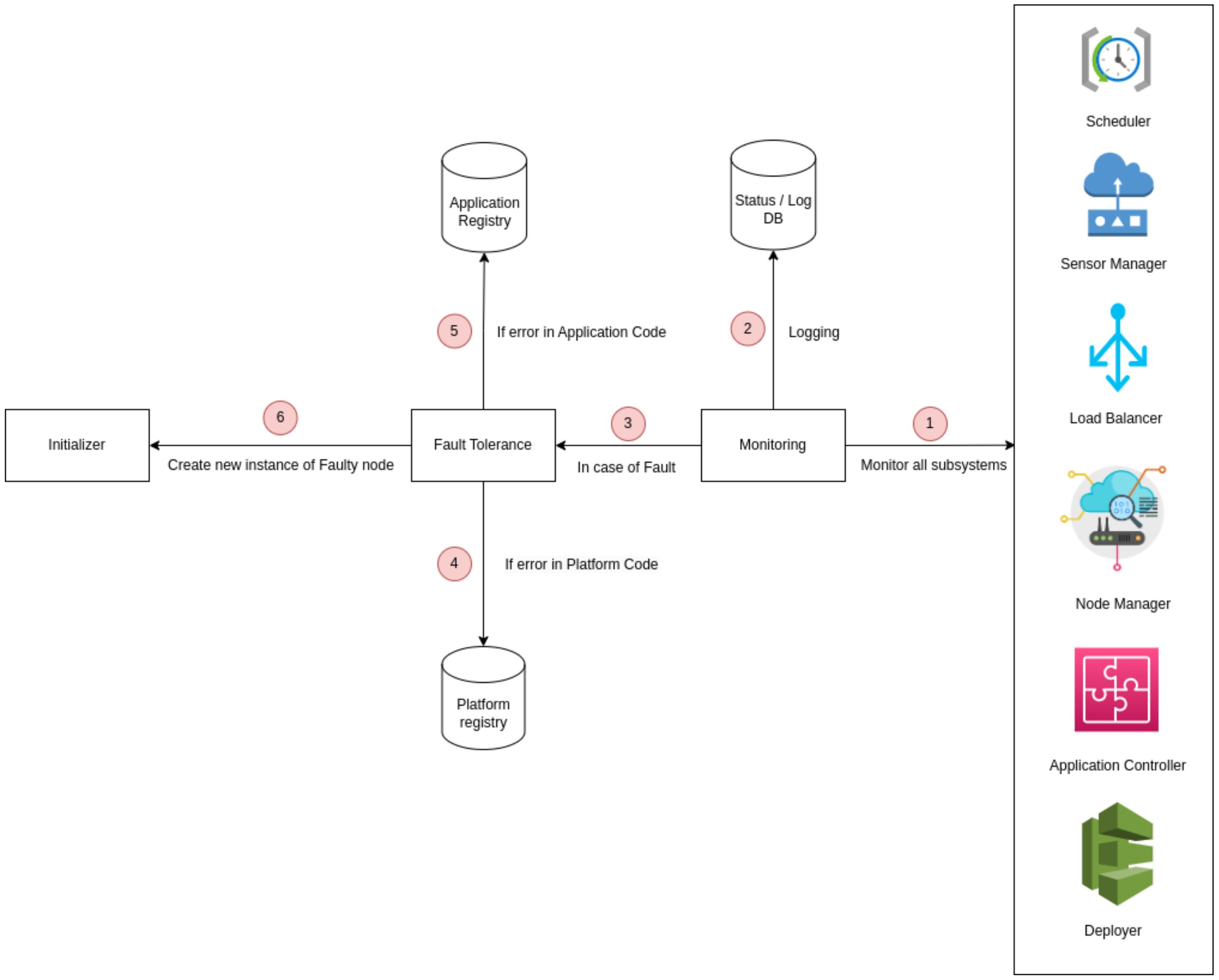
        - The status is updated in a status table in the status registry.

- Submodules:
    - Instance creator
        - It is a submodule of platform initializer executed only when there is a request for creation of a new instance of a subsystem.
        - The request can be received for two main reasons:
            - Load balancer requests for a new instance to reduce waiting period.
            - Monitoring and Fault Tolerance requests for re-initialization of a subsystem.
    - Monitor
        - It keeps track of active and dead instances of a subsystem.
    - Fault Tolerance
        - This system detects abnormality in the state of the subsystem and takes required actions.
    - Status registry
        - This is the location where the status table of active and inactive conditions are stored by the monitoring sub-module.
    - Logging registry
        - This is the location where the logs are stored and can be accessed by platform admin.

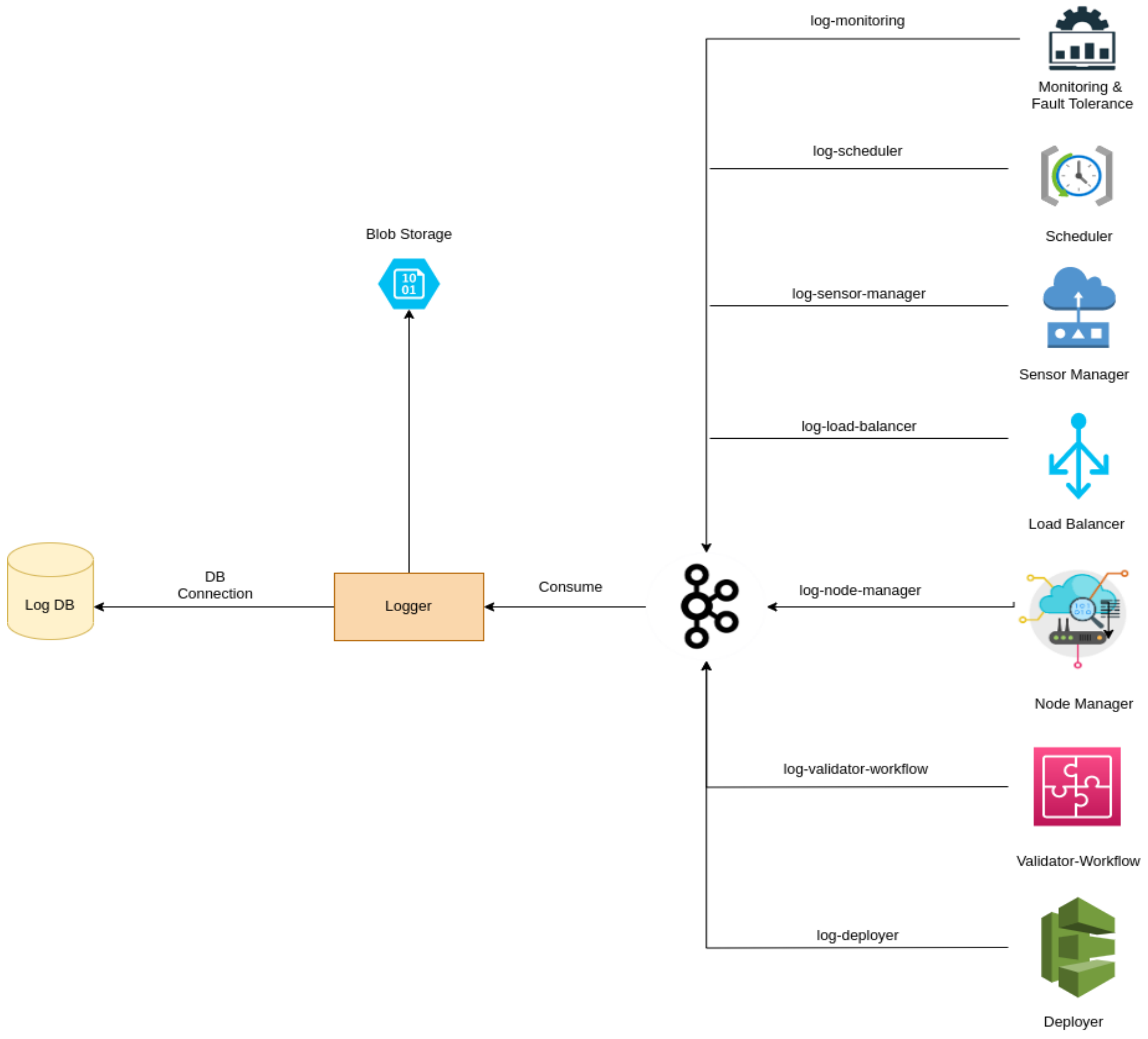
- Interactions between submodules:
  - Platform Initializer
    - This subsystem is responsible for initializing all the subsystems and hence communicating with all.
    - Killing and instance creation in case of load handling is done through communication with the load balancer.
    - Instance creation during a fault in an instance is done by the request from the Monitoring and Fault Tolerance subsystem.



- Monitoring and Fault Tolerance
  - The Monitoring and Fault Tolerance Module is responsible for continuously monitoring the status of all the subsystems mentioned in the services and hence, communicates with all the subsystems.
  - In case of an unresponsive subsystem, it communicates with the initializer to create a new instance.



- Logger
  - The responsibility of logger is to take account of every action and more importantly errors taking place in the platform.
  - This module makes the information available to the submodules and platform admin.
  - Just like monitoring and fault tolerance system, it is connected to every other module registered in this platform.



10. Bootstrap :

a) Subsystems to be initialized :

The initialiser/Bootstrapper will initialize the following subsystems :

- Kafka Server
- Sensor Manager.
- Application Controller(Validator & Workflow).
- Node Manager.
- Load Balancer.
- Deployer.
- Scheduler.
- Monitoring and Fault Tolerance
- Logger.

Each subsystem will have at least 2 instances running at all times.

b) Steps are to be followed :

Each subsystem will be run on a different Virtual Machine (VM).
Each subsystem will have its own docker image which will download its requirements into the respective container.
2 instances of the docker image of a particular subsystem will be run on a single VM.
In case of fault or requirement of load balancing, new instances can be added as and when required.

c) Different configuration files required

There will be a single config file based on which the Kafka Server, Mondo DB, Docker and other Services will be set up.

This config file will first install kafka on the server and will also contain code to configure the different topics needed by the subsystems modules :

The kafka topics needed are :

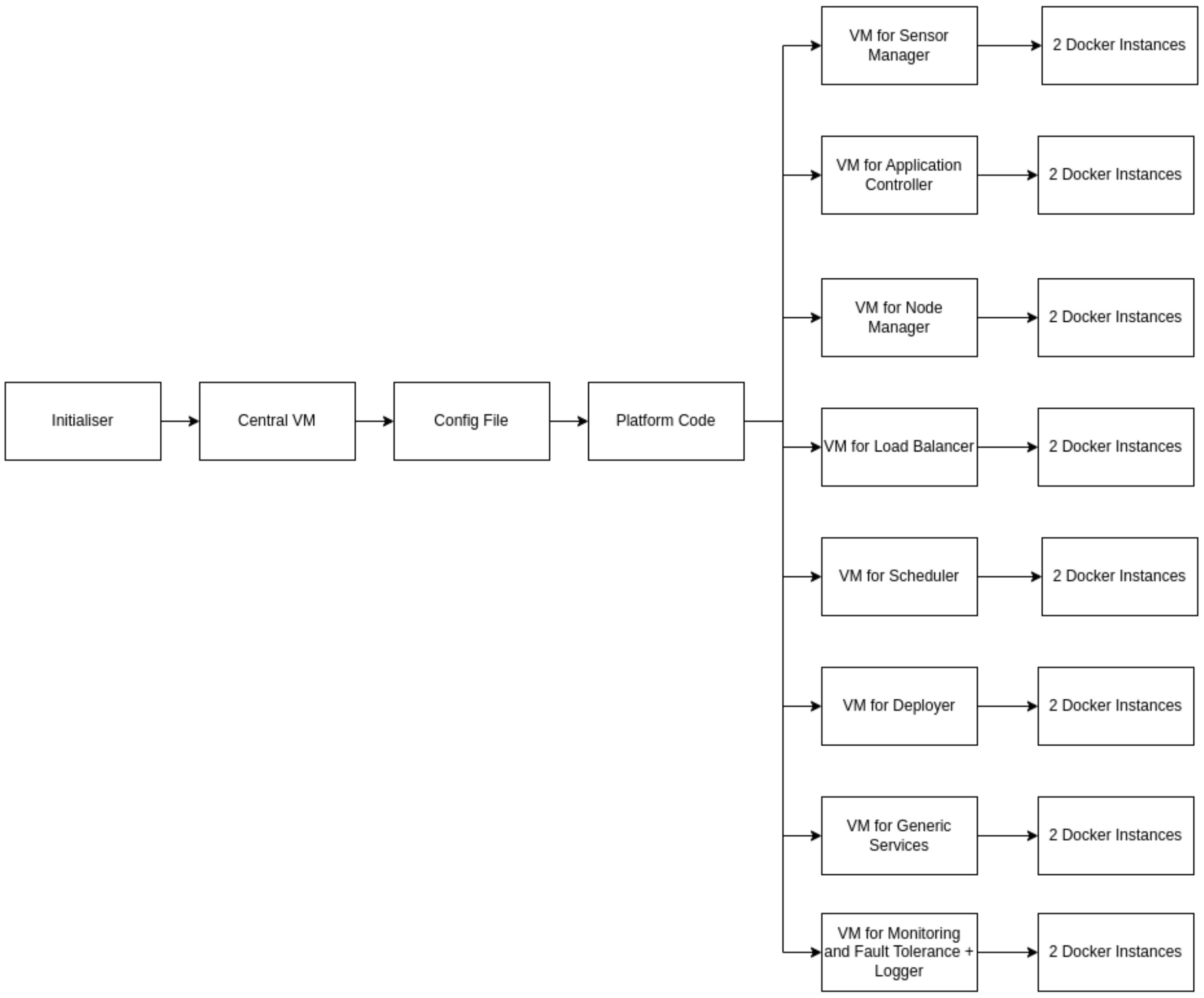
- Topic Name : Service request, having partitions = 1
- Topic Name : Request_to_scheduler, having partitions = 2
- Topic Name : heartbeatMonitoring having partitions = 2
- Topic Name : ACK, having partitions = 1
- Topic Name : Scheduler Request, having partitions = 2
- Topic Name : Node Request having partitions = 4
- Topic Name : loggingMonitoring, having partitions = 2
- Dynamic creation of Kafka topics will be performed at runtime by sensor manager, with each topic having the following characteristics: Topic Name: <App_id> Partitions: 2

For successful storage and registration, specific Mongo schemas must be present, including the following collections:

- "devices"
- "device_app_id"
- "controller_app_id"
- “Monitoring”
- “logs”

Commands required to install the docker engine will also be a part of the config file. Additional modules required by each subsystem will be present within the docker file of each subsystem and will be installed within the container by docker itself.

Certain services like Azure blob storage or AWS S3 will also need to be initialized wherein our platform and other application code would reside.


Initialiser
Central VM
Config File
Platform Code
VM for Sensor Manager
VM for Application Controller
VM for Node Manager
VM for Load Balancer
VM for Scheduler
VM for Deployer
VM for Generic Services
VM for Monitoring and Fault Tolerance + Logger
2 Docker Instances