Title:	*Trustworthy & agile operation of smart IoT systems – First version*
Authors:	*Nicolas Ferry (SINTEF), Stéphane Lavirotte (CNRS), Wissam Mallouli (Montimage), Andreas Metzger (UDE), Edgardo Montes de Oca (Montimage), Phu Nguyen (SINTEF), Alexander Palm (UDE), Diego Rivera (Montimage), Jean-Yves Tigli (CNRS).*
Editors:	*Alexander Palm (UDE), Andreas Metzger (UDE)*
Reviewers:	*Arnor Solberg (TellU), Modris Greitans (EDI)*
Identifier:	*Deliverable # D3.2*
Nature:	*Other*
Date:	*01 July 2019*
Status:	*v1.0*
Diss. level:	*Public*

Executive Summary

Deliverable D3.2 focuses on providing the first version of the different enablers of WP3 (namely the Online-Learning Enabler, Context Monitoring and Behavioural Drift Analysis Enabler and Root Cause Analysis Enabler as well as the Adaptation Enactment as support for Self-Adaptation). This document complements the deliverable of type OTHER by explaining the solutions that were developed in WP3. This includes the conceptual design of the solutions, as well as the technical details for the different enablers. The executable code of the enabler implementations is provided in the ENACT online repository (https://gitlab.com/enact).

The research leading to these results has received funding from the European Community's H2020 Programme under grant agreement n° 780351 (ENACT).

In particular, for the online learning enabler, a 1st prototypical implementation of the enabler was provided by employing policy-based reinforcement learning as the underlying learning paradigm. A validation and demonstration of the enabler was done using the benchmark system Brownout RUBiS.

For the context monitoring & behavioural drift analysis enabler, a first implementation of this enabler provides: (1) A tool for collecting contextual information and providing the streaming data for behavioural drift computation; (2) A first behavioural drift analyser (BDA) for apprehending the difference between expected and observed behaviour of the system. This first BDA is based on Gaussian Mixture Model of the observed behaviour.

For the root cause analysis enabler, the implementation will include a preliminary database of anomalies (at least 3 patterns), the agent plugins for at least 2 agents and the system graph construction algorithm. I will be available during the following months.

For the adaptation enactment (GeneSIS execution engine) enabler, the initial implementation of the orchestration and deployment enabler (modelling language and execution environment), supporting deployment over IoT, Edge and Cloud infrastructure is available. This enabler (aka. GeneSIS) provide initial support for the dynamic adaptation of the deployment of a SIS. A first version of its models@run.time engine is implemented and the GeneSIS execution engine expose a set of API for third-parties to trigger an adaptation.

The research leading to these results has received funding from the European Community's H2020 Programme under grant agreement n° 780351 (ENACT).

Members of the ENACT consortium:

SINTEF AS	Norway
BEAWRE	Spain
MONTIMAGE	France
EVIDIAN SA	France
INDRA Sistemas SA	Spain
FundacionTecnalia Research & Innovation	Spain
TellU AS	Norway
Centre National de la Recherche Scientifique	France
Universitaet Duisburg-Essen	Germany
Istituto per Servizi di Ricovero e Assistenza agli Anziani	Italy
Baltic Open Solution Center	Latvia
Elektronikas un Datorzinatnu Instituts	Latvia

Revision history

Date	Version	Author	Comments
12/02/2019	V0.1	Alexander Palm (UDE)	Table of contents and document structure
21/05/2019	V0.5	All partners	Draft version of enabler sections provided
31/05/19	V0.7	All partners	Update based on WP3 internal feedback
12/06/19	V0.9	All partners	Refinement and completion for project-internal review
30/06/19	V0.99	Alexander Palm (UDE)	Integration of updated sections and consolidation
04/07/19	V1.0	Andreas Metzger (UDE)	Final version

Contents

1 Introduction

1.1 Context and objectives

The operation of large-scale and highly distributed IoT system can easily overwhelm operation teams. Major challenges are to improve their efficiency and the collaboration with developer teams for rapid and agile evolution of the system. In particular, automated solutions for run-time operations are required in order to ensure timely reaction to problems and changes of the IoT system's environment.
WP3 aims to develop enablers for the operational part of the DevOps process (see Figure 1). WP3 thus will provide enablers that furnish the IoT systems with capabilities to (i) monitor their status, (ii) indicate when their behaviour is not as expected, (iii) identify the origin of the problem, and (iv) automatically perform typical operation activities (including self-adaptation of the systems). As it is impossible to anticipate all problems and environment situations systems may face when operating in open contexts, there is an urgent need for mechanisms that will automatically learn and update the operation and adaptation activities of Smart IoT Systems (SIS).

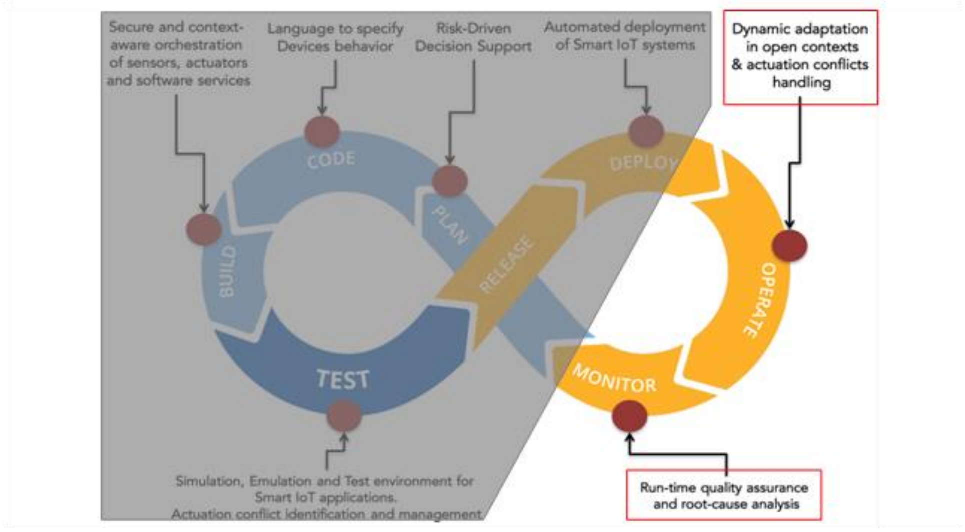

Secure and context-aware orchestration of sensors, actuators and software services
Language to specify Devices behavior
Risk-Driven Decision Support
Automated deployment of Smart IoT systems
Dynamic adaptation in open contexts & actuation conflicts handling
CODE
PLAN
BUILD
TEST
RELEASE
DEPLOY
OPERATE
MONITOR
Simulation, Emulation and Test environment for Smart IoT applications. Actuation conflict identification and management
Run-time quality assurance and root-cause analysis

Figure 1: Focus on the Ops of the DevOps cycle

The three enablers developed by WP3 are:

- **Online Learning Enabler:** Because anticipating all possible context situations that SIS may encounter during their operation is not possible, it is difficult for software developers to determine how a run-time adaptation of the system may impact the satisfaction of the system behaviour and of the interactions with the environment. To address this challenge, this enabler will apply online learning techniques to improve the way a SIS adapts during its operation. Online learning means that learning is performed at run-time, taking into account observations about the actual system execution and system context. Online learning incrementally updates the SIS's knowledge base; *e.g.*, its adaptation rules or the models based on which adaptation decisions are made.
- **Behavioural Drift Analysis Enabler:** Because of the uncertain, dynamic, and partially known nature of the physical environment, it is very difficult or even illusory to assess at run-time the conformity of the effects of actions in this environment with deterministic models. This enabler will provide a set of observers to monitor the behavioural drift of SIS that may arise when operating in such open context. In addition, it will exploit the computed drift measure to dynamically adjust the behaviour of the system.
- **Root Cause Analysis Enabler:** When anomalous conditions start to arise in a complex system, determining which anomalies are related and to which part focus attention is crucial to reduce the mean time to resolution. Thus, the root-cause analysis enabler will try to sensibly group anomalies related to the same problem and compute likely culprits of that problem with the least amount of human involvement possible. Since the number of open incidents in a large

deployment can be large, it will as well prioritize the different grouped problems by potential impact, based on past experience.

Deliverable D3.2 provides the initial version of the ENACT enablers. The executable code of the implementations is provided in the ENACT online repository (https://gitlab.com/enact) and a link to each subproject of each enabler is provided in section 3 together with an overview of the conceptual solution of each enabler, which builds a theoretical foundation for the implementations and a description of the prototype. Furthermore, a documentation on how to use the enabler is provided in the readme-files of the subprojects in GitLab. Planned and already implemented communication between the different enablers are described in D5.2. This is the accompanying document of the software delivery of D3.2, providing descriptions of the set of enablers delivered. Below is the overview of the presented enablers:

1.2 Achievements

Objectives	**Achievements**
Provide 1st version of each enabler of WP3 • Online Learning • Context Monitoring & Behavioural Drift Analysis • Root Cause Analysis	Based on the theoretical foundations gained throughout the first half of the project we developed initial versions of the different enablers of WP3 and provided them in an online repository. The status and progress of each of these is described in Section 1.3.
Provide description of conceptual solution of each enabler	Based on the finding during the first half of the project we developed a conceptual solution for each enabler which laid the foundation for the corresponding enabler implementation. The status and progress of each of these is described in Section 1.3.
Provide documentation	We provided a documentation for each enabler stating how to use. The status and progress of each of these is described in Section 1.3.

1.3 Status of the different enablers

The following table gives a brief overview of WP3's enablers and their current status:

Enabler	**Status**
Online Learning	1st prototypical implementation of the enabler. Demonstration of the enabler has been done using Brownout RUBiS. The enabler is able to support the adjustment of a system parameter that influences the trade-off between two contrasting quality requirements during runtime. Possible interfaces with other enablers have been defined in D5.2. Development of REST API is ongoing. Demonstration of usability in use case (cf. D1.1) is pending.

Context Monitoring & Behavioural Drift Analysis	A first implementation of this enabler provides: (1) A tool for collecting contextual information and providing the streaming data for behavioural drift computation (2) A first behavioural drift analyser (BDA) for apprehending the difference between expected and observed behaviour of the system. This first BDA is based on Gaussian Mixture Model of the observed behaviour.
Root Cause Analysis	A first version of the RCA enabler is expected to be available during the following months. This implementation will include a preliminary database of anomalies (at least 3 patterns), the Agent plugins for at least 2 agents and the system graph construction algorithm.
Adaptation enactment (GeneSIS execution engine)	The initial implementation of the Orchestration and deployment enabler (modelling language and execution environment), supporting deployment over IoT, Edge and Cloud infrastructure. This enabler (aka. GeneSIS) provide initial support for the dynamic adaptation of the deployment of a SIS. A first version of its models@run.time engine is implemented and the GeneSIS execution engine expose a set of API for third-parties to trigger an adaptation. Future work will focus first on extending these APIs with support for high level adaptation commands (e.g., software migration).

1.4 Structure of the document

The remainder of the document is structured as follows. After a brief introduction, Sections 2-4 provide the conceptual solutions for each of the enablers of WP3 and a description of the prototypical implementation of the enablers. Section 5 describes how the self-adaptation is supported by adaptation enactment. Section 6 concludes the document and gives a brief overview of planned next steps.

2 Online Learning for Adaptation Self-improvement of Smart IoT Systems

In this section the conceptual solution and the prototypical implementation of the online learning enabler is described.

2.1 Conceptual solution

A well-known reference model for self-adaptive systems is the MAPE-K model [1-3], which is depicted in Figure 1. Following this reference model, a self-adaptive software system can be logically structured into two main elements: the *system logic* (aka. the managed element) and the *self-adaptation logic* (the autonomic manager).

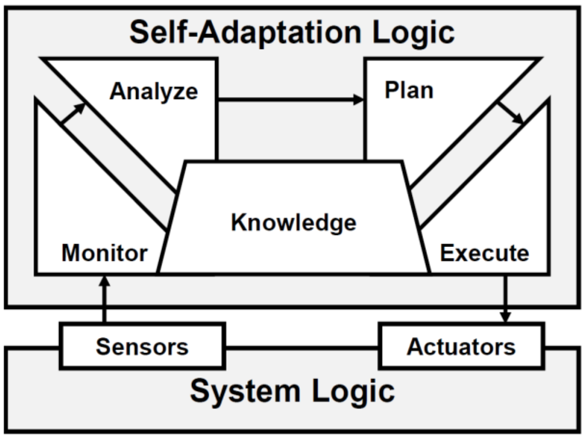


Figure 1: MAPE-K reference model for self-adaptive systems (based on [1])

As shown in Figure 1, the self-adaptation logic can be further structured into four main conceptual activities that leverage a common *knowledge* base [4]. The knowledge base includes information about the managed system (e.g., encoded in the form of models at run time), its environment, and its adaptation goals and adaptation policies (e.g., expressed as rules). The four activities are concerned with *monitoring* the system logic and the system's environment via *sensors*, *analysing* the monitoring data to determine the need for an adaptation, *planning* adaptation actions, and *executing* these adaptation actions via *actuators*, thereby modifying the system logic at run time.

Figure 2 shows the basic reference model for reinforcement learning [5-7]. In this model, a so-called *agent* (i.e., system in our case) learns how to perform optimally in an unknown *environment*.

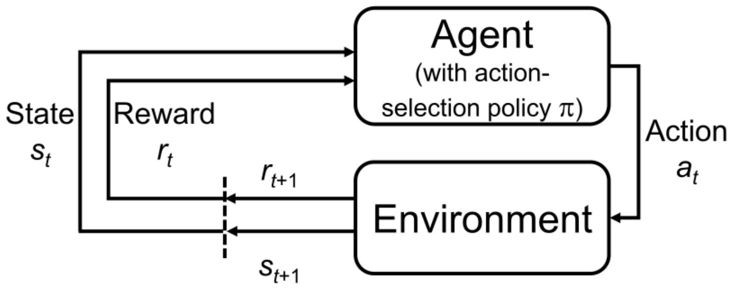


Figure 2: Conceptual model of reinforcement learning (based on [7])

In general, reinforcement learning techniques allow solving sequential decision-making problems of an agent by learning the effectiveness of the agent's actions through interactions with the agent's environment [7, 8]. At each point *t*, an agent observes the current state s_t of its environment. The agent then selects an action a_t, which may cause the environment to change to state s_{t+1}. In each environment state s_t, the agent receives feedback in the form of a reward r_t. The goal of reinforcement learning is to learn an *action-selection policy* π that optimizes the agent's cumulative (long-term) rewards. Depending how rewards are defined for the concrete learning task, the agent's goal may be to maximize or minimize cumulative rewards.

2.2 Online learning enabler using policy-based RL and MAPE-K

Our overall approach is to enhance the MAPE-K loop with policy-based reinforcement learning. Below, we first explain how we conceptually integrate elements of the MAPE-K loop with elements of reinforcement learning, before providing a formalization and implementation of our approach.

Finally, we explain the implementation our approach using a concrete policy-based reinforcement learning algorithm.

Figure 3 shows the conceptual architecture of our approach and how the elements of policy-based reinforcement learning are integrated into the MAPE-K loop.

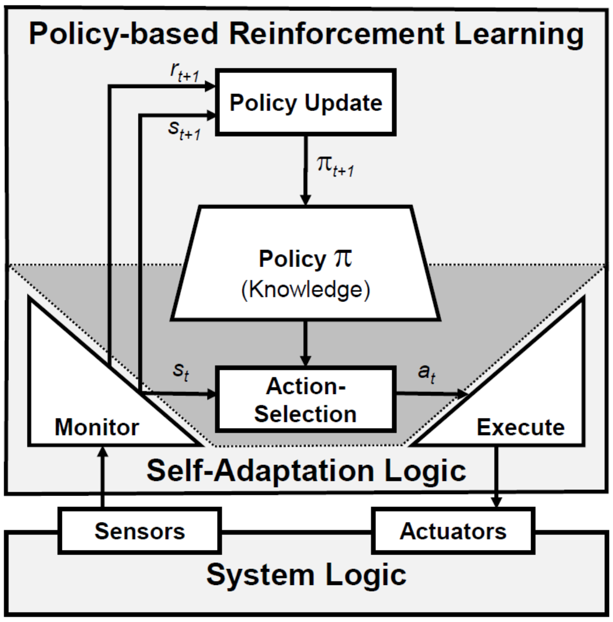


Figure 3: Conceptual architecture of integrating policy-based reinforcement learning with the self-adaptation logic

The overlapping area (dark-gray in Figure 3) shows how in our approach the *action-selection* of reinforcement learning takes the place of the analyze and plan activities of the MAPE-K reference model. In particular, the learned action-selection *policy* π takes the role of the knowledge base of the self-adaptive system. At run time the policy is used by the autonomic manager to select an adaptation action a_t based on the current state s_t, which is determined by the *monitoring* activity.
This means action selection determines whether there is a need for an adaptation (given the current state) and plans (i.e., selects) the respective adaptation action to *execute*.
As we explained before, the monitoring activity monitors the system logic and the system's environment. This means that the state s_t may be determined using observations of both the system's environment and the system logic itself. This is an important difference from the basic reinforcement learning model (as introduced in the beginning), where only observations from the agent's environment are considered to determine the state s_t.
As a result of an action a_t, the state may change to state s_{t+1}. This new state is used to trigger the learning process to deliver an updated policy π_{t+1}. Input to the *policy update* are the new state s_{t+1}

together with the reward r_{t+1} obtained in the new state. In our architecture this reward is also computed by the monitoring activity, as this activity has access to all sensor information collected from the system and its environment.

2.3 Formal background of the conceptual solution

The sequential decision-making problem to be solved by reinforcement learning can be formalized as a Markov decision process, which defines the interactions between an agent and its environment in terms of states, actions, and rewards [7].
A Markov decision processes is formally defined as $MDP = (S, A, T, R)$ with

- S being a set of environment states $s \in S$,
- A being a set of possible actions $a \in A$ that lead to a transition among environment states, i.e., from a state s_t to a successor state s_{t+1},
- $T : S \times A \times S \rightarrow [0, 1]$ being the transition probability among states with $T(s_t, a_t, s_{t+1}) = \Pr(s_{t+1} | s_t, a_t)$, which gives the probability that action a_t in state s_t will lead to a state s_{t+1},
- $R : S \times A \times S \rightarrow \mathrm{IR}$, a reward function which specifies what numerical reward the agent receives when performing a particular action a_t in a particular state s_t that leads to successor state s_{t+1}.

The solution to a Markov decision process is a so-called optimal policy π, which maps states to actions, such that the future reward is maximized. More concretely this policy is defined as $\pi : S \times A \rightarrow [0, 1]$, which gives the probability of taking action a in state s, i.e., $\pi(s, a) = \Pr(a | s)$.
Policy-based reinforcement learning relies on a parametrized policy in the form $\pi_\theta(s, a) = \Pr(a\ s, \theta)$, where θ IR^d is a vector of the policy's parameters. As an example, if the policy is represented as an artificial neural network, the weights of the network are the policy parameters [9]. The policy parameters can be learned via so called policy gradient methods. Policy gradient methods update the policy according to the gradient of a given objective function [7, 9], which can be based on the average reward per learning step
Policy-based reinforcement learning does not rely on a value function for action selection and thus does not require quantization of the state space. The concrete action is selected via sampling over the probabilistic policy π_θ. Because of this sampling and the probabilistic nature of the policy, policy-based reinforcement learning does not suffer from the exploration/exploitation dilemma and thus does not require manually fine-tuning the balance between exploitation and exploration. Exploration is automatically performed, because sampling over the probabilistic policy leads to some degree of random action selection.
With respect to formalizing the self-adaptation problem as a Markov decision process, we know the set of possible actions A. These are the system's adaptation actions, i.e., possible ways the system may be adapted using the system logic's actuators. As an example, we may know which optional system features of a web application may be deactivated in case of performance problems. We also know the set of potential environment states S. That is, even if we do not know the exact environment states a system may face at run time (due to design time uncertainty in anticipating all potential environment changes), we at least know the typical state variables. As an example, even if we do not know the exact workload (and maybe not even the maximum workload) a web application may face, we can express a state variable workload $w \in \mathrm{IN}$.
In contrast, we do not know T due to design time uncertainty about how adaptation impacts on system quality. We thus employ a model- free variant of policy-based reinforcement learning, which does not require a model of the environment (i.e., known T) but can learn directly from interactions with the system's environment.
Finally, defining the reward function R depends on the concrete learning goal to achieve. We define a concrete reward function as part of our evaluation.

To select a concrete policy-based reinforcement learning algorithm for our implementation, we need to consider that, from the point of view of reinforcement learning, self-adaptation is a *continuing task*. This means that the adaptation process cannot be naturally broken down into episodes, which start from

an initial state s_0 and terminate after a certain number of actions in a known terminal state [7]. The adaptation process goes on continually. Value-based reinforcement learning algorithms like Q-Learning and SARSA are well-suited for continuing learning tasks, because they use bootstrapping during the learning process. Bootstrapping means that the algorithms update the knowledge base after each time step t without waiting for a final outcome, i.e., without waiting for reaching a terminal state. Actor-critic algorithms are a variant of policy- based reinforcement learning algorithms that also use bootstrapping and thus are suited for continuing tasks and thus for self-adaptive systems.
To show the feasibility of using policy-based reinforcement learning for self-adaptive software systems, we use proximal policy optimization (PPO), as a state-of-the-art actor-critic algorithm [10]. PPO algorithms may perform comparably or better than other policy gradient methods, while at the same time being easier to implement due to a simpler objective function for gradient descent than other state-of-the-art actor-critic algorithms. PPO is used as default reinforcement learning algorithm by OpenAI.

2.4 Prototypical implementation

As already mentioned before, the heart of the online learning enabler are Reinforcement learning algorithms. For the sake of simplicity, we used the baseline implementations of policy-gradient algorithm provided by OpenAI[1]. As these baseline implementations are meant to be used with the environments provided by OpenAI[2], we implemented an environment adapter, which makes an arbitrary system, whose parametrization can be formulated as a sequential decision-making problem (see above), to appear as an OpenAI gym environment to the algorithm. For this first prototypical implementation we focused on proximal policy optimization as a state-of-the-art actor-critic algorithm [11].

The architecture of the prototypical implementation of the online learning enabler can be seen in Figure 4. PPO2, TRPO and DDPG are learning algorithms. As mentioned above, we us PPO in our solution, as it is a state-of-the-art actor-critic algorithm.

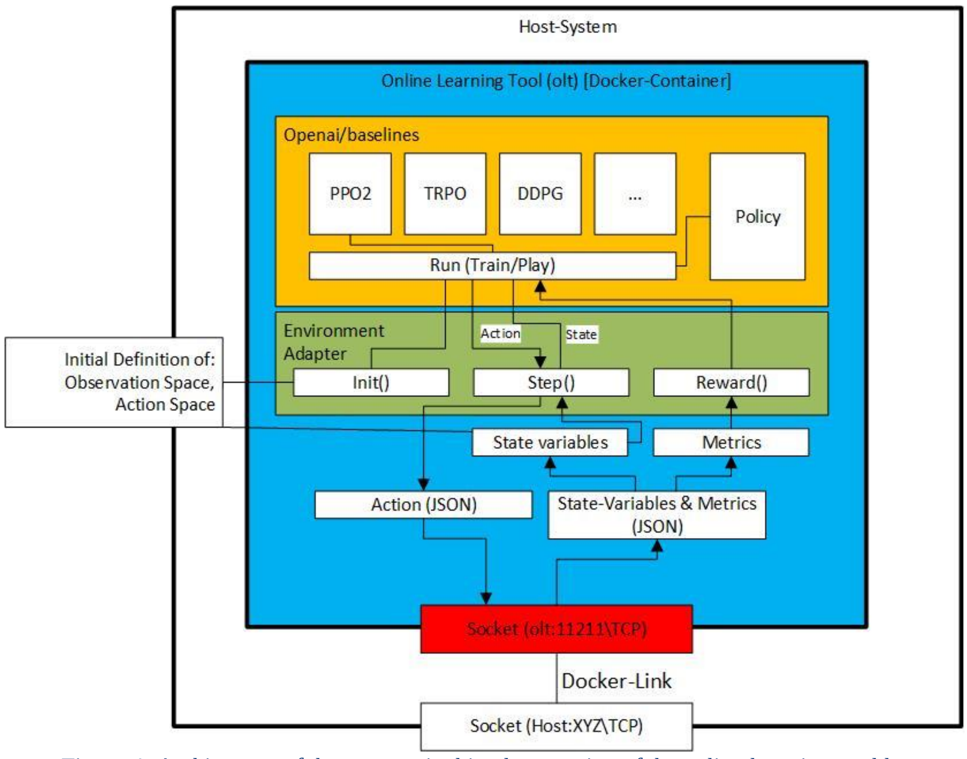


Figure 4: Architecture of the prototypical implementation of the online learning enabler

1 https://github.com/openai/baselines
2 https://gym.openai.com/

The online learning enabler resides in a docker container which can be accessed through TCP-Port 11211 to provide input information about the current environment state and metrics of the system to adapt. State variables are used to feed the step-method, so that the next timestep according to the underlying sequential decision-making problem is reached. The metrics are used by the reward-function to compute the reward. Reward and state-variables are then used to update the policy which is done by the PPO baseline implementation of OpenAI (usage of other algorithms like TRPO, DDPG, should technically also be possible). According to the underlying algorithm an update of the policy is not done after every timestep, because batches of experience are used for an update. The policy is then used to determine the next action to be evaluated in the current environment state.
While listening on the according socket, the actions issued by the online-learning enabler are received by the system to be adapted.
To make the online learning enabler properly work, several parameters and methods need to be configured beforehand according to the system whose adaptation should be enhanced. A guide describing how to configure the different parameters of the online learning enabler is given in the readme of the repository. In the final version of the enabler, all the parameters can be configured through a proper API. As described in D5.2, potential other enablers being connected to the online learning enabler are Genesis, ACM, Context Monitoring and Behavioural Drift Analysis and Root Cause Analysis (RCA). Genesis or ACM can be used for the execution of the proposed action and the delivery of proper information about the action space, Context Monitoring & Behavioural Drift Analysis for the proper delivery of the state information to feed the algorithm and RCA for further adjustment of the action space boundaries.

The prototype can be accessed via GitLab: https://gitlab.com/enact/online-learning-enabler (A setup guide is provided in the readme of the repository).

3 Context Monitoring and Behavioural Drift Analysis for Smart IoT Systems

Context-awareness is key for Operational monitoring of Smart IoT Systems (SIS). The so-called "context-aware" SIS, by understanding their operational context, are thereby able to relevantly adapt upon environmental changes. In this context, the behavioural drift computation enabler aims at providing a common context monitoring tool whose innovation resides in using contextual information for monitoring their behaviour, detecting and, more importantly, analysing their drift over time. This objective raises the following research question: "*How one can observe the evolution of the system from different contextual points of view so as to evaluate how much this behaviour is close to the expected one?*".

3.1 Motivation and Illustration

Because SIS operate within the real environment, models of their physical surroundings are required to design them. However, as complex as they might be, models remain abstraction of the physical world. This makes difficult to ensure that a SIS behaves as expected if it is unlikely to evolve as predicted by the models. For instance, let us consider a smart building scenario, such as the TECNALIA use case. When can a room be considered as *effectively* illuminated? Must we verify that the luminosity level is exceeding a predetermined threshold? Which threshold? Everywhere in the room, or only on the tables, on the seats? Moreover, what happens when the context is changing, e.g., the sun is shining, windows are opened and shadow of the trees outside home is projected into the living room?

Developers rarely ask themselves with all of these questions and assume a simple model of the world where switching lights within a room is good enough for ensuring expected lighting level for the whole space. A more realistic assumption is that one is unlikely to obtain a comprehensive model of the physical environment because of its complexity and changeability.

Although environmental and behavioural models are needed at development time, these models are no longer effective for evaluating the IoT system conformance at operation time.

An alternative approach, more pragmatic, consists in monitoring the behaviour of the IoT system by leveraging contextual information built from observations and to ensure that the observed behaviour is close to what designers expect.

In the next section we describe the Enabler for Context Monitoring and Behavioural Drift Analysis (BDA).

3.2 Conceptual solution and Highlights on Contributions

Unlike classical behavioural monitoring approaches, we introduce the concept of "behavioural drift", that is neither limited to the strict conformance between the observed and the expected behaviour, illusory in a real world, but is quantitative. Behavioural drift can be computed but also analysed in different ways thanks to the available contextual information, i.e., observable properties of the environment.

In ENACT, most of the time, use case providers have their own middleware for collecting contextual information from sensors (e.g., TECNALIA with SMOOL, Indra with FIWARE). The Enabler provides context monitoring and behavioural drift analysis Tools for major IoT platforms (Figure 5).

Historically, we started with a Behavioural Drift metrics, computed as a scalar value reflecting the "*likelihood of the observed behaviour to correspond to the expected one*" (result [10] prior to ENACT but potentially interesting as a reward for WP2 self-adaptive actuation conflict management based on

UDE reinforcement learning solution). For ENACT, our ambition is to develop a set of behavioural analysis tools providing designers with a more interpretable feedback from Operational System Monitoring [12].

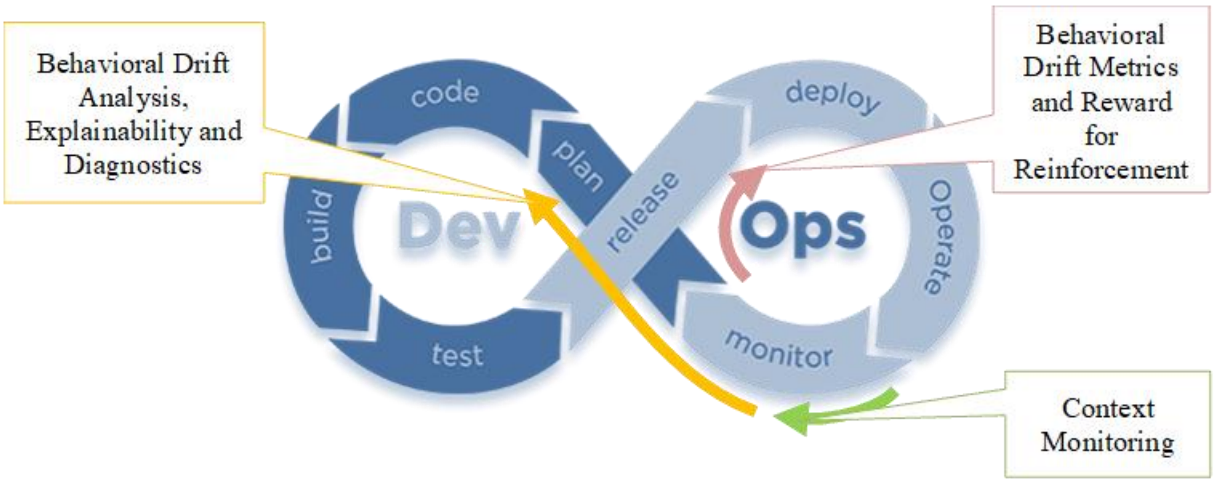

Behavioral Drift Analysis, Explainability and Diagnostics
code
plan
build
Dev
test
release
deploy
Ops
Operate
monitor
Behavioral Drift Metrics and Reward for Reinforcement
Context Monitoring

Figure 5: Context Monitoring and Behavioural Drift Analysis in DevOps for IoT

3.2.1 Overall Context Monitoring and Behavioural Drift Analysis Enabler

Our contribution is about analysing SIS (Smart IoT System) behavioural drift one step beyond a quantitative value representing the difference between the observed and the expected behaviours of a Smart IoT system. Thanks to different set of sensors, different points of view on the SIS behaviour can be observed at a same time and provide a set of quantitative metrics. The behavioural drift metrics [10] is a way to overcome a Boolean conformance evaluation limited to PASS/FAIL results. Such indicators are well adapted for monitoring the system and detecting its behavioural drifts.

Unfortunately, although we can detect and measure behavioural drifts, these metrics are not informative about their root cause(s), i.e., it does not explain the drifts. Such information would be valuable for the designers to potentially catch the possible weaknesses of the design.
In this context, the main purpose of the behavioural drift analysis tools are to produce interpretable feedback in the form of *models* of the observed behaviour that might be compared to the expected ones.

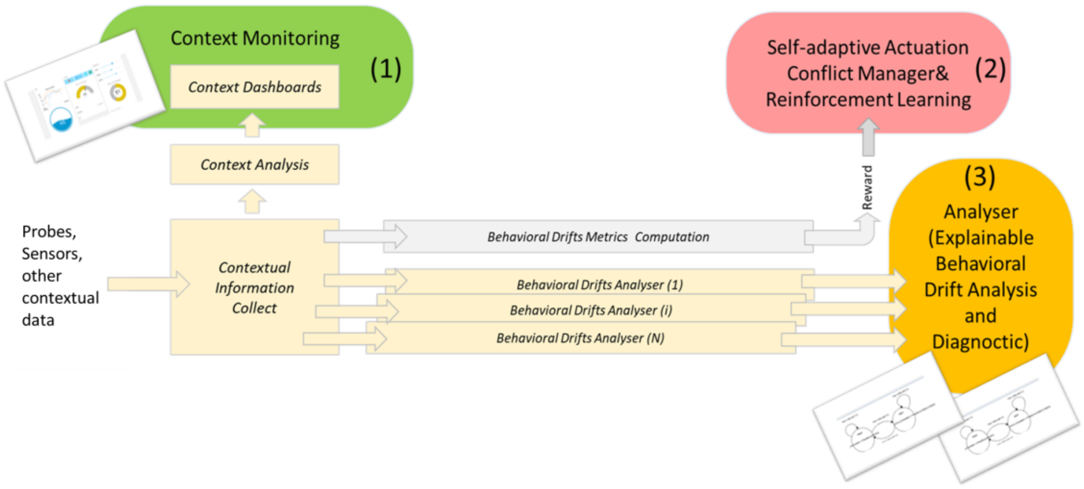

Context Monitoring
Context Dashboards
(1)
Self-adaptive Actuation Conflict Manager& Reinforcement Learning
(2)
Context Analysis
Reward
(3)
Analyser (Explainable Behavioral Drift Analysis and Diagnoctic)
Probes, Sensors, other contextual data
Contextual Information Collect
Behavioral Drifts Metrics Computation
Behavioral Drifts Analyser (1)
Behavioral Drifts Analyser (i)
Behavioral Drifts Analyser (N)

Figure 6: Enabler Overall Architecture

Thus, as depicted in Figure 6, besides context monitoring (1) and behavioural drift metrics (2) (detailed in section 3.2.2), the BDA enabler consists in analysers (3), based on different algorithms, provides interpretable feedback in the form of deterministic and stochastic models of the observed behaviour. To this end, the first algorithm implemented in BDA enabler is the Gaussian Mixture Model (detailed in Section 3.2.3.1).

3.2.2 Highlights on Context Monitoring and Behavioural Drift Metrics

Context Monitoring (see Figure 7) is a basic requirement for other ENACT enablers (see D5.2) and use case providers (see D1.1). At the first level, it allows to store, process and visualize contextual data of interest. These data are collected thanks to:

- Different existing middleware for context management (e.g., TECNALIA with SMOOL, Indra with FIWARE) leveraged by the use-cases providers,
- Additional equipment directly connected to the *contextual data collector*.

Contextual data collector ensures the interoperability between the context monitoring part of the enabler and the different middleware exploited by the use-cases providers.
In order to be as generic as possible, contextual data are formatted as follows:

```
<contextual topic, data value, metadata>
```

Each field is a high-level concept and may be expanded later according to the use cases. Contextual data may be stored in a Knowledge Base, potentially associated with an ontology allowing to reason on the contextual information and metadata.

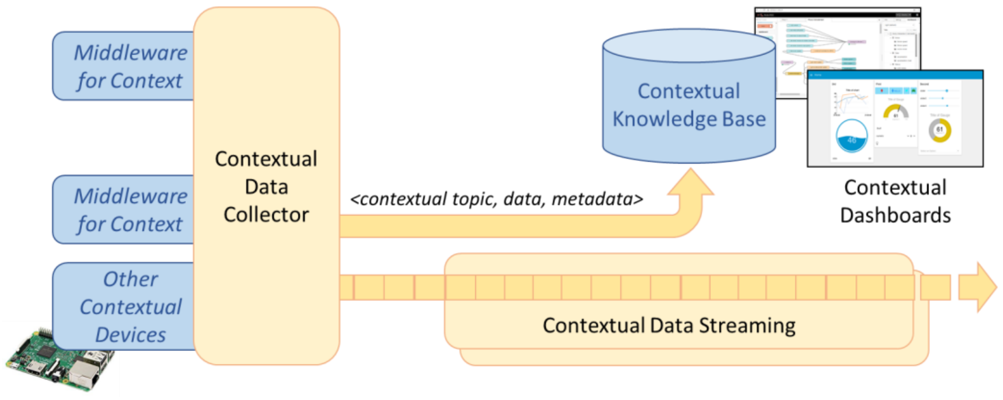

Middleware for Context
Middleware for Context
Other Contextual Devices
Contextual Data Collector
<contextual topic, data, metadata>
Contextual Knowledge Base
Contextual Dashboards
Contextual Data Streaming

Figure 7: Context Monitoring Tool

Enabler V1 stores contextual data in a classical database. Front-end applications can then implement some contextual data processing and contextual data dashboards according to the needs of the use cases providers.

Beyond this classical approach, Enabler algorithms consume a continuous stream of synchronized contextual information with the aim to monitor the behaviour of IoT systems (Figure 7). In section 3.2.2, for example, the stream of synchronized contextual data is used to compute behavioural drift [10].

In order for the designer to easily interact with the behavioural drift measurement tool, the description of the probabilistic behavioural model is based on a deterministic Finite State Machine (FSM) describing the ideal expected behaviour and added probabilistic information on inputs/outputs occurrences. A deterministic Finite State Machine (FSM) is describing the ideal expected behaviour, supposed to be known a priori. Developers are more familiar in working with FSM rather than complex stochastic models that include uncertainties modelling through complex distributions of probabilities.

This *a priori* model can be further transformed into its probabilistic counterpart by adding some probabilistic information on inputs/outputs occurrences and values. The uncertainties model is a probabilistic one. For instance, the probabilistic model may be an Input/Output Hidden Markov Model (IOHMM) based on Gaussian distributions of probabilities accounting for uncertainties pertaining the physical environment. This model comes with computationally efficient algorithms for evaluating the effectiveness of the system's behaviour (i.e., Likelihood computation on the I/O observed sequences).

In Figure 8, a probabilistic-based behavioural drift metric is described. The objective is to ensure that the behaviour of an indoor light control system is consistent with its expected behaviour whose model is depicted in the figure on the left. Based on input/output observations where input corresponds to a motion detector (blue curve) and output is associated with a sensor measuring the luminosity level (green curve, the behavioural drift is computed (bottom curve). The expected behavioural model is quite simple, the luminosity level must be greater than a threshold when the presence of an inhabitant is detected in the room, below threshold otherwise.

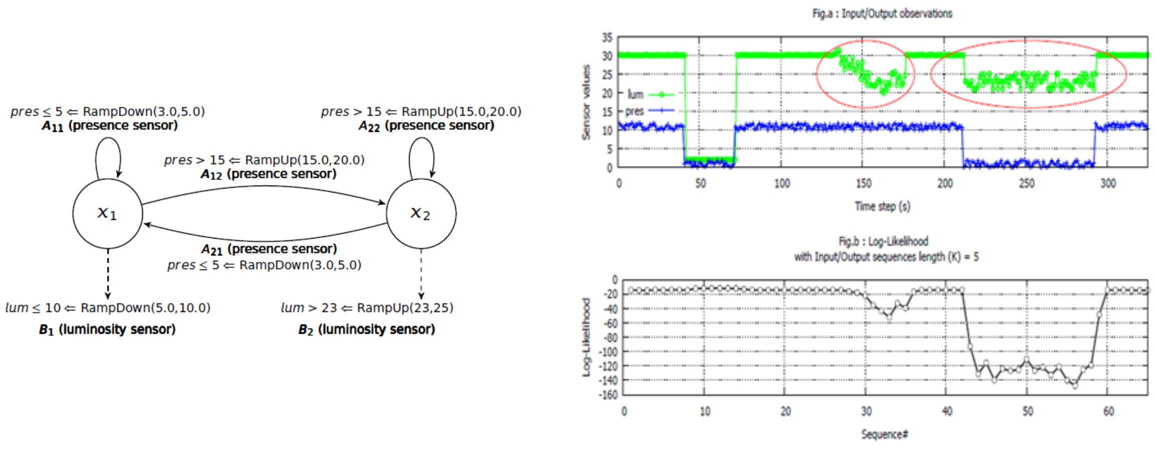

pres ≤ 5 ⇐ RampDown(3.0,5.0)
A11 (presence sensor)
pres > 15 ⇐ RampUp(15.0,20.0)
A22 (presence sensor)
pres > 15 ⇐ RampUp(15.0,20.0)
A12 (presence sensor)
x1
x2
A21 (presence sensor)
pres ≤ 5 ⇐ RampDown(3.0,5.0)
lum ≤ 10 ⇐ RampDown(5.0,10.0)
B1 (luminosity sensor)
lum > 23 ⇐ RampUp(23,25)
B2 (luminosity sensor)
Fig.a : Input/Output observations
Sensor values
lum
pres
Time step (s)
Fig.b : Log-Likelihood with Input/Output sequences length (K) = 5
Log-Likelihood
Sequence#

Figure 8: Example of expected observed behaviour probabilistic model and behavioural drift metrics evolution

The main interest of the behavioural drift metric in ENACT is not only to detect radical changes in the behaviour of the IoT system, but also to provide an original reward for reinforcement learning in self-adaptive actuation conflict manager provided by UDE (see Section 2).

3.2.3 Highlights on Behavioural Drift Analysis

Behavioural drift analysers are introduced in ENACT to provide an intelligible and explainable feedback to the designers of the system considered.

3.2.3.1 First Behavioural Drift Analyzer v1 for ENACT

In this section, we do elaborate on the Behavioural Drift Analyser, which borrow some approaches from AI domain [13]. Specifically, it consists in learning the behaviour of the system from contextual observations. The outcome of the learning process consists in a deterministic part in the form of a Finite State Machine (FSM) and its stochastic counterpart in the form of a probabilistic Markov model. Bolstered by these learned models, designers can compare them with their initial understanding of the behaviour of the system, analyze the differences and apprehend the variability of the behaviour towards uncertainties.

Assumptions:
The Behavioural Drift Analysis framework relies on several simplifying assumptions:

- The first assumption concerns the observability of the states of the model used to describe the expected behaviour of the system considered. Here, it is assumed that ***each state is partially observable***, i.e., it can be identified by a subset of contextual observations.
- The second assumption concerns the model of the uncertainties towards the expected behaviour. ***Tolerances are supposed to be described through unimodal Gaussian distributions*** where the modes of the distributions represent the expected behaviour.
- Finally, the third assumption concerns the evolution of the behaviour of the system considered. The occurrence of a state x at time t+1 does not depend on the occurrence of the states at time t, t-1, t-2, etc.

Gaussian Mixture Model [14]

When learning the behaviour of a system from observations without a priori considerations on the behaviour (i.e., unsupervised learning), the main challenge is to determine the number of states that best fit the observed behaviour. The approach we selected consists in finding clusters from input/output observations, which, in turn, can be used to identify different states of the behaviour of the system (Figure 9). Observations consist in the effects produced by the system (called outputs of the system in the sequel) and events leading the system to evolve over time (called inputs in the sequel). For instance, in the context of a luminosity controller, the output of the system corresponds to the luminosity of the room whose luminosity is controlled, the input corresponds to any event leading the controller to change the luminosity of the room.

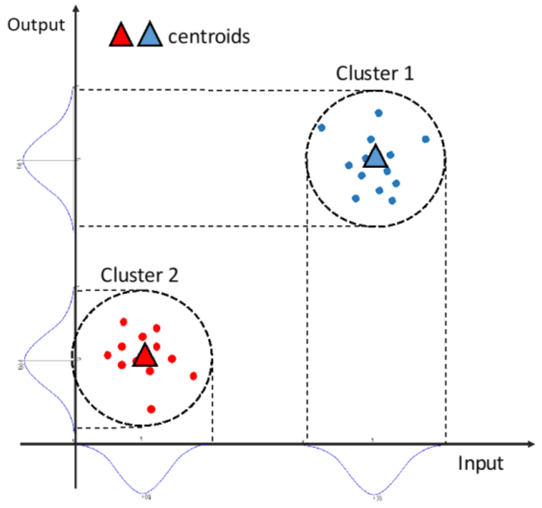


Figure 9 : Gaussian Mixture Model clustering

In BDA v1 (current version), we use the Gaussian Mixture Model (GMM) unsupervised algorithm for obtaining clusters from observations.

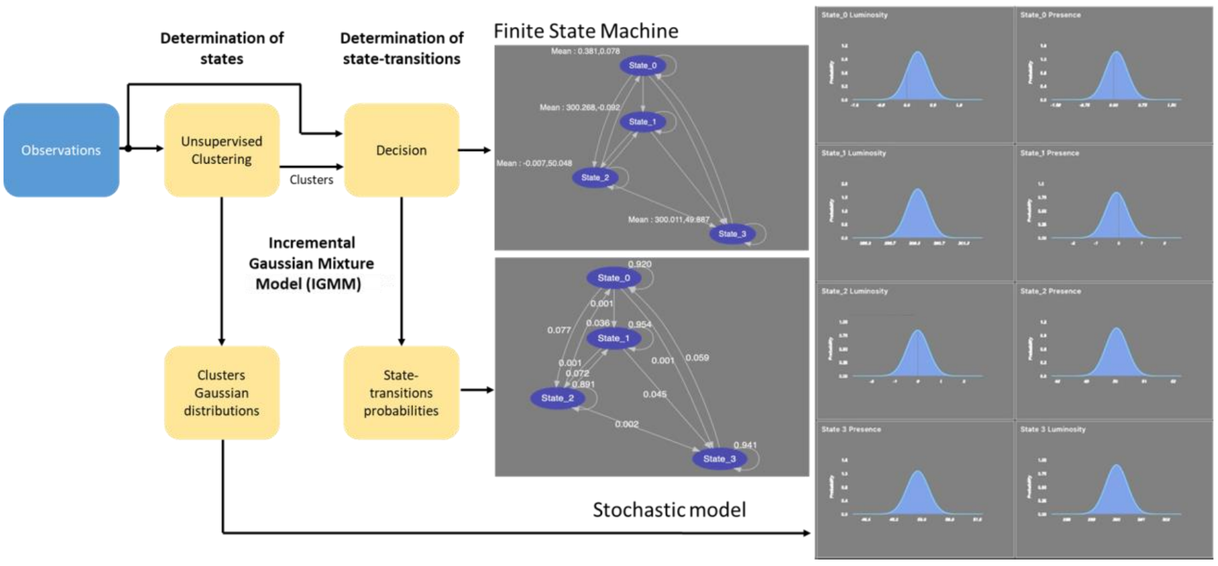


Figure 10 : Observed Behavioural Models Estimation

The process is described in Figure 10 and consists in the five steps described hereafter:

1. The GMM is first used to determine clusters from a sequence of observations and their associated distributions.
2. The sequence of observation is applied to the GMM which associates one of the determined clusters to each observation. Considering that each cluster corresponds to a state of the behavioural model, this step in the process is equivalent to the Viterbi algorithm used in Markov models to determine the state sequence underlying a sequence of observations.
3. With the states defined on the basis of the clusters generated in step (1) and the sequence of states determined in step (2), one can compute the state transitions. This provides us with the FSM whose states are characterized by the centroids of the distributions associated to each cluster/state.
4. Applying the same process than the one described in step (3), one can compute the state-transition probabilities associated to each state-transition from observations. It is worth noting that the probabilities depend on the observation sequence.
5. State-emission probabilities are given by the GMM algorithm and correspond to the cluster's distributions.

Steps (1) to (3) allow to compute the FSM. Steps (4) to (5) allow to compute the stochastic model.

Further evolutions are planned towards BDA v2 (next version):

Considering inputs in state-transitions:

While the state-transitions in BDA v1 (see Figure 11) are associated with hard-coded probabilities, it would be interesting for these probabilities to be input dependent hereby being more coherent with the evolution of the behaviour of the system considered as a dynamical system.

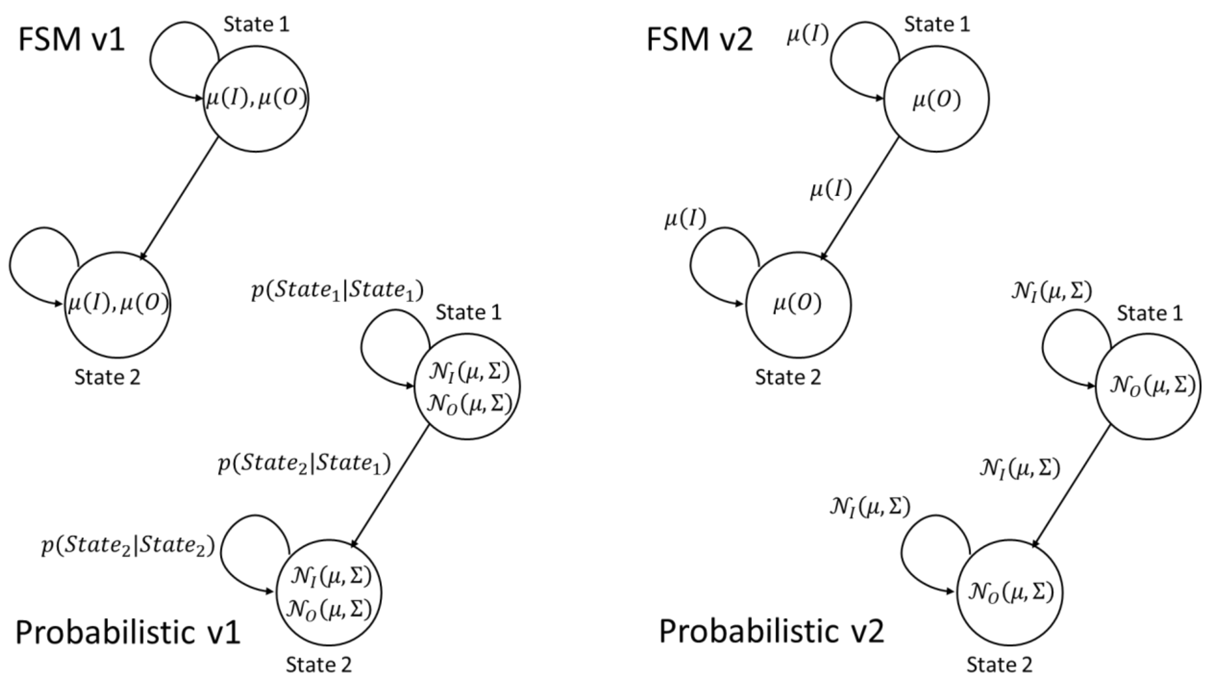

FSM v1
State 1
$\mu(I), \mu(O)$
$\mu(I), \mu(O)$
State 2
Probabilistic v1
$p(State_1|State_1)$
State 1
$\mathcal{N}_I(\mu, \Sigma)$
$\mathcal{N}_O(\mu, \Sigma)$
$p(State_2|State_1)$
$p(State_2|State_2)$
$\mathcal{N}_I(\mu, \Sigma)$
$\mathcal{N}_O(\mu, \Sigma)$
State 2
FSM v2
$\mu(I)$
State 1
$\mu(O)$
$\mu(I)$
$\mu(I)$
$\mu(O)$
State 2
Probabilistic v2
$\mathcal{N}_I(\mu, \Sigma)$
State 1
$\mathcal{N}_O(\mu, \Sigma)$
$\mathcal{N}_I(\mu, \Sigma)$
$\mathcal{N}_I(\mu, \Sigma)$
$\mathcal{N}_O(\mu, \Sigma)$
State 2

Figure 11 : BDA v1 vs BDA v2: considering inputs in state-transitions

This transformation is straightforward from information obtained from BDA v1.

Considering GMM incremental learning:

GMM works on sequences of observations. The clustering is achieved based on the sequence of observation and any new sequence leads the learning process to be started from scratch irrespective of the previous sequences of observations. In the BDA v2 (see Figure 11), an incremental GMM (IGMM) will be implemented so as to cope with the aforementioned limitation.

3.3 Prototypical implementation

Based on the proposed approach, a concrete example developed with the Behavioural Drift Analysis enabler V1 is shown in the sequel. As presented in section 0, let us consider the example of a luminosity controller. A developer could have specified how the light is supposed to be switched on and off depending on whether a person is present or not in the room. We would like to compute and analyse the behavioural drift on two different aspects: (1) the room is illuminated when someone is present inside (BDA 1 on Figure 13) and (2), power consumption is correlated to the presence and the outdoor luminosity (BDA 2 on Figure 13).

The developer must first specify a deterministic model as a Finite State Machine (FSM) of the expected behaviour he would like to analyse. For the described example about the room illumination, the developer specifies the FSM shown on Figure 12. This model can be read as follows: while a presence is detected in the room, its luminosity must be higher than 95. Some constraints are added on states and transitions for each sensor value used to compute and analyse the drift. The constraints define tolerance on the specified sensor's values. The developer can specify these constraints on the right part of the interface as shown on Figure 12.

Although the model described here is simple, the proposed approach does not limit its complexity. The only limit is relative to the number of sensors available and accessible throughout the context monitoring.

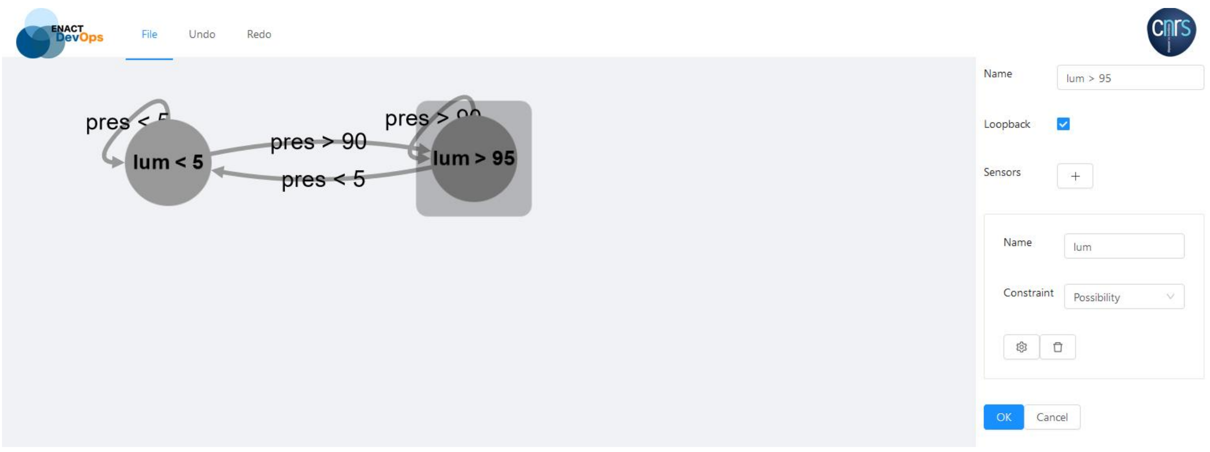


Figure 12: Deterministic model describing the expected evolution of physical model (relation between luminosity and presence in the room)

Once the FSM defined, the developer can export this model which is transformed into a Node-RED data flow. Figure 13 shows the Node-RED dataflow for the two different Behavioural Drift Analysis described previously. Two BDA are implemented in Figure 13. The first one is the result obtained after exporting the FSM described in Figure 12.

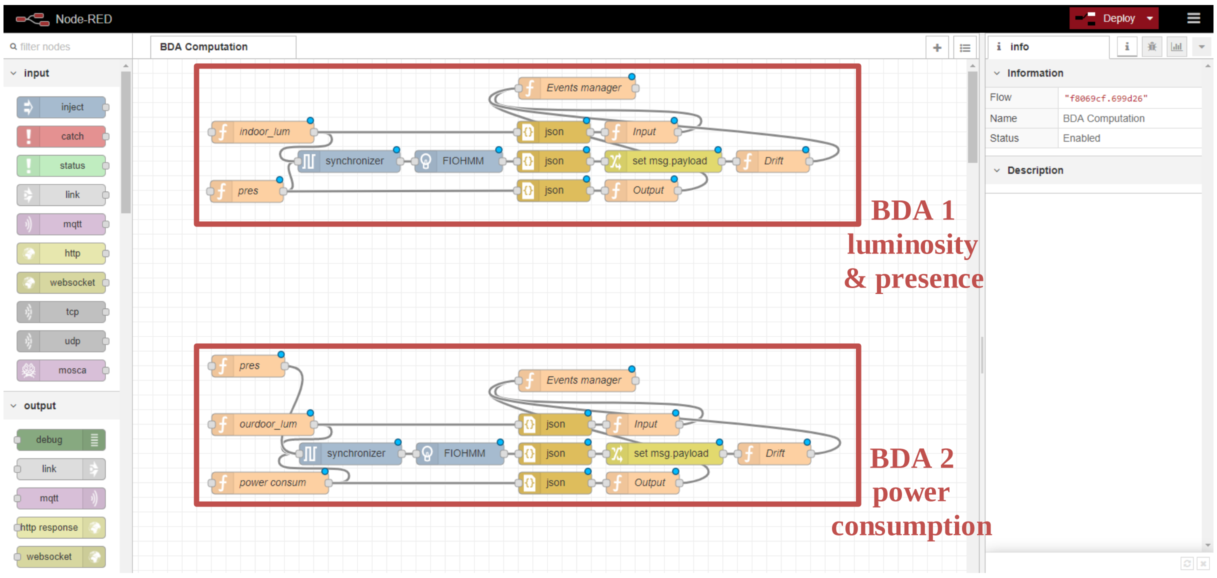


Figure 13: Behavioural drift analysis dataflow generated in Node-RED

On the left part of this data flow, the developer must provide the sensors values as input of the Behavioural Drift Analysis process. These inputs come from the context monitoring. On the right part of the data flow, the developer adds the desired treatment on the outputs (Figure 14).

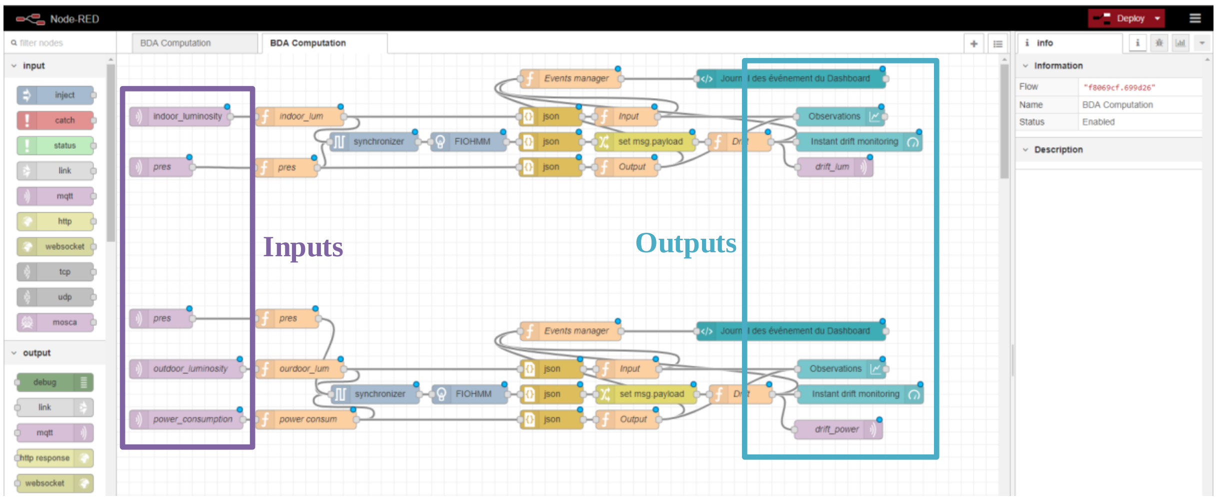


Figure 14:Connected context monitoring inputs and outputs to the generated BDA dataflow

As presented on Figure 15, the Behavioural Drift Metrics Computation and Analysis can be used for different purposes such as: exposing computing outputs to other tools (like proposing the Behavioural Drift Metrics Computation to the Online Learning Enabler as described in deliverable D5.2), presenting output data throughout Dashboards (Figure 15) or proposing to the developer some representations of true observed behaviour for further Behavioural Drift Analysis (Figure 16).

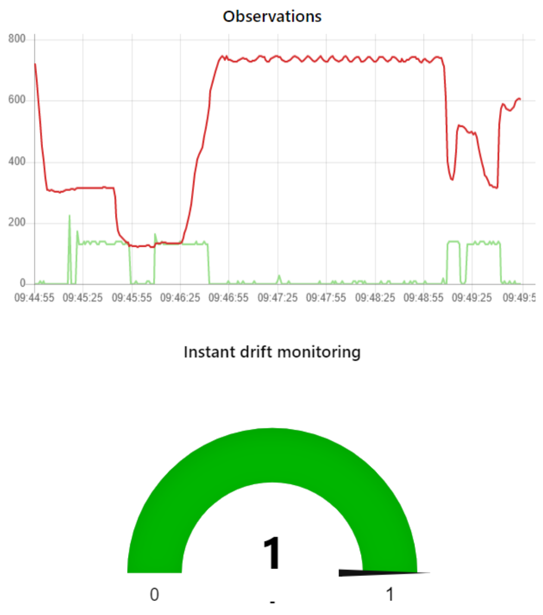


Figure 15: Dashboard monitoring sensors values and computed drift value

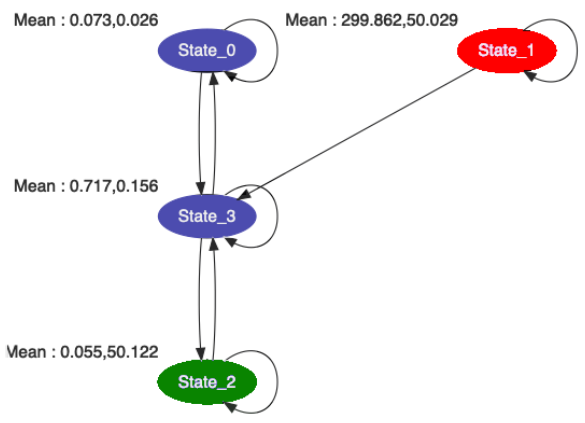


Figure 16: Behavioural model learned from observations

The developer can then use this information to understand the behaviour of the system running in the real physical environment. For instance, the behavioural model learned from observations can be compared with the initial one and further used as a basis to fix it or understand the source of deviation between the expected behaviour and the observed one.
The current implementation of the Behavioural Drift Analysis enabler is available at the following address: https://gitlab.com/enact/behavioural_drift_analysis

4 Root-Cause Analysis for Smart IoT Systems

The ENACT DevOps Framework aims to provide a set of tools to ease the operations of smart IoT systems. In order to support the managing and diagnosing of the whole system, ENACT includes a Root Cause Analysis enabler that helps detecting and identifying the cause of any attack or malfunctioning of the platform. To this end, the provided tools will collect data from different network probes or instrumented software and will analyse it in order to provide the Root Cause of the issue along with an enhanced view of the collected data.

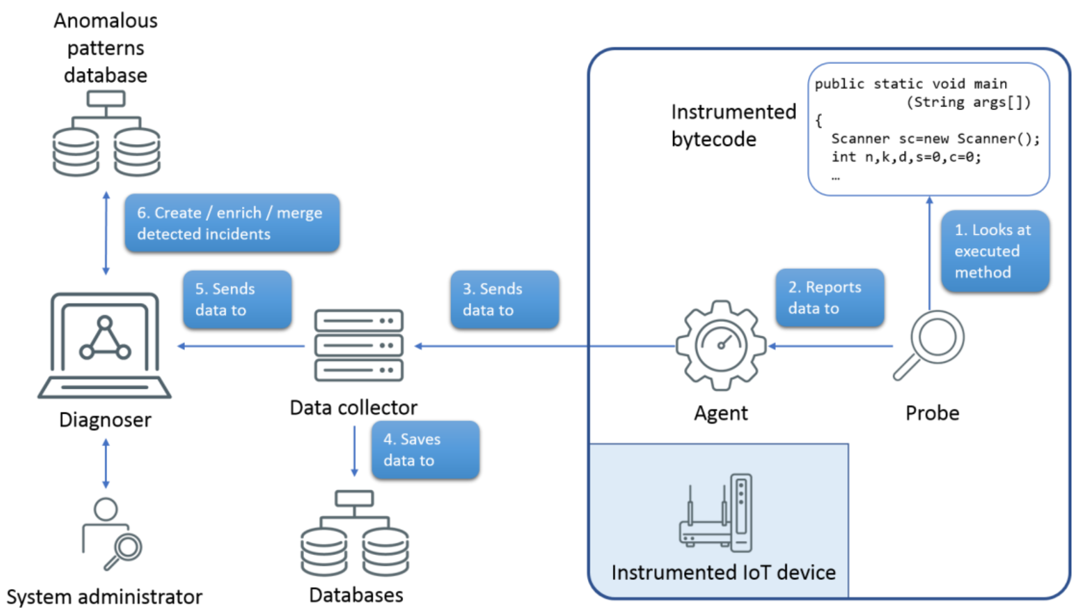
The following subsections provide insights of the conceptual solution of the Root Cause Analysis component of the ENACT DevOps Framework.

4.1 Conceptual solution

The ENACT Root Cause Analysis (RCA) enabler is composed of several innovations that will be presented in the following sections, including: (1) the ENACT RCA enabler and the architectural design that relies on Agents that will collect the data required to monitor the devices; (2) a graph-based root cause analysis algorithm that will detect anomalies and determine their cause; (3) aggregation algorithms that will construct the system graph using the extracted data; and (4) a graph-similarity algorithm that will be in charge of determining the similarity level of the detected issue (a part of the system graph) with respect to a set of already-known anomalies and their root cause.
The following sections provide further insights about how these innovations will be implemented in the ENACT RCA enabler.

4.1.1 Root Cause Analysis Design

The previous document D3.1 presents the general design of the RCA enabler, which is depicted in Figure 17. This section provides insights about how each of these components will be implemented in the ENACT DevOps Framework.



```
public static void main
        (String args[])
{
  Scanner sc=new Scanner();
  int n,k,d,s=0,c=0;
  …
```

Figure 17: General Design of the RCA enabler

4.1.1.1 Probes and Agents

Probes are defined as objects that are used to obtain information from the network or the device under test. Depending on the nature of the probe (software- or hardware-based) there are different ways the probe retrieves the information from the monitored devices.

Firstly, software-based probes are the extractors that are implemented and deployed as purely software solutions; however, they can also be deployed in different ways. The simplest way of implementing software-based probing is by instrumenting the software of the devices that need to be monitored. This can be implemented by using specific instrumentation frameworks (which hardly depend on the programming language used) or by simply embedding logging capabilities in the code of the devices that require monitoring. Similarly, a probe can also be implemented as purely software by means of running specific software that is capable of periodically checking the state and extract information from the devices being monitored. Regardless of the technique used, both approaches can be complemented in order to provide an enriched view of the status of the devices.
Secondly, hardware-based probes are physical devices that contain software developed specifically to monitor the devices (and/or the network) under study. These devices are deployed in the proximity of the involved devices in order to maximize the information they can extract. This type of probe is usually more expensive than the software-based ones (since they require specific hardware to run) but they offer a significant advantages with respect to the pure-software ones. In particular, hardware probes minimize the overhead introduced by the monitoring process (since they can run on dedicated hardware). In this sense, hardware-based probes can be implemented in a more general way, being able to monitor a wider range of devices. For example, instead of instrumenting a client on the network to extract the statistics of the transferred data, is it possible to use a general-purpose network sniffer. In this way, it would be possible not only to analyse the client under study, but also retrieve statistics from other protocols that can have impact on the performance of the monitored device.
Despite the substantial difference in the aforementioned probe types, in both approaches the probes interact with the Agent component, which is the interface between the probe (concrete implementation to extract the data) and the external consumers of this information. The agents will be in charge of processing the raw data collected directly from the monitored device and putting it in a format that can be understood by other parts of the RCA enabler.
Considering both types of probes and agents, the ENACT RCA enabler will be conceived to interact with both software- and hardware-based probes. In particular, the ENACT Probes/Agents will not only retrieve information from the logs generated by the devices on the IoT System, but they will also be able to interact with hardware-based probes (such as the MMT-IoT Sniffer) and software-based solutions (such as Snort and Suricata).

4.1.1.2 Data Collector and Databases

Before being detected, a failure might cause slight anomalies that are not detected as failures when they are produced. Such anomalous events can provide valuable information in order to prevent future failures and perform Root Cause Analysis, since such anomalies might be the precursor of a bigger failure that will occur in the system. In this sense, the extracted data have to be collected and stored in a centralized node before its analysis. By doing this, the analysis module will have access not only to the data extracted at runtime, but also to a historical registry about the data extracted from all the monitored devices.
To provide the described functionality, a Data Collector component will be implemented. This component is based on a plug-in architecture as depicted in Figure 18.

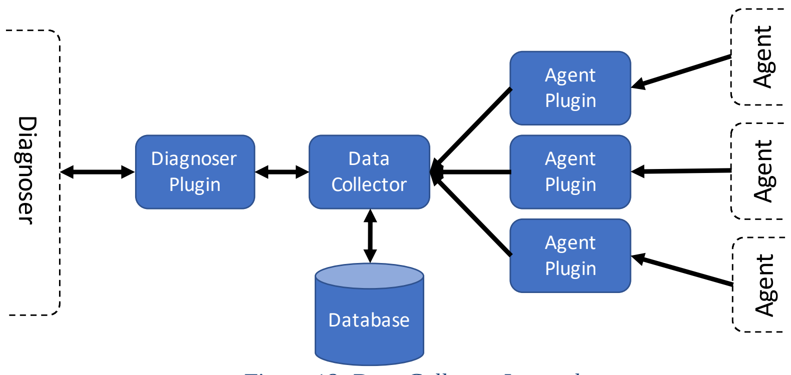


Figure 18: Data Collector Internals

The *Agent plug-ins* of the Data Collector have a double goal. On one hand they parse the specific data formats that used by each agent, and also extract the information from the messages in order to generate the base data that will be used for further analyses. On the other hand, they provide an interface to send the extracted data to the Diagnoser component.
Finally, the *Data Collector* will also be in charge of storing the data in a corresponding database for further reference. As a result, the Diagnoser will be able to request historical data from the Data Collector (using the interface mentioned above) that will help the RCA process.

4.1.1.3 Diagnoser

This is the core component of the RCA enabler. This component contains the Root Cause Analysis algorithms that will analyse the extracted information (and the historical log in the Data Collector Databases) and generate the corresponding reports of the Root Cause once an anomaly is detected. The internals of this module will contain, at a first iteration, a graph-based algorithm that will perform the Root Cause analysis. This algorithm uses a set of already-known issues, determining the similarity level of the observed malfunctioning against these known problems.
Using this database of known problems, the Diagnoser will be able to determine which is the most likely cause of the issue and inform the user – for instance the system administrator – about the findings. The details of the known problems database are given in the next section. In a similar way, the details of the algorithm that will be implemented are discussed in Section 4.1.2.
During a second iteration of the RCA module, the Diagnoser component will be enhanced further with machine learning capabilities.

4.1.1.4 Anomalous Pattern Database

As mentioned before, the Diagnoser component uses a database to check the existence of anomalies and maps them to their Root Cause.
This database will be composed of a set of patterns, which will be the baselines for the similarity computation performed by the Diagnoser component. Such patterns are composed of different data:

- A graph-based representation of affected nodes, where each node represents a device being impacted by the anomaly.
- A set of values of the monitored device’s variables. Each monitored device generates a flow of data that is stored in the Data Collector database. An anomaly pattern not only contains the ranges of values that can be used to identify an attack, but also the “expected” values of such variables (which might be related with the affected devices or not). The values serve to characterize the observed issue.
- A known Root Cause of the issue. Besides having a “list” of affected devices, it is also required to have an already-identified Root Cause.

Since this database is the reference that the Diagnoser uses to compare the monitoring parameters, it has to contain a set of known issues whose Root Cause analysis has been assessed by the system administrator. We aim to improve this process in later iterations by introducing Machine Learning algorithms in order to automatically learn new patterns, therefore expanding this database as more information becomes available.

4.1.2 Graph-based Root Cause Analysis

As mentioned before, the implemented RCA procedure will rely on a graph-based algorithm. The algorithm presented in this section describes the type of data that are used to compute the root cause, as well as the general line of the computation of the Root Cause of an issue.

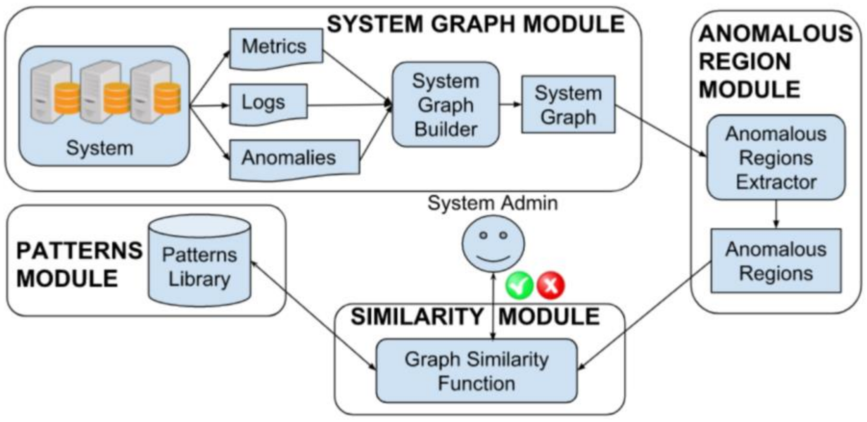


Figure 19: Root Cause Analysis Engine Architecture

Figure 19 shows the general architecture and the data flow of the Root Cause Analysis engine, which is composed of four principal sub-components. These components are described in the following sections.

4.1.2.1 System Graph Module.

The data extracted by the Agents/Probes – and potentially data stored in the Data Collector module feeds this module. It is in charge of transforming all the collected metrics (e.g., network activity, system logs, detected anomalies) in the system into a graph representation for a given time window. The generated graph can be understood as a snapshot of the current status of the monitored system, and is composed of the following elements:

- *Nodes*: Each node represents a device or a virtual container within the system running a specific part of the general process. Each node can have different features, which are explained later.
- *Edges*: Edges represent network communications between the nodes. In a similar way, edges can also have other features that will be explained later.
- *Attributes:* These elements will capture the values of the metrics collected. They can be of several types:
  - *Numerical:* Any value that can have a discrete or continuous value. Some examples here are the memory usage, CPU load, bandwidth, among others.
  - *Categorical:* Any value that can be interpreted as names or labels, such as “HTTP Server”.
  - *Ontological:* The ontological class used by the system to classify the hierarchy of this particular device. This information is used to generalize problems that are conceptually equivalent.
- *Anomaly Level:* An attribute to tag devices that are being affected by an anomaly. This field is set by any anomaly detection mechanism in order to identify the devices that are affected by a particular anomaly. This field is later used by the algorithm to compute a sub-graph of the System Graph that contains anomalies.

All the aforementioned information is used to create the System Graph following the procedure explained later in Section 4.1.2.6.

4.1.2.2 Anomalous Region Module

The snapshot of the system generated by the previous module will be analysed by the Anomalous Region Module, in order to construct a list of affected graph components. This task will be performed with the aid of a function that will help determine the list of potentially affected nodes of the graph. In particular, we provide a function that select nodes and edges that show an anomaly level higher than a given threshold, which will hardly depend on the system under analysis.
It is important to remark that this module extracts a subgraph of the System Graph previously generated and, therefore, also stores its nodes, edges and attributes. A *weighting function* is in charge of assigning weights to each of the elements aforementioned (nodes, edges and attributes) according to a previously-

defined criteria. These weights represent the importance of that element in the anomaly potentially detected. For example, if a rise in the CPU usage is observed, the node with the detected anomaly would have its "CPU load" attribute with a higher weight compared to other parts of the graph.
The outputs of this function (the selected subgraph and the weights) are the main input to the Similarity Module that will match it with the already-troubleshooted anomalies stored in the Patterns Module.

4.1.2.3 Patterns Module

This module corresponds to the "Anomalous Pattern Database", which was described in Section 4.1.1.4. The Patterns Module contains a set of sub-graphs that represent the anomalies and Root Causes that have already been identified by the system administrator. These patterns will be used as templates to match the information obtained from the system.
The user (e.g. system administrator) is capable of accessing this database and reviewing the patterns, and can also update the stored information, such as:

- Changing weights: As mentioned before, nodes have a weight that represents the importance of each node in the system. In a similar way, it is possible to assign weights to edges and attributes to change their importance when computing the similarity level. The user can change such weights in order to give more importance to other nodes, edges or attributes.
- Cause label: This is the main reason of the anomaly. This field is not restricted to show a device name or tag but can also contain a set of steps used to resolve the encountered anomaly. In a supervised learning scheme, this field can be used to automatically compute the weights using, for example, a common tag for similar anomalies to be able to find correlations among them.

4.1.2.4 Similarity Module

The Similarity Module is in charge of determining a possible match between the anomalous region computed by the Anomalous Region Module and one or more of the pattern anomalies stored in the Patterns Module. In this sense, the Similarity Module will obtain information from both of these modules (i.e., graph A from the Anomalous Region Module and graph B from the Pattern Library Module) and compute a *similarity score* based on an inexact matching function that considers the similarity distance between two multi-attributed-weighted graphs. Following this procedure, the algorithm's output will be two-folded:

- The *Similarity Score:* A value ranging from 0 to 1, where 0 means that graphs A and B are completely different and 1 means that they are complete identical.
- A *Mapping of Nodes:* A relationship describing which elements of the graph A match the elements of the graph B.

Each time a new anomalous region arrives, the Similarity Module will try to match it with one or more patterns stored in the database. Even though it is possible to start by matching all the patterns stored in the database, a better approach will be to discard all the patterns where the devices are not concerned with anomalies. This will avoid trying to match with patterns that do not cover the observations.
In case no match is found, the anomality will be treated as a new pattern, and it will be presented to the user – the system administrator – for validation. In case one or more pattern(s) is(are) found, they will be shown to the user as a ranked list with their respective Root Causes, so that they can accept or reject the matchings.
The feedback provided by the user will enrich the matching algorithm in two ways: the validation of the matching tells the algorithm that the matching is correct and makes it trustworthy, or the rejection indicating that the comparison did not capture the underlying information of the graphs. In order to adjust the patterns, the system administrator will have to provide further information related to the weights and/or the concerned root cause. This feedback will be used to adjust the weights of elements stored in the patterns database, in order to enhance the matching for improving the effectiveness of the analysis.

4.1.2.5 Graph Similarity Algorithm

The Similarity Module uses a graph-based algorithm to compute the *similarity score* between two given graphs. This section gives details about the algorithm used to compute such score.

4.1.2.5.1 General Definitions

We understand a graph as a 5-tuple:

$$G = (V, E, att, C, w)$$

where V is a set of vertices, $E \subseteq V \times V$ is a set of edges, att is a function such that $att(V \vee E) = A$, A is the list of attributes a represented by tuples $a = \{val, weight, c \subset C)$, val is the value of the attribute, $weight$ is the weight of the attribute, and c is the context of the attribute. C is the graph context, and it is used to explain the attributes that are included in the graph and to give a context in which to compare its values. Finally, w is a function such that $w(x)$ returns the weight of an edge or node within a graph, or the weight of an attribute within is node or edge.

4.1.2.5.2 Problem statement

For stating the problem we have to consider two graphs $G_1 = (V_1, E_1, att, C, w)$ and $G_2 = (V_2, E_2, att, C, w)$, and two injective functions $m_n = V_1 \to V_2$ and $m_e = E_1 \to E_2$, where the first function return the matched node from G_1into G_2 and the latter returns the same for the edges. Given this context we want to fund the mapping that maximises the similarity between the two graphs:

$$S = \frac{\sum_{v \in V_1} \left(w(v) + w\big(m_n(v)\big)\right) \cdot sim\big(v, m_n(v)\big) + \sum_{e \in E_1} \left(w(e) + w\big(m_n(e)\big)\right) \cdot sim\big(v, m_n(e)\big)}{\sum_{v \in V_1} w(v) + \sum_{v \in V_2} w(v) + \sum_{e \in E_1} w(e) + \sum_{e \in V_2} w(e)}$$

$$arg\ max_{m_n, m_e}$$

Where $sim(x_1, x_2)$is a function that represents how similar are two edges or nodes using a numeric value between 0 and 1.

The formula presented above summarises that the similarity S between two graphs is the weighted average of the similarities between the optimally mapped elements of the graph through the functions m_n and m_e. In the formula we notice that the elements that have a bigger weight of importance in the graph will contribute more to the global similarity score.

The minimization problem stated in the equation above can be solved by any state-of-the-art technique. For the ENACT framework we will use a “hill climbing” approach, in order to reduce the computational complexity with the trade-off of finding a local maximum instead of a global one.

At this point, we have stated the optimization problem to find the *similarity level* of two graphs. However, we still need to clearly define how to compute the similarity level between two nodes or edges. The next section will give details on how to measure this similarity.

4.1.2.5.3 Similarity Between Two Nodes or Edges

Given two edges or nodes $x_1$and x_2, the similarity between them is given by:

$$sim(x_1, x_2) = \frac{\sum_{a_1 \in att(x_1), a_2 \in att(x_2)} (w(a_1) + w(a_2)) \cdot sim(a_1, a_2)}{\sum_{a_1 \in att(x_1), a_2 \in att(x_2)} (w(a_1) + w(a_2))}$$

where $sim(a_1, a_2)$ is the similarity between two attributes of the element, which we will explain in the next section. Similarly, to the previous criterion, the similarity between two edges or nodes is also a weighted average between all the shared attributes of both elements. Moreover, it is also important to remark that the weights $w(a_1) + w(a_2)$ is used to change the comparison calculation. Indeed, if a higher weight is assigned to an attribute it will increase even more the similarity between the two nodes or edges involved. This is a very useful feature for including domain knowledge into the anomalous graph.

4.1.2.5.4 Similarity Between Two Attributes

In the last section we defined the similarity between two nodes or edges as a function of the similarity between two attributes. Depending on the type of attribute we need to consider the context of the attribute before comparing them. Three different context type will be considered:

1. *Categorial Context*: An attribute a_1 with a categorical context will have a label as its value. For this reason, we use exact equality of labels to compare two categorial attributes:
$$sim(a_1, a_2) = \begin{cases} 1\ if\ a_1\ =\ a_2 \\ 0\ otherwise \end{cases}$$
2. *Numerical Context*: A numerical context is represented by a tuple $c_{numerical} = \{min, max\}$. When comparing two attributes with numerical context we use the function:

$$sim(a_1, a_2) = 1 - \frac{|a_1 - a_2|}{max - min}$$

which gives a measurement about how much the value differ with respect to the maximum difference.

3. *Ontological Context*: An ontological context is represented by a tree data structure. To compare two attributes with an ontological context, we use a modification of the Wu and Palmer similarity metric [15]. Given two attributes C_1 and C_2 two concepts on the ontology and let be C its closets common ancestor, then

$$sim(C_1, C_2) = \frac{2 \cdot d(C)}{d(C_1) + d(C_2)}$$

where $d(x)$ is the number of nodes to traverse from the root of the taxonomy to the concept x, which always includes the concept x itself, therefore $d(x) \geq 1$.

The proposed approach is not restricted to comparing two attributes, and it allows to define new contexts and new metrics to enrich the comparison process. Despite this flexibility, it is required to perform a theoretical analysis to correctly identify the context and types of attributed that will be used. Such analysis will hardly depend on the domain on which the. Root Cause Analysis engine will be used, and it is out of the scope of this document.

4.1.2.6 Building the System Graph via Monitoring

In Section 4.1.2.1 it was described the principal elements that compose the System Graph. This section extends the analysis with the procedure used to construct the System Graph using monitoring solutions. The objective is to expose how to extract the attributes of the vertexes and edges, as well as how to define the connections between different elements of the graph.

4.1.2.6.1 Nodes

Any element that is active within the system will be added as a node. This means that any element on the system that can answer any type of request (web servers, databases, routers, among others) will be modelled as a node on the graph.

From the node elements, we need to extract the metrics. Since the goal is to build a temporal snapshot of the system, we will store the metrics in a multi-dimensional data series. This means that each extracted metric will be stored as a tuple $(ts, id, vallue, labels)$, where ts is a timestamp, id is an identifier for the system element, $value$ is the value for the extracted metric and $labels$ is a set of key-value pairs representing particular dimensions of that metric.

To build the whole System Graph for a given time interval we will simply aggregate all the tuples coming from different agent plugins. With this information we will create a node on the graph for each id detected on the tuples. Later, we will add the attributes of the fields $(value, label)$ as attributes of such nodes. This process will ensure that each instrumented software will be modelled as a node on the System Graph and contain all the attributes reported by the instrumentation software.

4.1.2.6.2 Edges

Once we have created the nodes, we need to add the edges of the system. This process is performed by observing the communication between the nodes already observed, using network monitoring techniques such that Deep Packet/Flow Inspection (DPI/DFI). Using these technique, it is possible to generate a tuple of the observed flows in the format $(ts, id, cip, sip, io\_dir, bytes)$, where ts is the timestamp of the observation, id is the identifier of the container where the communication occurred, cip is the client IP address, sip is the server IP, io_dir is a value representing the direction of the communication and $bytes$ is the amount of information transferred measured in bytes.

Similarly, to the previous process, we use these tuples to aggregate the information within a given time window and compute the final statistics for the edges, which will be used as the attributes of the edges. These aggregated tuples will be used to compute the attribute values of the edges. It is important to remark that despite that the communication between two nodes can be in both directions, we want to keep such statistics apart, so we will add a directed edge for each sense of the communication, using the

cip, sip and io_dir attributes to determine the sense of the edge to add; one for the communication from A to B and another one for the communication in the other sense.

Following the building approach previously described before, we will generate a directed graph that contains all the nodes, edges and their respective attributes.

4.2 Prototypical implementation

At this point, a prototypical implementation of the described algorithms is currently being developed. Once ready, it will be made available through the ENACT distributions channels. Further details about this implementation will be given in D3.3 – the next version of this deliverable.

5 Adaptation enactment as support for self-adaptation

5.1 Conceptual design

A detailed description of GeneSIS (a.k.a. the orchestration and deployment enabler) can be found in deliverable D2.2. In this section we focus on presenting how GeneSIS supports dynamic adaptation in the deployment of a Smart IoT System (SIS).

From a deployment model specified using the GeneSIS Modelling language (please refer to D2.2 for details about the GeneSIS modelling language), the GeneSIS deployment execution engine is responsible for: *(i)* deploying the deployable artefacts (i.e., installing and configuring the actual software binaries, code, scripts, etc. to be deployed on the target), *(ii)* ensuring communication between them (i.e., deployed components are configured to ensure communications between them), *(iii)* provisioning cloud resources (i.e., cloud resources need to be created before a software can be deployed on it. For insance, a virtual machine has to be created and started before a software component can be deployed on it), *(iv)* monitoring the status of the deployment (i.e., monitoring if hosts are still reachable and if software component are still running), and *(v)* adapting a deployment (i.e., modifying how the SIS is deployed).

5.1.1 Overall architecture

As depicted in Figure 20, the GeneSIS execution engine can be divided into two main elements: *(i)* the facade and *(ii)* the deployment engine.

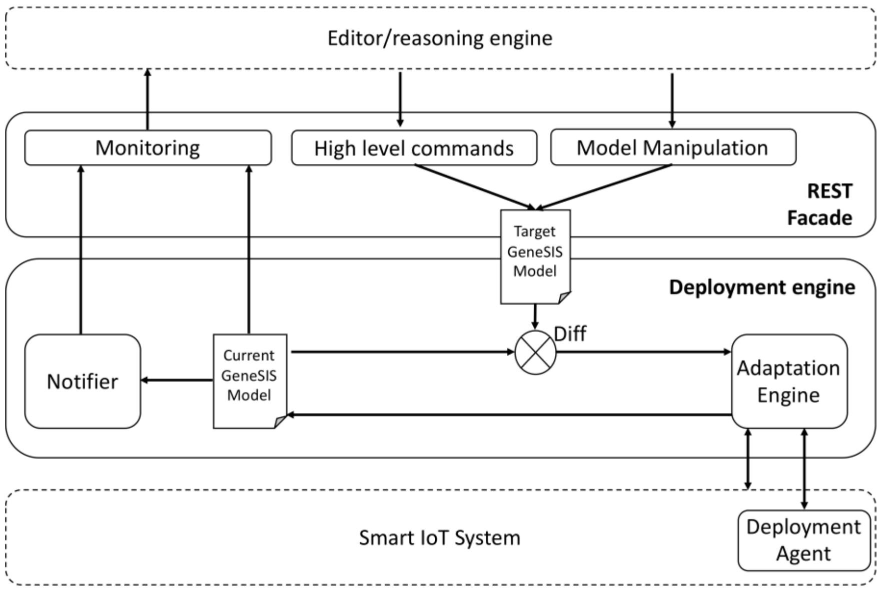


Figure 20. Architecture of the GeneSIS execution environment

The facade provides a common way to programmatically interact with the GeneSIS execution engine via a set of three APIs.

- The monitoring API offers mechanisms for remote third parties (*e.g.*, reasoning engines) to observe the status of a system. Third parties can either consume the whole GeneSIS model of the running system enriched with runtime information or subscribe to a notification mechanism.
- The high-level commands API exposes a pre-defined set of high-level commands that avoid direct manipulation of the models (*i.e.*, the model is automatically updated when the command is triggered). In the future, this API will include a migrate command that supports the migration of an `InternalComponent` from one host to another.

- Finally, the model manipulation API provides the ability to load a new target deployment model. In the future, it is also planned to provide support for the atomic MOF-level[3] modifications of the deployment model.

GeneSIS follows a declarative deployment approach. From the specification of the desired system state, which captures the needed system topology, the deployment engine computes how to reach this state. It is worth noting that the deployment engine may not always compute optimal deployment plans.

The GeneSIS deployment engine implements the Models@Run-time pattern to support the dynamic adaptation of a deployment with minimal impact on the running system. Models@Run-time [models@runtime] is an architectural pattern for dynamic adaptive systems that leverage models as executable artefacts that can be applied to support the execution of the system. Models@Run-time enables to provide abstract representations of the underlying running system, which facilitates reasoning, analysis, simulation, and adaptation. A change in the running system is automatically reflected in the model of the current system. Similarly, a modification to this model is enacted on the running system on demand. This causal connection enables the continuous evolution of the system with no strict boundaries between design- and run-time activities.

Our engine is a typical implementation of the Models@Run-time pattern. When a target model is fed to the deployment engine, it is compared (see Diff in Figure 20) with the GeneSIS model representing the running system. Finally, the adaptation engine enacts the adaptation (*i.e.*, the deployment) by modifying only the parts of the system necessary to account for the difference and the target GeneSIS model becomes the current GeneSIS model. Finally, the deployment engine can delegate part of its activities to deployment agents running on the field (see Section deliverable D2.2 for more details.).

5.1.2 The GeneSIS Comparison Engine

The comparison engine is at the core of the GeneSIS support for dynamic adaptation. The inputs to the *Comparison engine* (also called *Diff*) are the current and target deployment models. The output is a set of lists of changes (that can directly be mapped to actions) representing the required evolutions to transform the current model into the target model. The different lists are presented in Table 1. The evaluation of this list by the GeneSIS execution engine results in: *(i)* modification of the deployment and provisioning topology and *(ii)* deployment of the evolved SIS.

Table 1. Types of output generated by the Comparison engine

List	Content	Used for
list_removed_ infrastructure_components	List of Infrastructure components removed in the target model compared to the current one.	Terminate or stop a cloud service
list_added_ infrastructure_components	List of Infrastructure components added in the target model compared to the current one.	Provision and configure a cloud service and check availability of the other type of infrastructure components (i.e., the ones that cannot be provisioned)
List_of_removed_ software_components	List of software components components removed in the target model compared to the current one.	Remove the internal component instance from its current host
list_of_added_ software components	List of software components added in the target model compared to the current one.	Deploy the internal components and configure the external ones

[3] https://www.omg.org/ocup-2/documents/Meta-ModelingAndtheMOF.pdf

list_of_added_ hosted_components	List of sofware components that are hosted on another component added in the target model compared to the current one.	Deploy the internal component instance from its current host
list_of_removed_ communications	List of communications removed in the target model compared to the current one.	Disconnect the endpoints of the communication
list_of_added_ communications	List of communications added in the target model compared to the current one.	Configure the endpoints of the communication
list_of_removed_ containments	List of containments removed in the target model compared to the current one.	Check validity of the deployment
list_of_added_ containments	List of containments added in the target model compared to the current one.	Check the validity of the deployment
list_of_added__ deployer	List of communications with the property 'isDeployer' set to true (i.e., meaning that a deployment agent will be required) added in the target model compared to the current one.	Check the list of components that need to be deployed by a deployment agent and deploy them.

The comparison engine processes the entities composing the deployment models in the following order: infrastructure components, software components, communications, containments, on the basis of the logical dependencies between these concepts. In this way, all the components required by another component are deployed first.

For each of these concepts, we compare the two sets of instances from the current and target models. This comparison is achieved on the matching of the various properties of the instances, of their types, and on their logical dependencies with other components. More precisely, at the current moment, the following changes can lead to mismatch between two instances of components:

- *Properties*:
    - Name has changed
    - IP has changed
    - Resource has changed
    - Open ports have changed
- *Logical dependencies*:
    - The component is not anymore on the same host (can be because the host itself has been updated)
    - The other endpoint of a communication with the mandatory property set to true has changed

Similarly, at the current moment the following changes can lead mismatch between two instances of communications or hosts:

- *Properties*:
    - The source or the target of a communication has changed.

For each unmatched instance from the current model, the instance is added to the corresponding list of removed instances. Similarly, for each unmatched instance from the target model the instance is added to the corresponding list of added instances.

As said before, from the list of changes identified by the comparison engine, the GeneSIS execution environment derives (typically a one-to-one mapping) an adaptation plan, which, in turn, is used to enact the adaptation of the system (i.e., of the deployment of the system). This process can be triggered by providing the GeneSIS execution environment with a new deployment model via its Façade. In the following we detail the different means to interact with the GeneSIS execution environment.

5.1.3 Interacting with the GeneSIS execution environment

The GeneSIS execution engine currently supports five types of commands accessible via its REST-based interface:

Method	Resource	Content-type	Description
GET	/types	Response: application/json	to retrieve all the component types registered in the execution engine
POST	/deploy	Response: application/json Parameter: application/json	to provide the engine with a new deployment model and trigger a re-deployment
GET	/logs	Response: text/plain	to retrieve all the logs from the execution engine
GET	/model	Response: application/json	to retrieve the current deployment model
GET	/model_ui	Response: application/json	to retrieve the current deployment model including its graphical representation

More commands will be added later in particular to: *(i)* manipulate the current model without providing a complete new target model, and *(ii)* to trigger high level adaptation actions such as migrating one component from one host to another.

For its monitoring interface to send notification to third parties, GeneSIS embed a MQTT message broker. MQTT has been selected because as it follows the Publish-Subscribe pattern as it is scalable and offers great space, time, and synchronization decoupling [16]. At the current moment, GeneSIS exploits two topics in the message queue to distributes its messages:

- **Status:** this topic is used to distribute messages about the status of the different components that form the SIS. Possible status are: error, running, and under_configuration. Messages are written in JSON as follows:

```
{
    node: <name of the component>,
    status: 'error'
}
```

- **Notification:** this topic is used to distribute messages describing the status of the deployment process. Messages are written in JSON as follows:

```
{
    action: <name of the deployment action>,
    status: ‘success’
    message: ‘Component A has been successfully deployed’
}
```

5.2 Prototypical implementation

GeneSIS is available as an open-source project[4] implemented in JavaScript using npm as the build tool. The current code base consists of around 20 000 lines of code. The GeneSIS models are represented in plain JavaScript. These models can be serialized in JSON. The different Node-RED nodes (in Node-RED, a node corresponds to a software component) implemented to build deployment agents are also available as a set of open-source projects[5].

The GeneSIS GitLab repository includes:

- A readme detailing how to setup and start GeneSIS from: *(i)* Git, *(ii)* the GeneSIS official Docker image, and *(iii)* the Docker Build file
- A set of examples of deployment models: https://gitlab.com/enact/GeneSIS/tree/master/docs/examples
- A set of six tutorials gradually explaining how to use GeneSIS is available at https://gitlab.com/enact/GeneSIS/tree/master/docs/tutorial and include instructions for:
    - Deploying a single instance of Node-RED.
    - Deploying a single ThingML Component.
    - Deploying two instances Node-RED.
    - Deploying via Ansible.
    - Deploying via SSH.
    - Deploying multiple nodes including a nodes requiring a deployment agent.

[4] https://gitlab.com/enact/GeneSIS
[5] https://gitlab.com/enact/docker-node-red-thingml
https://gitlab.com/enact/node-red-contrib-arduino
https://gitlab.com/enact/node-red-contrib-docker-compose
https://gitlab.com/enact/node-red-contrib-thingml-compiler
https://gitlab.com/enact/node-red-contrib-thingml-preparedeploy-docker

6 Conclusion & next steps

WP3 aims at developing enablers for the operational part of the DevOps process. WP3 will provide enablers that provide IoT systems with capabilities to (i) monitor their status, (ii) indicate whether they behave as expected or not, (iii) identify the origin of the problem, and (iv) automatically perform typical operation activities (including self-adaptation of the systems). The main focus of the document is the studies performed on the state-of-the-art on the following topics related to the aforementioned capabilities: online learning, behavioural drift analysis, root-cause analysis and the support for self-adaptation of SIS.

Based on the conceptual designs described in D3.1 we revised the conceptual designs in D3.2 and provided prototypical implementations for all tools of WP3, except RCA which is ongoing. These first versions of the tools will serve as a baseline for the final tools delivered during the project. Next steps are the proper implementation of APIs and focus on the interactions with other tools. Also, some tools need to be linked to use-cases more related to the core of the ENACT project.

7 References

1. Kephart, J.O. and D.M. Chess, *The vision of autonomic computing.* Computer, 2003(1): p. 41-50.
2. De Lemos, R., et al., *Software engineering for self-adaptive systems: A second research roadmap*, in *Software Engineering for Self-Adaptive Systems II*. 2013, Springer. p. 1-32.
3. Salehie, M. and L. Tahvildari, *Self-adaptive software: Landscape and research challenges.* ACM transactions on autonomous and adaptive systems (TAAS), 2009. **4**(2): p. 14.
4. Iglesia, D.G.D.L. and D. Weyns, *MAPE-K formal templates to rigorously design behaviors for self-adaptive systems.* ACM Transactions on Autonomous and Adaptive Systems (TAAS), 2015. **10**(3): p. 15.
5. Busoniu, L., et al., *Reinforcement learning and dynamic programming using function approximators*. 2017: CRC press.
6. Lewis, F.L. and D. Liu, *Reinforcement learning and approximate dynamic programming for feedback control*. Vol. 17. 2013: John Wiley & Sons.
7. Sutton, R.S. and A.G. Barto, *Reinforcement learning: An introduction*. 2018: MIT press.
8. Dietterich, T.G., *Machine Learning.* ACM Comput. Surv., 1996. **28**(4es): p. 3.
9. Sutton, R.S., et al. *Policy gradient methods for reinforcement learning with function approximation*. in *Advances in neural information processing systems*. 2000.
10. Rocher, G., J.-Y. Tigli, and S. Lavirotte. *On the behavioral drift estimation of ubiquitous computing systems in partially known environments*. in *Proceedings of the 13th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*. 2016. ACM.
11. Schulman, J., et al., *Proximal policy optimization algorithms.* arXiv preprint arXiv:1707.06347, 2017.
12. Adadi, A. and M. Berrada, *Peeking Inside the Black-Box: A Survey on Explainable Artificial Intelligence (XAI).* IEEE Access, 2018. **6**: p. 52138-52160.
13. Khreich, W., et al., *A survey of techniques for incremental learning of HMM parameters.* Inf. Sci., 2012. **197**: p. 105-130.
14. Engel, P.M. and M.R. Heinen. *Incremental Learning of Multivariate Gaussian Mixture Models*. 2010. Berlin, Heidelberg: Springer Berlin Heidelberg.
15. Wu, Z. and M. Palmer, *Verbs semantics and lexical selection*, in *Proceedings of the 32nd annual meeting on Association for Computational Linguistics*. 1994, Association for Computational Linguistics: Las Cruces, New Mexico. p. 133-138.
16. Eugster, P.T., et al., *The many faces of publish/subscribe.* ACM computing surveys (CSUR), 2003. **35**(2): p. 114-131.