Artifact: implementation of an adaptive flow management framework for IoT spaces

Houssam Hajj Hassan, Georgios Bouloukakis, Ajay Kattepur, Denis Conan, Djamel Belaïd

Artifact: Implementation of an Adaptive Flow Management Framework for IoT Spaces

Houssam Hajj Hassan*, Georgios Bouloukakis*, Ajay Kattepur†, Denis Conan*, Djamel Belaïd*
*SAMOVAR, Télécom SudParis, Institut Polytechnique de Paris, France
{houssam.hajj_hassan, georgios.bouloukakis, denis.conan, djamel.belaid}@telecom-sudparis.eu
†Ericsson AI Research, India, ajay.kattepur@ericsson.com

***Abstract*—This paper presents the implementation and guideline of PlanIoT [1], an adaptive flow management framework for IoT-enhanced spaces. Such spaces are composed of applications deployed at the Edge with varying QoS requirements in terms of response time, timely delivery, throughput, etc. Configuring the Edge infrastructure requires tuning multiple parameters for optimal QoS satisfaction of applications. This is a complex task especially when the system has to be re-adapted (e.g., emergency situations). The PlanIoT framework manages application data flows in an adaptive manner. This is achieved via the following core software components: (i) a queueing network composer; (ii) an automated planning modeler; and (iii) an AI planner. This artifact presents implementation details of these components as well as guidelines for using the PlanIoT framework.**

***Index Terms*—Adaptive Systems, Automated Planning, IoT, QoS, Smart Spaces**

I. Introduction

As the Internet of Things (IoT) and supporting technologies become more prevalent, spaces such as building and homes are becoming more intelligent and connected. Smart spaces usually include IoT devices that sense and produce data, applications that receive and process flows of data, a data exchange system that manages data flows between devices and applications (e.g., a message broker), and the networking infrastructure. Applications deployed in such spaces often have different Quality of Service (QoS) requirements that have to be met. For instance, a video streaming application requires a higher throughput than an energy monitoring application. Finding the optimal settings to satisfy the QoS requirements of all applications requires applying different configurations of the IoT system and tuning multiple parameters.

This task becomes even more challenging for ***Intersecting Applications*** receiving the same flows of data but having different QoS requirements; for example, a video surveillance and an evacuation planning applications require the same data flows (images), however, their QoS requirements are different (high throughput vs. timely delivery). In addition, smart spaces are usually dynamic requiring adaptation due to different circumstances (e.g. adding/removing applications, emergency situations). The PlanIoT framework [1] supports adaptive data flow management at the Edge by making sure that the QoS requirements of deployed IoT applications are satisfied. PlanIoT is based on a dataset-driven methodology that leverages automated planning to generate configuration plans for adaptive flow management.

This artifact paper presents implementation details of the PlanIoT framework [1] and provides installation and usage guidelines for end-users. The PlanIoT code is provided at https://gitlab.com/planiot/planiot-seams2023. The paper is organized as follows. Section II provides background information related to the main concepts used as well as the PlanIoT approach. Section III presents the implementation of the main PlanIoT components. Section IV provides installation guidelines for using the PlanIoT framework. Finally, we conclude the paper with a brief discussion about future works in Section V.

II. Background

This section presents background information on the key concepts behind PlanIoT: queueing network modeling (§II-A) and automated planning (§II-B). We then provide an overview of the PlanIoT approach (§II-C).

A. Queueing Networks

PlanIoT relies on Open Queueing Networks [2], [3] as the mathematical foundation for modeling the performance of data flows in IoT systems. Queueing networks can be automatically created and simulated using open-source queueing simulators, such as Java Modelling Tools (JMT) [3], [4]. We develop a generic *QoS Model* that represents a publish/subscribe-based IoT system. In this model, queueing stations are used to represent applications/devices, the message broker, and the network infrastructure. The parameters of the model (e.g., queues' service rates, routing strategies) enable simulating different configurations of an IoT system. For example, the available networking resources between the message broker and the IoT applications are used to estimate the service rate of queueing stations that represent the network infrastructure. Data flow management techniques such as priorities, drop rates, and resource allocation policies can also be simulated using this model. A more detailed description of the *QoS Model* can be found in [1].

B. Automated Planning

Automated Planning (AI Planning) deals with the process of determining the best sequence of actions to achieve a specific goal or set of goals. PlanIoT uses automated planning as a decision-making tool for finding the optimal configuration of an IoT system that best satisfies the QoS requirements of

applications. AI planning is also used to enable re-adaptation in response to runtime changes in the IoT system. To express planning models, we use the Planning Domain Definition Language (PDDL) [5], [6], an action centered language that provides a standard syntax to describe actions by their parameters, preconditions, and effects. PDDL divides the definition of a planning problem into two parts: the *domain* and the *problem* [7]. The *domain* file contains a description of the actions that can be taken by the planner; in our case, the actions represent configurations of the IoT system. The *problem* file includes a description of the initial state of the system, as well as the desired goal state to be achieved (e.g., to satisfy the QoS requirements of IoT applications). Algorithms and techniques such as search-based or reasoning-based planning are used by the planner to find an optimal plan given the domain and problem descriptions provided. We provide more details about the planning process in §III-D.

C. The PlanIoT Approach

PlanIoT [1] uses a dataset that contains performance metrics of data flowing at the Edge of IoT systems under different parameters settings. To generate this dataset, the PlanIoT generic *QoS Model* is instantiated based on the system specifications provided by the user. For example, for the same smart space description, multiple models can be generated with different configuration parameters (e.g., varying network resources, prioritization techniques). After simulating these models, the dataset that captures the performance of the IoT system under different situations is obtained. This dataset is further used to generate the domain and problem files used as input by the AI planner. The planner then provides the best *configuration plans* of the IoT system given the domain and problem file descriptions. At runtime, when changes in the Edge infrastructure occur (e.g., the IoT system becomes overloaded), the planner can be triggered to adapt the Edge infrastructure and provide a (potentially) better configuration plan of the system.

III. Implementation

Fig. 1 depicts the process for generating configuration plans that enable the adaptive management of IoT data flows at the Edge. PlanIoT relies on two main components: (i) the *Queueing Network Composer*, which instantiates the generic *QoS Model* and composes a queueing network to evaluate the performance of the IoT system, and (ii) the *AI Planner*, which uses the generated dataset to find the optimal configuration of the IoT system that satisfies the QoS requirements of the deployed applications. We present next the implementation details of the components shown in Fig. 1. We start by describing how the IoT system specification file is defined (§III-A), then show the process to compose and simulate the queueing model (§III-B). We show how we generate a performance metrics dataset (§III-C) from the simulations, and how the dataset can be used to generate the PDDL *domain* and *problem* files required by the AI planner (§III-D) .

A. IoT System Representation

JSON files are leveraged to define the *IoT system specification* representing in a structured way the overall Edge infrastructure of a smart space. The specification includes information related to the (i) *IoT devices*: average message sizes, data frequency (e.g., messages/sec), the topics they publish to, etc; (ii) *IoT applications*: applications deployed, their category, their topic-based subscription filters; and the (iii) *Edge broker*: the available network resources and the system capacity. Note that multiple IoT system specifications can be defined for representing multiple situations (e.g., emergency cases).

Listing 1 shows how to define IoT devices in the IoT space specification file. Each device is a JSON object identified by a `deviceId` that contains a `publishFrequency` integer that represents the frequency at which the device sends messages (in messages/second), a `messageSize` integer that represents the average size of the messages sent by the device (in bytes), and a list of topics that the device publishes to. Moreover, we assume that devices can publish messages based on a probability distribution (e.g. normal, exponential, etc.).

```
{
 "iotDevices": [
{
      "deviceId": "temp_r324",
      "deviceName": "temperature_sensor",
      "publishFrequency":5,
      "messageSize":200,
      "publishesTo":["temperature_r324"],
      "distribution":"exponential"
},
...
```

Listing 1. IoT Devices definition

Similarly, we define applications deployed in the smart space as shown in Listing 2. Each application belongs to an `application category` and is assigned a `priority`. Note that a lower integer represents a higher priority. Applications can subscribe to one or more topics, which are defined as a list in the `subscribesTo` field.

```
{
 "applications": [
{
    "applicationId": "app1",
    "applicationName": "dashboard",
    "applicationCategory" :"AN",
    "priority": 0,
    "subscribesTo": ["temperature_r324", "smoke_r324"]
},
...
```

Listing 2. Applications definition

Finally, the IoT system specification includes parameters related to the network infrastructure and to the Edge broker (Listing 3). These parameters include the available bandwidth between the broker and the applications (in Mbps), the bandwidth allocation policy used (e.g., max-min [8]), the priority policy followed (prioritizing applications vs prioritizing topics), the drop rates assigned for every application category (in percent), and the capacity of the message broker (in messages).

Note that standard ways of representing IoT systems can be used to define the IoT system specification. Our ongoing work includes modeling IoT-enhanced spaces using the Next

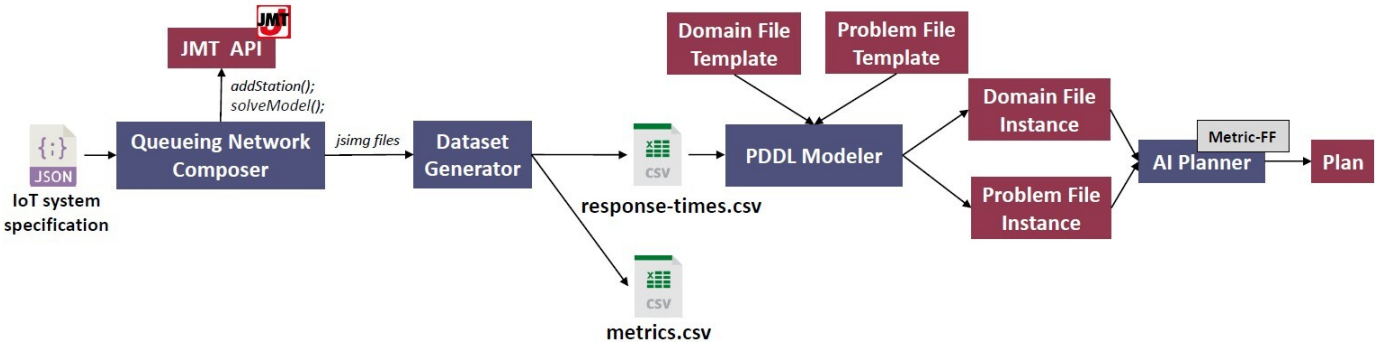

Fig. 1. Plan generation process

Generation Service Interfaces-Linked Data (NGSI-LD) [9] standard. These data models can be used to provide the IoT system specification and subsequently fed as input to the *Queueing Network Composer* component described next.

```
"systemBandwidth": 70,
"bandwidthPolicy": "default",
"priorityPolicy": "apps",
"dropRateAN": 0,
"dropRateRT": 0,
"dropRateTS": 0,
"dropRateVS": 0,
"brokerCapacity": 10000
```

Listing 3. IoT system parameters

We use a `csv` file to define application requirements in terms of end-to-end response time (in seconds), throughput (in Kbps), and drop rate (in %), as shown in Table I.

category	response_time	throughput	drop_rate
analytics	10	0	0.1
realtime	0.4	28.8	0.02
transactional	4	0	0
videostreaming	2	202	0.2

TABLE I
APPLICATION REQUIREMENTS AS A CSV FILE

B. Queueing Network Composition

Once the IoT system specification file is defined, the **Queueing Network Composer** generates the corresponding queueing network that evaluates the performance of data flowing between Edge devices and IoT applications via a message broker. This is implemented by relying on the JMT open-source queueing simulator [3], [4]. JMT is a suite of applications that offer a comprehensive framework for system modelling and evaluation using analytical/simulation techniques. While JMT JSIMgraph provides a graphical user interface to design queueing models, we use JMT's API to compose and run the simulations.

The Queueing Network Composer parses the JSON file, and composes the queueing network by using the JMT API. This composition is done based on the *QoS Model* generated from the IoT system specification. The Queueing Network Composer first instantiates a *JMT Common Model*. This model holds all the elements of the queueing network and saves them in a `jsimg` file (more details later). Then, the `addStation` function is used to add sources, queues, forks and joins, and sinks. To create *QoS Models* that represent different configurations of an IoT system, we can prioritize some of the subscription flows. This is done by using the `setStationQueueStrategy` method that enables defining priority queues. After adding all the components, connections between stations are created by calling the `setConnection` method. Data flowing from IoT sources to the data recipients are configured by setting the appropriate routing strategies using the `setRoutingStrategy` method. Finally, the `setServiceTimeDistribution` method is used to set the service time of the network queue based on network resources allocation.

The JMT composed model is provided as an XML-based file (`jsimg` file) that contains the topology of the queueing network and the simulation parameters. Once all the elements are added to the network, we save the model as a `jsimg` file by calling JMT's *XMLWriter*. We then call JMT's dispatcher to solve the simulation model. The dispatcher runs the simulation and saves the simulation results in another `jsimg` file.

C. Dataset Generation

As shown in Listing 4, JMT provides the results of the simulations in XML files. These files are not straightforward to use, especially for users who are not familiar with JMT. Hence, we implement a **Dataset Generator** that is responsible for extracting the results of the simulations from JMT's `jsimg` files and creating the dataset needed by PlanIoT. We save the simulation results as a set of `csv` files. For each simulation, we extract results related to the response time, throughput, and drop rate for each subscription. We save these results in the format shown in Table II. In addition, we create a `response-times.csv` file that holds the value of the response time for each subscription (mapping a data flow) under different configurations of the Edge broker, as shown in Table III. This file is then used to generate the domain and problem files fed as input for the AI planner (see next section).

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<solutions xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       logDecimalSeparator="." logDelimiter="," logPath="your_local
_path\JMT\" modelDefinitionPath="." modelName="default.jsimg" solutionMethod="
       simulation" xsi:noNamespaceSchemaLocation="SIMmodeloutput.xsd">
   <measure alfa="0.05" analyzedSamples="8194" class="" discardedSamples="65"
         logFile="your_local
_path/JMT/networkQueue_Utilization.csv" lowerLimit="0.40003963524075425"
       maxSamples="1000000" meanValue="0.42805882812360384" measureType="
       Utilization" nodeType="station" precision="0.05" station="outputQueue"
       successful="true" upperLimit="0.4560780210064534"/>
...
```

Listing 4. JMT-generated results file

topic	app	response_time	throughput	drop_rate
topic10	app11	0.6927146474811501	0.003779867027741613	0.0
topic10	app9	0.6996978145171252	0.003779867027741613	0.0
topic11	app12	0.7982395522950397	0.1172509965121246	0.0
topic11	app15	0.799657539567181	0.1172509965121246	0.0
topic11	app8	0.7977678454742368	0.1172509965121246	0.0
topic12	app10	0.7950121622847551	0.003110639907232652	0.0

TABLE II
JMT SIMULATION RESULTS AS A CSV FILE

topic	app	default	prioRT	prioVS	prioRTVSTSAN	dropVS1	dropVS2AN2	...
topic1	app1	0.899	1.224	1.136	0.036	0.886	0.927	...
topic10	app11	0.692	0.773	0.854	0.0162	0.684	0.661	...
topic10	app9	0.699	0.756	0.855	0.015	0.671	0.658	...
topic11	app12	0.798	0.977	1.084	0.024	0.789	0.776	...
topic12	app9	0.800	0.882	0.980	0.017	0.802	0.798	...
topic13	app10	0.792	0.890	0.978	0.0197	0.789	0.795	...

TABLE III
RESPONSE-TIMES.CSV FILE

D. Automated Planning for IoT Flow Management

PlanIoT uses automated planning to provide an optimal flow-handling configuration at the Edge broker. For this purpose, we employ PDDL [5], [6] to provide plans that consist of two descriptions: (i) the *domain* description that decouples the parameters of actions from specific objects, initial conditions, and goals, and (ii) the *problem* description that instantiates a grounded problem with objects, initial conditions, goals and cost functions (metrics). To enable the composition of concrete domain and problem files, we use *templates* of domain and problem files that are instantiated using the `response-times.csv` file.

Listing 5 shows how we define the actions in the *template* domain file. Each action represents a configuration of the IoT system that can be chosen by the planner. For example, the `action` at Line 1 indicates a configuration of the IoT system where RT (realtime) applications are prioritized. Similarly, the `action` at Line 12 represents a configuration where a drop rate of 1% is applied to VS (video streaming) flows. The pre-conditions of these actions include the QoS not being met and the priorities not set. The effects of these actions accurately map the required QoS to flows. Note how the effects of the actions include a string of the form `#configuration_effects#`.

The **PDDL Modeler** component is responsible for instantiating the domain and problem files by replacing the `#configuration_effects#` strings by the value of the response time for each flow, as shown in Listing 6 (e.g., Lines 9–14). This is done by parsing the `response-times.csv` file and filling in the corresponding value of the response time for each subscription under the different configurations.

```
(:action prioritize_RT
   :parameters (?t - topic ?app - application )
   :precondition (and
         (priority_not_set ?t ?app )
   )
   :effect (and
      priority_not_set ?t ?app ))
      (priority_set ?t ?app )
      #prioRT_effects#
   )
)
(:action droppingVS1
   :parameters (?t - topic ?app - application )
   :precondition (and
         (baseline ?t ?app )
   )
   :effect (and
      (not (baseline ?t ?app ))
      (QoS_achieved ?t ?app )
     #dropVS1_effects#
   )
)
```

Listing 5. PDDL domain file template

```
(:action prioritize_RT
   :parameters (?t - topic ?app - application )
   :precondition (and
         (priority_not_set ?t ?app )
   )
   :effect (and
      (not (priority_not_set ?t ?app ))
      (priority_set ?t ?app )
      (increase (latency topic10 app11) 0.77)
      (increase (latency topic10 app9) 0.76)
      (increase (latency topic11 app12) 0.98)
      (increase (latency topic11 app15) 0.98)
      (increase (latency topic11 app8) 0.36)
      (increase (latency topic12 app10) 0.87)
      ...
)
(:action droppingVS1
   :parameters (?t - topic ?app - application )
   :precondition (and
         (baseline ?t ?app )
   )
   :effect (and
      (not (baseline ?t ?app ))
      (QoS_achieved ?t ?app )
      (increase (latency topic10 app11) 0.68)
      (increase (latency topic10 app9) 0.67)
      (increase (latency topic11 app12) 0.79)
      (increase (latency topic11 app15) 0.79)
      (increase (latency topic11 app8) 0.79)
      (increase (latency topic12 app10) 0.8)
      ...
)
```

Listing 6. PDDL domain file instance

In the PDDL problem file template (Listing 7), we instantiate objects (topics and applications) and specify the initial state of our system. Since the problem file template is instantiated by the PDDL Modeler, the topics and applications deployed in the IoT system are not known beforehand. The PDDL Modeler replaces the strings `#topics#` and `#apps#` by the corresponding topics and applications deployed in the smart space. The initial state of the system includes initializing the `responsetime` function to 0, with each action providing increments to the `responsetime` function (Line 10). Our

goal is to transmit all data flows while minimizing the total time for transmission. This is why the optimization metric (Line 16) focuses on minimizing the weighted end-to-end response time of flows. Therefore, the AI planner searches for the actions that introduce minimum end-to-end response time for all flows. Note that the metric expression can easily be manipulated to give higher importance (i.e., weights) to some flows. An example of an instantiated problem file is provided in Listing 8.

```
1  (:define (problem problem_name) (:domain domain_name)
2  (:objects
3     #topics# topic_all - Topic
4     #apps# app_all - Application
5  )
6  (:init
7        (baseline topic_all app_all)
8        (priority_not_set topic_all app_all)
9
10       #init_predicates#
11 )
12 (:goal (and
13       (QoS_achieved topic_all app_all)
14       (priority_set topic_all app_all)
15 ))
16 (:metric minimize #metric#
17 )
```

Listing 7. PDDL problem file template

```
1  (define (problem problem_name) (:domain domain_name)
2  (:objects
3     topic1 topic2 topic3 topic4 topic5 topic6 topic7
          topic8 topic9 topic10 ... topic_all - Topic
4     app1 app2 app3 app4 app5 ... app_all - Application
5  )
6  (:init
7        (baseline topic_all app_all)
8        (priority_not_set topic_all app_all)
9        (= (latency topic10 app11) 0)
10       (= (latency topic10 app9) 0)
11       (= (latency topic11 app12) 0)
12       (= (latency topic11 app15) 0)
13       ...
14 (:goal (and
15       (QoS_achieved topic_all app_all)
16       (priority_set topic_all app_all)
17 ))
18
19 (:metric minimize ( + ( + ( + (+
20       (* 1 (latency topic2 app7 ))
21       (* 1 (latency topic2 app1 )))
22       (* 1 (latency topic5 app14 )))
23       (* 1 (latency topic25 app7 )))
24       (* 1 (latency topic8 app6 )))
25 ...
26 )
```

Listing 8. PDDL problem file instance

IV. Using PlanIoT

In this section, we provide details about the installation process and guidelines to run PlanIoT. The PlanIoT code and the full list of dependencies, as well as guidelines for how to use it are also provided in https://gitlab.com/planiot/planiot-seams2023.

A. Setting up PlanIoT

We recommend using a GNU/Linux system with the X11 system and with Docker[1] installed. We provide a Docker container using Ubuntu version 20.04 with the Metric-FF[2] version 2.0 planner [10] and JMT[3] version 1.2.2 installed. After cloning the PlanIoT repository at https://gitlab.com/planiot/planiot-seams2023, we build and run the Docker container provided. Instructions for building and running the container can be found in the repository. After running the container, we start using PlanIoT. The container includes a `planiot` directory containing the following folders:

- `Code/PlanIoT-SEAMS2023`: contains the code for the Queueing Network Composer and the Dataset Generator components.
- `Scenarios`: contains IoT system specifications, datasets, and domain and problem templates and files used in the *Experimental Results* section of [1].
- `Scripts`: contains scripts needed to run the Queueing Network Composer, Dataset Generator, PDDL Modeler, and the AI Planner.

Fig. 2 shows the workflow to generate plans for managing data flows in IoT-enhanced spaces. To demonstrate how PlanIoT works, we choose to instantiate the process to generate an optimal configuration plan for the medium-loaded system described in [1]. This IoT system is composed of 16 applications that fall into the AN (*analytics*), RT (*realtime*), TS (*transactional*), and VS (*video streaming*) categories. The applications receive data from 30 topics published to the message broker. We use this example to showcase how to create the performance metrics dataset (§IV-B) and how to generate plans for IoT flows configuration using the AI planner (§IV-C). The example follows the flow ❶a → ❷ → ❸ → ❹ → ❺ → ❻a.

B. Generating Performance Metrics Datasets

The files needed to conduct the experiment are located in directory `Scenarios/medium-load`. Step ❶a entails defining the IoT system specification file following the format described in §III-A. The sub-directory `system-specifications` contains `json` files that represent the IoT system under different configurations, such as applying priorities to certain application categories (e.g., `prioRT.json`, `prioAN.json`), applying some dropping rates for loss-tolerant applications (e.g., `dropVS1`), or applying resource allocation policies (e.g., `maxmin.json`). To generate the performance metrics dataset, we need to run simulations for the *default* configuration, as well as other (possibly) optimized configurations. To do this, we run the script `run_simulation.sh` located in the `Scripts` directory. The script calls the Queueing Network Composer to compose and launch the JMT simulation (Step ❷), and calls the Dataset Generator to extract the results of the simulation and add them to the dataset (Step ❸). The script takes as input the IoT space specification (e.g., `default.json`), the `response-times.csv` file, the duration of the simulation (in seconds), and an alias that references the simulation. We recommend running the simulation for 5

[1]https://docs.docker.com/get-docker/

[2]https://fai.cs.uni-saarland.de/hoffmann/metric-ff.html

[3]https://jmt.sourceforge.net/

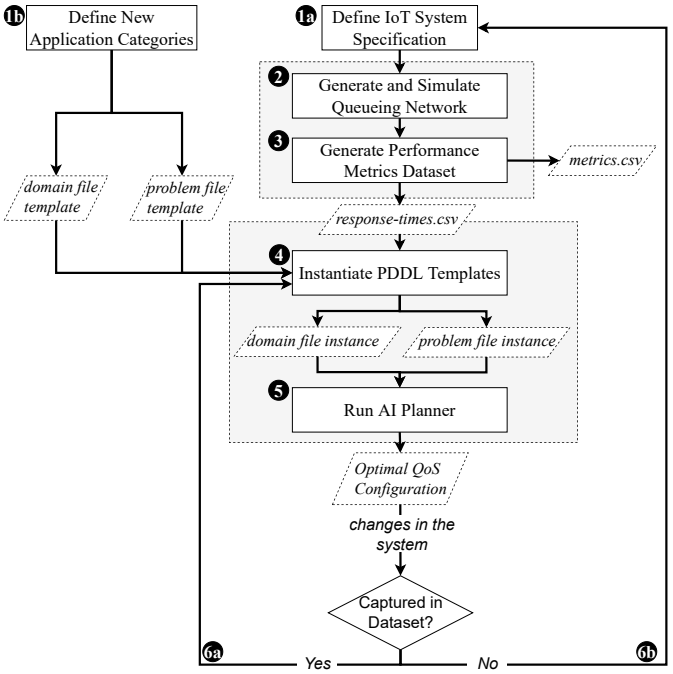

Fig. 2. PlanIoT workflow.

minutes to reach a 95% confidence interval. We launch the script from root directory `planiot`:

```
$ pwd
/home/planiot/planiot
$ Scripts/run_simulation.sh \
Scenarios/medium-load/system-specifications/default.json \
Scenarios/medium-load/dataset/response-times.csv \
300 default
```

The metrics related to response time, throughput, and drop rates are saved in the `dataset` directory under the name `metrics_alias.csv`, where *alias* is the alias we chose for the simulation when running the `run_simulation.sh` script. To create the dataset found in the `dataset` directory, we have run simulations using all the system specification configurations contained in the `system-specifications` folder.

C. Generating Plans for IoT Flows Configuration

Next, we instantiate the PDDL *template* domain and problem files by relying on the `dataset` created (Step 4). We do this by running the `InstantiatePddlTemplates.py` script:

```
$ python Scripts/InstantiatePddlTemplates.py \
Scenarios/medium-load/dataset/response-times.csv \
Scenarios/medium-load/pddl-templates/domain-template.pddl \
Scenarios/medium-load/pddl-templates/problem-template.pddl
```

The script creates the domain and problem files needed to run the planner. The generated files are saved in the `pddl-files` directory. Note that the script takes an additional argument when using templates for overloaded systems (`-o`) and emergency situations (`-e`). Such templates contain actions for increased dropping for loss-tolerant applications. We then use the generated domain and problem files to run the planner (Step 5). This can be done by running the `run_planner.sh` script that calls the Metric-FF planner:

```
$ Scripts/run_planner.sh \
Scenarios/medium-load/pddl-files/domain-generated.pddl \
Scenarios/medium-load/pddl-files/problem-generated.pddl \
Scenarios/medium-load/plans/solution.pddl
```

The script takes as arguments the domain file, the problem file, and the path of the file where the output of the planner is stored.

In the case of changes in the IoT system, such as an emergency situation (Step 6a), runtime adaptation can be performed by regenerating the domain and problem files by running the `InstantiatePddlTemplates.py` with the corresponding option that corresponds to the change (i.e., `-e`). We can then regenerate an adaptation plan using the `run_planner.sh` script.

V. Conclusion

This artifact paper presents implementation details and usage guidelines of the PlanIoT framework [1]. PlanIoT can be used to (re)adapt data flows of Edge infrastructures in IoT-enhanced spaces by providing optimal plans for satisfying QoS requirements of deployed applications. PlanIoT uses a Queueing Network Composer to generate a dataset that captures the performance of the IoT system. A set of automated planning components are then responsible for choosing an optimal configuration plan for the adaptive management of IoT flows at the Edge server. This artifact paper presents how PlanIoT can be used to generate plans for enabling adaptive IoT systems.

References

[1] H. Hajj Hassan, G. Bouloukakis, A. Kattepur, D. Conan, and D. Belaïd, "PlanIoT: A Framework for Adaptive Data Flow Management in IoT-enhanced Spaces," in *IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2023.

[2] J. Shortle, J. Thompson, D. Gross, and C. Harris, *Fundamentals of queueing theory*. John Wiley & Sons, 2018, vol. 399.

[3] M. Bertoli, G. Casale, and G. Serazzi, "Jmt: performance engineering tools for system modeling," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 4, pp. 10–15, 2009.

[4] ——, "An overview of the jmt queueing network simulator," *Politecnico di Milano-DEI, Tech. Rep. TR*, vol. 2007, 2007.

[5] M. Fox and D. Long, "PDDL 2.1: An Extension to PDDL for Expressing Temporal Planning Domains," *Journal of Artificial Intelligence Research*, vol. 20, pp. 61–124, 2003.

[6] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning: Theory and Practice*. Elsevier, 2004.

[7] P. Haslum, N. Lipovetzky, D. Magazzeni, and C. Muise, "An introduction to the planning domain definition language," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 13, no. 2, pp. 1–187, 2019.

[8] D. Pan and Y. Yang, "Max-min fair bandwidth allocation algorithms for packet switches," in *2007 IEEE International Parallel and Distributed Processing Symposium*, 2007, pp. 1–10.

[9] "Context Information Management (CIM) NGSI-LD API V1.4.2."

[10] J. Hoffmann, "Extending ff to numerical state variables," in *ECAI*, vol. 2. Citeseer, 2002, pp. 571–575.