# QoS for Dynamic Deployment of IoT Services

Isakovic Haris*, Luis Lino Ferreira †, Irmin Okic*,
Adam Dukkon*, Zlatan Tucakovic*, Radu Grosu*
Technsiche Universität Wien*, Vienna, Austria
CISTER Research Center†, ISEP Polytechnic Institute of Porto, Porto, Portugal
* name.surname@tuwien.ac.at, † llf@isep.ipp.pt

2021 22nd IEEE International Conference on Industrial Technology (ICIT) | 978-1-7281-5730-6/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICIT46573.2021.9453670

***Abstract*—This paper introduces RVAF, a runtime verification (RV) extension of the Arrowhead Framework (AF) with container-based service-deployment and runtime-enforcement of a desired quality of service (QoS). AF is a service-oriented middleware architecture for IoT-applications, consisting of a set of core and auxiliary services and systems, respectively. The QoS manager (QoSM) is one AF's most important auxiliary systems, which can be used to guarantee the application's QoS for a wide set of parameters. In RVAF the QoS offered to a particular IoT-application is specified in signal temporal logic, and is continuously monitored by the RVAF-QoSM. In case of an imminent violation, RVAF automatically initiates a container-based reconfiguration, which is ensured to maintain the desired QoS. RVAF is beneficial to large IoT-applications, where the use of continuous-integration and continuous-deployment tools, is not only a recommended practice but also a necessity. Moreover, the use of RVAF is advantageous both during the development of an IoT application, and after its deployment. We describe the architecture of RVAF, provide its formal underpinning, and demonstrate the usefulness of RVAF supported by an industrial IoT application. The main contribution of this work is to show what it takes to incorporate RV concepts into modern SOA frameworks supporting the development of IoT applications.**

## I. Introduction

Industrial digital systems (IdS) were originally designed with a mission-oriented focus in mind. As a consequence, IdS were homogeneous pieces of software, most often unable to adapt to the changes occurring during their evolution, such as the IdS' subsequent integration with management, analysis, or financial systems, respectively. In order to alleviate these problems, the design of IdS is now increasingly adopting the IoT paradigm, which is one of the major pillars of the industrial-digitization initiative to optimizing production chains [1].

The arrowhead framework (AF) is a middle-ware enabling the service-oriented approach (SOA) to the development of IoT applications [2]. SOA facilitates the construction of generic services, that are later on integrated in various applications, as application-specific services. SOA thus greatly enhances the reuse of services, and their integration into fine-grained functionalities.

AF includes a set of core services, such as service discovery, orchestration, and authentication, which in turn support the interaction between application services, such as sensor-reading or data-storage services [2]. Each service is a part of a system-of-systems (SoS) and acts as a completely separate software entity. As such, services are developed, operated, and maintained by different groups, organizations, or individuals. Ensuring a seamless integration between development and maintenance on the one side, and the quality of service (QoS) during operation on the other side, are overlapping tasks.

This paper introduces RVAF, a continuous integration and deployment, RV-extension of AF. RVAF supports the continuous service integration/deployment in a container-based fashion, and the runtime enforcement of a desired quality of service (QoS) through adaptation. In RVAF the QoS guaranteed to a particular IoT-application is specified in signal temporal logic (STL), and continuously monitored by the QoS manager (QoSM). As shown in Figure 4, in case of an imminent violation, RVAF automatically initiates a container-based reconfiguration, which is ensured to maintain the desired QoS. RVAF is beneficial to large IoT-applications, where the use of continuous-integration and continuous-deployment tools, is not only a recommended practice but also a necessity. Moreover, RVAF use is not only beneficial after the deployment of an IoT application, but also during its development. In addition to describing the architecture of RVAF and providing its formal underpinning, we give proof of concept of the usefulness of RVAF based on a real-life, industrial scenario.

The rest of the paper is organized as follows. In Section II we introduce AF, followed by related work in QoS on Arrowhead. In Section IV we discuss RVAF architecture and in Section V its formalization. In Section VI we present our experimental results on a conveyor-belt scenario. Finally in Section VIII we draw our conclusions and discuss future work.

## II. AF: Arrowhead Framework

AF is the result of a set of EU projects in which SOA principles have been applied to IoT and Industrial IoT applications, respectively. As the main result of the Arrowhead projects, the AF continued its development in an independent fashion, and it is now being used in multiple industrial installations. It is further developed in other projects, like for example, the Arrowhead Tools project.

AF has what is needed to design, implement and deploy an Arrowhead SOA-compliant system. The objective is to allow all users to work in a common and unified framework, thus enabling a high level of interoperability. The AF includes three core services and systems [2], where the Orchestrator plays a key role in defining which application services are connected with each other, and the QoSM guaranteeing adequate levels of QoS for each connection.

978-1-7281-5730-6/21/$31.00 ©2021 IEEE

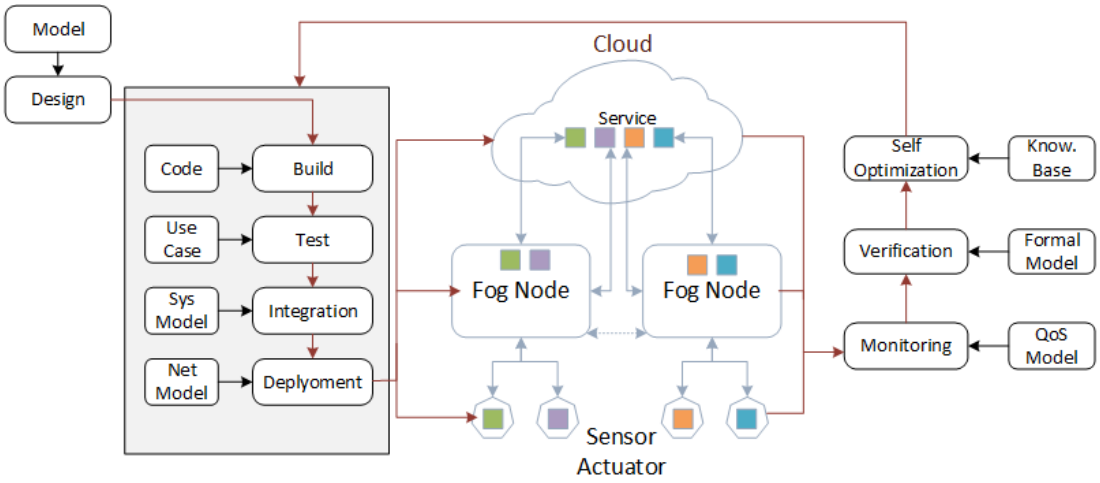


Fig. 1. RVAF-supported and IoT-based full-stack system architecture

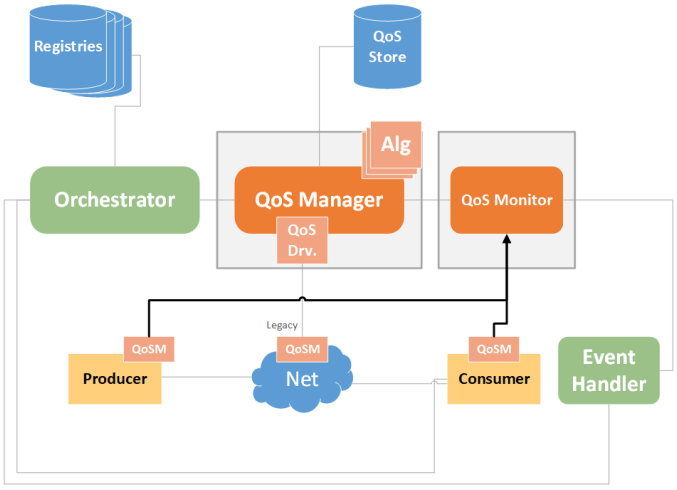


Fig. 2. QoS Manager Architecture

The AF builds upon the local-cloud (LC) concept, where local automation tasks are encapsulated and protected from outside interference. Application services usually communicate with services within the LC (intra-cloud orchestration), except when they communicate through the Gatekeeper Service with other AF-compliant LCs (inter-cloud orchestration). Each LC must contain, at least, the three mandatory core systems: Registry, Authorization, and Orchestration. They allow to establish connections between AF-application services. The core systems are accompanied by optional supporting systems, that further enhance the capabilities of an LC. The Event Handler is one of these systems, capable of implementing a producer/consumer communication pattern on an SOA logic.

## III. Related Work

As shown in Figure 2, the QoS Manager (QoSM) in Arrowhead interacts with the Orchestrator and the QoS-Monitor, to ensure that connections between application-service instances, respect given QoS requirements. The QoSM handles information regarding the network topology, device capabilities, QoS requirements, etc, residing in the QoS Store (and other databases of the AF), accessible through an SOA interface. The management of the QoS is mainly concerned with the reservation of communication and computational resources. Their relevant information is maintained in a QoS Store. The QoSM is able to determine if the requested QoS can be granted or not, and configure the involved devices and active network elements (e.g., router and switches), to grant a QoS request.

The elements that can be configured by the QoSM include traffic-smoothing filters (on the output of service producers or consumers), traffic parameters (like traffic priority), and delivery guarantees of message-oriented middle-ware with QoS capabilities (like DDS [3], RabbitMQ [4] or XMPP [5]).

The Alg module contains different algorithms for QoS verification (examples will be given Section V, as Algorithms 1 and 2, respectively). The configuration of network actives and nodes is done by the QoS Drv module. This provides the drivers for interacting with custom or proprietary protocols. Some applications might need to know the current status of the system, in order to be able to adapt to changing conditions. This is achieved by monitoring the network status with the QoS Monitor, and informing the interested parties (e.g., application services in charge of ensuring the robustness of the system) through the Event Handler. A detailed description of the QoS support in AF can be found in [6].

## IV. RVAF: System Architecture

An IoT architecture can be split in three distinct scopes of operation (see Figure 1): the cloud, the fog (edge), and the swarm (sensors/actuators) [7]. Moreover, sensors and actuators provide us with the ability to extract data from the physical environment and to actuate upon its state. In the fog specific tasks are performed such as data aggregation, filtering, or real-time control. In the cloud we have ”unlimited” amount of storage and the ability to perform large-scale computation

within conceivable time. Development of IoT systems that spread over multiple scopes of operation is a challenging task. It involves several levels of development such as system or infrastructure services; user application services; and specification, configuration and validation services. We describe an approach on increasing QoS of IoT systems and making them more dependable in general by using it's own existing infrastructure for continuous integration and deployment (CI/CD). RVAF system can monitor IoT services, detect faults and errors, ensure service stability, reduce downtime and increase overall availability.

### A. CI/CD: Continuous Integration and Deployment

Automation of system development and operation is achieved through a continuous cycle of development, integration, and deployment (CI/CD). This cycle ensures an automated evolution from code to full applications, and an automated maintenance and optimization, where newly built code is automatically integrated with the rest of the system and deployed to a target.

If the system is malfunctioning, the target code will provide feedback to the developer, such that potential bugs and errors in the system are corrected. CI/CD is therefore an essential component of distributed and heterogeneous systems, such as a full-stack IoT-based systems. Being part of the infrastructure that supports the development and maintenance of IoT platforms and applications, CI/CD contributes to a reduction of development time and costs, and ensures a higher reliability and robustness of a system.

Arrowhead is an open-source project [8] and it was mainly using a centralized development method together with custom designed manual deployment methods implemented by the end user. In this paper we also introduce an automated continuous integration and deployment system for Arrowhead core services that can be implemented with the CI/CD system at the application level. It provides a novel approach to integrate user level services with core services in a fully automated way. This ensures that not only user level services, but also, system level services are kept updated. As shown in Figure 3, RVAF services are deployed through the use of containers, such as the ones provided by the Docker Swarm or Kubernetes. Each service is installed in a single container with all its required dependencies. Containers are pre-built and stored in a container-image registry, the Docker Registry, which is maintained automatically by the continuous integration system (CI). In the example shown in Figure 3, the CI is implemented using the Jenkins CI tool [9]. It contains a build server for Arrowhead core services and user level services. Each service needs authorization certificates to join an Arrowhead local cloud, each service needs to be configured and provide endpoint information to the rest of the system. This configuration process is performed using Infrastructure as Code and deployed by Jenkins pipelines. The actions of integration and delivery to the Docker Registry are automatically triggered by major version commits in Arrowhead code repository.

An RVAF continuous-deployment system (CD), connects the images of services with their respective targets (fog nodes, cloud virtual-machines), and deploys them accordingly. The CD is responsible for updating the containers in case a service changes in the Docker Registry. It allows to update services on demand, to rollback services if necessary, or to remotely control services or clusters of services. Container orchestration is observing container parameters such as CPU and memory usage, however this is not reflecting a functional health state of a service within the container. To ensure QoS on both levels it is necessary to observe certain parameters within the service that could affect its ability to deliver correct functionality.

### B. RM: Runtime Monitoring

A further benefit of RVAF container-based deployment, is the ability to monitor at runtime (RM) each container, in terms of resources or application-specific parameters. RM is performed either on a target node, or remotely. As each container is identified with a single service, one can single out properties of a specific service, and monitor it individually. RM thus allows to observe the behavior of each service, and evaluate the health of the entire system.

In the context of RVAF-supported full-stack IoT applications, we have to consider large scale multi-dimensional data streams, with multiple sources. To fully monitor such distributed applications, all sub-agents need to be observed simultaneously. As an RM infrastructure we therefore chose to integrate the Prometheus monitoring tool [10] with the QoSM. This combination supports real-time, RM capabilities on either a local machine, or in the cloud. RVAF uses soft real-time monitors wrapped around PromQL [10] queries to observe metrics from individual containers. The data collected from a service or container is exported into time series database, whereof it can be aggregated, filtered or monitored raw.

These features perform run-time monitoring and run-time verification of the individual services, metrics within the services, containers and whole clusters. It provides an additional and novel approach to ensure QoS in industrial IoT systems. It increases fault-tolerance, and can even be used to ensure resilience to attacks on an IoT system.

### C. SO: Self-Optimization

The IoT applications considered are heterogeneous hardware and software systems. This and the fact that they are implemented at multiple scopes of operation, makes them very difficult to develop, manage, and operate. Thus it is important to design these applications with basic principles of autonomic computing, such as, self-adaptation, self-organization, self-configuration, self-protection, self-healing, self-description, self-matchmaking, and self-energy supply [11].

We identify self-optimization (SO) as the most relevant property for the QoS of IoT frameworks. SO is the ability of a system to take corrective measures, in order to maintain the optimal usage of constrained resources [11]. The concept of QoS management in AF, as described in [12], detects violations and informs the other core services about the

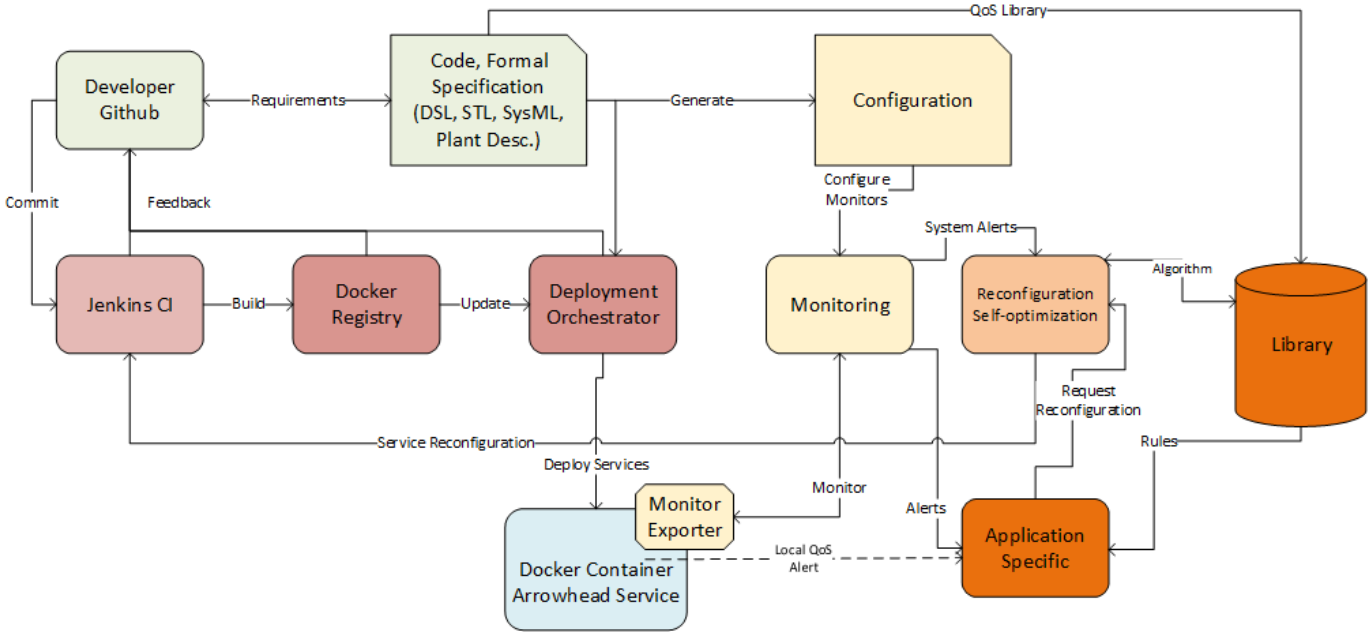


Fig. 3. RVAF container-based service deployment

violation, it is the responsibility of these core services to take any corrective, or containment actions.

In RVAF we extend the violation detection/notification ability, with the ability to take corrective actions, on services, Arrowhead clouds, or target nodes in general. This ability falls under the notion of SO, as defined in [11]. The RVAF-SO corrective infrastructure is realized by merging the AF-QoSM functionality, as described in [12], with the functionality provided by the RVAF-CI/CD and RVAF-RM described in the previous two subsections.

## V. Formalizing the QoS Problem

### A. System model

In its current form, RVAF is mostly suited for soft real-time IoT applications consisting of a set of interconnected services on the same local cloud and where multiple services can be deployed on the same node. Nevertheless, its principles can be easily extended to other scenarios and QoS properties.

As shown in Figure 4, an RVAF model considers the system to be composed of a set of $N$ hosts (nodes) $H_1, ..., H_N$ and a set of $M$ services $S_1, ..., S_M$. Services are interconnected through links $l_{x,y}$, characterising the connection between services $S_x$ and $S_y$. The services $S_x$ and $S_y$ can be a typical Arrowhead SOA consumer/producer services, where each connection between a consumer or provider can have a set of QoS requirements. The QoS requirements are part of the formal specification of the code, and later on saved in the QoS Store.

Each node runs a Reconfiguration module $R_x$, which is mapped to the Jenkins CI tool, (see Figure 4), where $x$ is the index of the node. Each module $R_x$ can be connected to another Reconfiguration module $R$. The initial operation

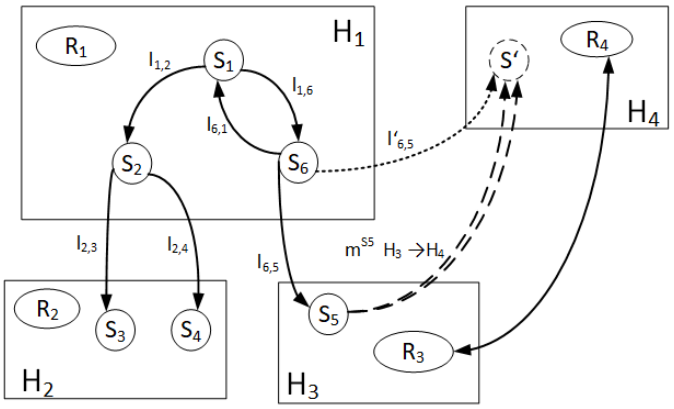


Fig. 4. System Model Example

occurs during service setup, where the Deployment Orchestrator module (in Figure 2) determines in which node the service can be deployed. This initial operation is denoted as $m^{S5}0 \rightarrow H_3$. As depicted in Figure 4, an operation $m^{S5}H_3 \rightarrow H_4$ represents the mobility/reconfiguration of service $S_5$, between nodes $H_3$ and $H_4$, respectively. In this case, $H_3$ represents the source node and $H_4$ represents the the destination node. Link $l'_{6,5}$ represents the connection that has to be established after the mobility operation is completed (rebinding). Consequently, connection $l_{6,5}$ will have to be safely deleted prior to $l'_{6,5}$ becomes operational. By safely, we mean that no messages should be lost or delivered to wrong nodes. This last requirement is ensured by the full implementation of SOA principles, where each transaction is independent from all other transactions.

### B. QoS requirements formal definition

A formal definition of the QoS parameters for a service has to capture functional as well as performance requirements, such as the resources needed by the service to be deployed (e.g., sensors/actuators, CPU power, memory). Therefore, service requirements can be represented by the tuple:

$$SQoS_x = \{U_x, D_x, C_x, Crit_x, M_x\} \quad (1)$$

$U_x$ is the required processor utilization when the service is running. Note that $U_x$ can be a set of values which apply to different hardware platforms (nodes) where the service can be executed. $D_x$ is the service deadline which has a different meaning if the service is a consumer or a producer. For a consumer, this deadline is measured from the moment in time when it is evoked, until it receives a response from the producer service. For a producer service, this deadline is the time that elapses from the moment the service if evoked by the reception of a message from a consumer, until the time it returns a response. $Crit_x$ represents the criticality level of the service, and $M_x$ is the memory required by a service to run. Many other QoS parameters can be defined for each service, like the percentage of lost messages, communications bandwidth, etc.

None of the parameters defined in Eq. 1 are mandatory. The total absence of QoS parameters means that the service is best effort. Also note that the values for these parameters is obtained from the code specification. We also assume that it is possible to know the status of each of the system node, by consulting the QoS Store. The structure of the systems is available by consulting the plant description (which is not depicted in the figures). Each entry on the QoS Store in Figure 2 should contain at least the following data for each host $H_x$:

$$H_x : \{LS_x, LM_x\} \quad (2)$$

$LS_x$ is a list of services with associated QoS requirements ($SQoS_x$). For each of these QoS requirements the QoSM will record the last $R$ measurements on the $LM_x$ list, which enables, as an example, to calculate the average, maximum, and minimum, and determine the tendency of the results. Each entry $i$ of $LM_x$ list is a measurement element $Me_{x^i}$ related with service $S_x$. These values are used in Algorithm 2 to update the QoS requirements of the service on each node.

In this paper we focus only on (performance) QoS parameters like CPU utilization for the service, and memory consumption. We also assume that each service $S_x$ requires a set of specialized resources in order to run, like access to specific sensors. Here we assume that service deployment is done with total independence from other services in a first come first served basis. In other words, if a service cannot be deployed to a node, then it will not be deployed into the system. In order to help the algorithm to decide where to deploy a service, a utility function is defined, which in this case is only based on two parameters, the percentage of CPU utilization ($U_x$) and the memory requirements ($M_x$):

$$U_x^H = f(U_x^h, M_x^h) = K1_x^h U_x^h + K2_x^h M_x^h \quad (3)$$

Eq. 3 uses parameters $K1_x^h$ and $K2_x^h$ to quantify the relative importance of CPU and memory utilization, respectively. The value $U_x^H$ of the utility function is used to optimally assign a service to a host according to its requirements and resources needed. The pseudocode for service deployment is given in Algorithm 1.

**Algorithm 1** RVAF Optimal service deployment

**Input:** List $\mathcal{H}$ of possible hosts $H$ in the SoS, and a service $S_x$ to be deployed

**Output:** The host $H$ with maximum utility $U_x^H$ where $S_x$ should be deployed

```
L = NULL //list of hosts H matching the deployment requirements
for all H ∈ H do
    F = CI("VerifyServiceRequirements",H,S_x) //are the requirements satisfied?
    if F == true then
        Append(L,H) //add host H to the list L of hosts matching S_x requirements
    end if
end for
H = argmax_{H∈L} U_x^H //get matching host with best utility or NULL if L is empty
return H
```

If the service's requirements match the host's properties the service can be deployed to the selected host. Service deployment is controlled by a global container orchestrator or a custom reconfiguration algorithm that triggers a set of tasks in the CI/CD system for the deployment of the service $S_x$ to a host $H_x$. In the example shown in Figure 3 the Service Deployment algorithm instructs the Jenkins CI tool to deploy service $S_x$ to the Host $H_x$, and push the new configuration to the deployment orchestrator.

The container network on the $H_x$ will be updated with the addition of service $S_x$. The capabilities of the AF allow services to be migrated from a host to another while maintaining full functionality. Moreover, local AF clouds can share services through secure connections. The deployment process on an infrastructural scale can be regarded as a technicality. However, service reorchestration at the application level needs to be coordinated with the local AF orchestrator service, and with the AF orchestrator service on the destination host $H_x$. The AF is designed to allow this kind of dynamic service orchestration [13].

### C. Algorithm for run-time adaptation

The problem of deriving the correct QoS parameters, like processor utilization and memory footprint, is very difficult in open IoT systems, where the burden of finding out what is a predictable behaviour for a service, prior to run-time, can have a very high cost to the system developer, or it can even be an impossible task. This problem is mainly due to the following factors:

**Algorithm 2** RVAF Run-time monitoring and adaptation: Migration Action

**Input:** A measurement $M$ of the monitored parameters related to service $S_x$

**Output:** System-redesign alert or a migration $m^{S_x} H_a \rightarrow H_b$ of $S_x$ from $H_a$ to $H_b$

1: $M$ = RM("getMonitoringData",$H_a$,$S_x$) //get monitoring data for $S_x$ from RM
2: $F$ = QoSM("UpdateQoSdata",$H_a$,$M$,$S_x$) //get violation flag from the QoSM
3: **if** $F$ == True **then**
4: RM("QoSViolationAlert",$H_a, S_x$) //publish violation alert to RVAF services
5: $\mathcal{H}$ = CI("getHostList") //get the current list of hosts in the system
6: $H_b$ = Algorithm-1($\mathcal{H}$, $S_x$) //get the optimal host for migration
7: **if** $H_b$ != NULL **then**
8: CD("migrate",$H_a$,$H_b$,$S_x$) //tell CD to migrate $S_x$ from $H_a$ to $H_b$
9: **else**
10: RM("SystemRedesignAlert",$H_a$,$S_x$) //publish redesign alert to service $S_x$
11: **end if**
12: **end if**

i Different programming languages and/or algorithmic constructs can be employed whose behavior in time may be hard to predict;
ii A service can be deployed on different hardware platforms and this can be accomplished with different hardware configurations;
iii No prior knowledge is available on how other application/services interfere with the behaviour of the newly developed service.

The solution in RVAF is to continuously monitor the execution data related to a particular service, in order to derive its appropriate QoS parameters. This is of utmost importance for services deployed on hosts, where no data is available in advance, for the execution of the given service. A simple solution is described in Algorithm 2. The function Update QoS data on $H_x$ can use a very simple algorithm, like calculating the weighted average of the last $N$ values, or it can use more sophisticated algorithms, capable of detecting outliers and filtering out the data. These solutions deserves further investigation.

The results of Algorithm 2 can be used by the RVAF-SO to trigger the migration of service $S_x$ from host $H_a$ to host $H_b$. In case the service deployment fails, when executing Algorithm 1, then a system redesign requirement is signalled to all subscribed hosts. In this extreme case, the application itself fails, and the overall system must return to design phase, as described in Figure 1.

In addition to migration, RVAF is capable of using various other actions, such that the IoT application best adapts to various scenarios. For example: reduction or expansion of available resources, version rollback, and service termination. This requires an extension of the decision process of RVAF-SO, based on the list $\mathcal{A}$ of available actions $A$. For each action $A$ and failure $F$, we assume that SO can compute a decision weight $w_A^F$ according to the current system load:

$$\forall A \in \mathcal{A}, \forall F \in \mathcal{F}, \exists w_A^F \in \mathbb{R} \tag{4}$$

Algorithm 3 shows a simple action-selection algorithm for RVAF-SO. This algorithm is supposed to be called by Algorithm 2, right after the QoS violation is detected. Algorithm 3 will then choose the optimal action for the specific failure $F$ of Service $S_x$, with "migration" as a default option. This SO ability in RVAF can be extended with other self-awareness mechanisms, such as GAMS [14].

**Algorithm 3** RVAF Action evaluation: Picking optimal action

**Input:** QoS violation alert $F$ for measurement $M$ of service $S_x$

**Output:** Optimal action $A$ for failure $F$

1: $\mathcal{A}$ = CI("getActionList",$F$) //get all available actions associated with failure $F$
2: **return** argmax$_{A \in \mathcal{A}}$ SO("getWeight",$A$,$F$) //return action $A$ with best $w_A^F$

## VI. Implementation Results

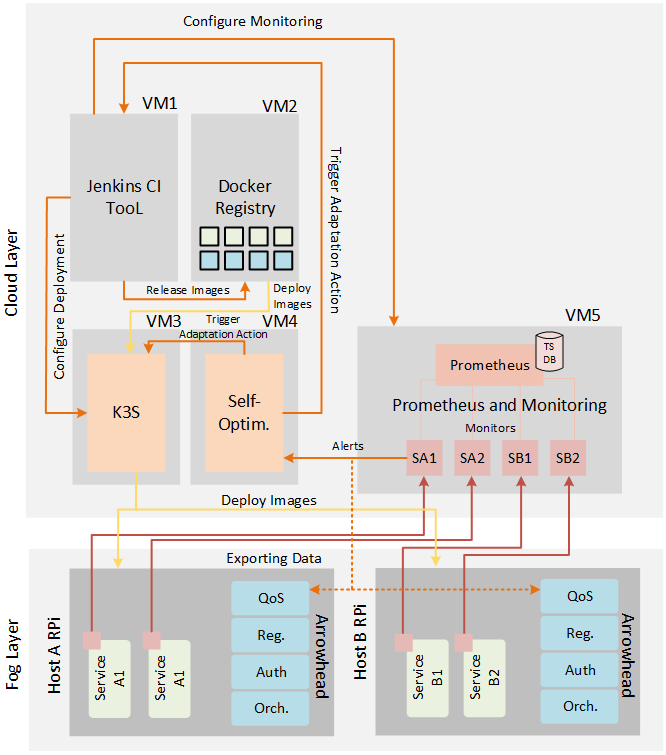


Fig. 5. Component diagram of the implementation

To illustrate the usefulness and versatility of RVAF, we implemented an industrial conveyor-belt application (CBA) with sorting capabilities. The CBA receives information from three induction sensors located on the belt. This information is aggregated in a fog node, in form of a sensor data aggregation service. This can be consumed by a sorting service, located either in the same fog node, in another fog node, or in the cloud. The sensors tell the sorting service, whether an object on the belt is metallic or not. If it is metallic it should be discarded from the belt. This is achieved by the sorting service by using a motor-driven actuator. The sensors and the actuators are interfaced to the CBA services through a soft PLC and an OPCUA server, both located on a fog node.

In Figure 5, we show a possible IoT-based realization of this application, with a redundant fog node, which is capable of replacing the main node, in case of failure, or QoS violation. Each fog node runs a local AF system, and a number of services, including the ones controlling the conveyor belt. The cloud consists of four virtual machines, which are allocated to the RVAF components as follows: CI in VM1, CD in VM1-VM3, RM in VM4, and SO in VM3. The Jenkins tool is used to implement the CI, the Docker Registry is used to store service-container images, the K3S is used as the deployment orchestrator, the SO is our reconfiguration platform given by Algorithms 1-3, and the Prometeus tool is used for data-logging and monitoring purposes. In order to monitor various services concurrently, we use a set of monitors, such as SA1-SA2 and SB1-SB2. The monitoring code is either automatically generated from the associated STL specifications, or written directly in PromQL. For the fog nodes we selected the Raspberry Pi platform.

Services are deployed in a single Docker container per service, and can be individually started, stopped, migrated or rolled back via a deployment orchestrator. Originally we used the Docker Swarm deployment orchestration, however due to the application in IoT, limited resources and development state of the tool we found K3S [15] to be much better suitable for the given task. K3S is a lightweight version of Kubernetes [16] designed for embedded devices and IoT. Each individual service can be observed on the application and container level, on top of the QoS management provided by K3S we add a an application level QoS management in combination with Arrowhead QoS manager.

Figure 6 shows a memory leak trace of a service in the system given in Figure 5, and its migration to the redundant host. It is in the nature of SOA approaches, that the development of services is done by different individuals and organizations. Thus, SOA approaches are error prone. However, the isolation capabilities of container-based deployment, allows us to detect faults and perform corrective actions with least amount of disruption for the entire system. The legend shows several metrics, where the last two (marked in bold) show the memory consumption in two containers. In the considered scenario, the memory leak in one of the services can be disruptive, and deny other services valuable resources, for reaching required reaction time. Mitigation by migration allows us to restart the service on the other fog node, where such behaviour can be better tolerated.

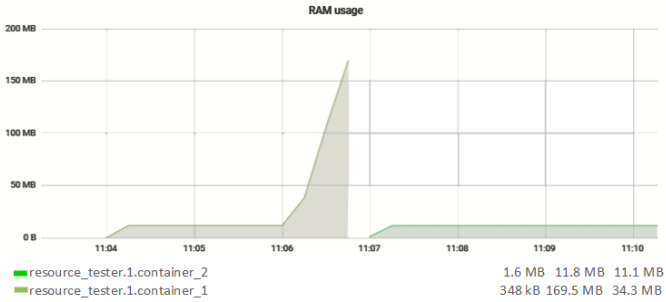


Fig. 6. Service migration results on memory leak

| Stage | Average Time (s) |
|---|---|
| From detecting fault to orchestrator alert | 0.82 |
| Terminating faulty service | 0.25 |
| Terminating container | 1.91 |
| Redeployment of the container | 1.74 |
| Service activation with full JVM stack | 3.49 |

TABLE I
TIME OVERHEAD ON SERVICE REDEPLOYMENT

Table I shows time measurements on different stages of the redeployment process. The response times can be further optimized by reducing overhead from different unnecessary functions in the used tools. As a example service we used a consumer template provided by Arrowhead. It is implemented in Java, hence longer startup time. This can be significantly reduced by using optimized service implementation. In case of migration of services from one node to the reaction times are similar to the ones showed in the I. An additional overhead can be created by using internal K3S scheduling mechanisms for migration. It will try to ”save” the node if possible before resorting to migration, this can create additional overhead on the process. This can be also reduced by configuring K3S for specific time limits during deployment or migration.

## VII. FUTURE WORK

Various runtime monitoring tools have been developed over time, and they use domain-specific languages, to define individual monitors [17], [18]. These languages typically have a formal semantics, such as the signal temporal logic (STL) of [19]. They allow to verify during runtime, if the observed (monitored) behavior of the system, satisfies its given specification, standard, or regulation.

In RVAF we plan to introduce a translation system to formally capture performance properties (e.g., CPU and Memory usage) and functional properties in STL, query the multi-dimensional data acquired from a variety of sources with PromQL, and check their satisfaction. As an example, suppose that the CPU usage of the service $Sx$ in an application $A$ cannot exceed 80%. This can be formally specified in STL, and thereafter be compiled to a runtime monitor containing PromQL language queries, as follows:

STL:

$$A \models G\ (Sx[t] < 80)$$

PromQL:

```
sum(rate(cont_cpu-total_sec="Sx" [1m])) /
count(node_cpu_seconds_totalmode="A")*100<80
```

In future work we also plan to improve AF quality of service manager (QoSM), by fully integrating the the RVAF-CI/CD with the AF-QoSM. We also plan to harmonize the RVAF orchestration-service with the one of AF, within the Arrowhead local cloud or cluster. Finally, we plan to extend the library of actions that are made available for self optimization to the RVAF users.

In particular, AF local clouds have an orchestration service maintaining application level interfaces between services. This ensures the security and stability of the whole ensemble. Maintaining stability during RVAF adaptation, requires the harmonization of RVAF-SO with AF-QoSM and AF-Orchestrator, within the cloud. This can be performed at runtime in the following ways:

i Directly interface AF-QoSM with the RVAF-SO. The QoS violation alerts are sent to both simultaneously, and they agree on a mitigation strategy. This way the AF-QoSM is aware of any requirements violations, and is able to communicate potential changes to the Orchestrator. The Orchestrator updates local or remote cloud connections to the faulty service.

ii Rely on the self-awareness of the local AF cloud and perform mitigation actions for QoS violation, without informing AF-QoSM and AF-Orchestrator. This way, the consumer services are not aware of the changes, until the service is available and reintegrated in the AF system.

iii Rebuild a static configuration of the whole cloud/cluster, and restart the cloud with a new service configuration.

## VIII. Conclusion

IoT applications are generally complex because they are often both distributed and heterogeneous. Moreover, the extension of IoT infrastructures to industrial domains, with high demands for dependability and security, requires the ability to monitoring and ensuring a certain level of quality of service.

This can be achieved by employing modern, continuous integration and continuous deployment tools (CI/CD), which are nowadays an essential part of the development lifecycle of any complex software artifact.

This paper introduced RVAF, an IoT-enabling and service-oriented (SOA) infrastructure, combining the QoS concepts of the Arrowhead Framework (AF), with the CI/CD deployment tools, runtime verification/adaptation, and the computational capabilities of cloud computing. RVAF supports the development of IoT systems that have the ability to: i) Monitor large scale applications in real time, ii) Verify and detect system properties, and iii) Perform self-optimization and service-redeployment during runtime.

## Acknowledgements

This work was partially supported by the EU Horizon-2020 project ADEPTNESS under grant number 871319, the EU ECSEL-JU project Productive4.0 under grant number 737459, the AT BMBWF project CPS-IoT Ecosystem, the PT FCT/MEC project within CISTER Research Unit CEC/04234, and the PT ANI project Portugal-2020.

## References

[1] "Plattform Industrie 4.0." [Online]. Available: https://www.plattform-i40.de/PI40/Navigation/DE/Plattform/Hintergrund/hintergrund.html

[2] P. Varga, F. Blomstedt, L. Ferreira, J. Eliasson, M. Johansson, J. Delsing, and I. Martnez de Soria, "Making System of Systems Interoperable The Core Components of the Arrowhead Framework," *J. Netw. Comput. Appl.*, vol. 81, no. C, pp. 85–95, Mar. 2017. [Online]. Available: https://doi.org/10.1016/j.jnca.2016.08.028

[3] G. Pardo-Castellote, "OMG Data-Distribution Service: architectural overview," in *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.*, May 2003, pp. 200–206, iSSN: null.

[4] Pivotal, "Rabbitmq." [Online]. Available: https://www.rabbitmq.com/

[5] XMPP, "XMPP." [Online]. Available: https://xmpp.org/

[6] L. L. Ferreira, M. Albano, and J. Delsing, "Qos-as-a-service in the local cloud," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2016, pp. 1–8.

[7] H. Isakovic, D. Ratasich, C. Hirsch, M. Platzer, B. Wally, T. Rausch, D. Nickovic, W. Krenn, S. Dustdar, and R. Grosu, "CPS/IoT Ecosystem: A platform for research and education," Oct. 2018, p. 8.

[8] M. T. Delgado, "Eclipse Arrowhead," May 2020. [Online]. Available: https://projects.eclipse.org/projects/iot.arrowhead

[9] Jenkins, "Jenkins." [Online]. Available: https://jenkins.io/index.html

[10] Cloud Native Computing Foundation (CNCF), "Prometheus." [Online]. Available: https://prometheus.io/docs/introduction/overview/

[11] S. K. Chaulya and G. M. Prasad, "Chapter 6 - Formation of Digital Mine Using the Internet of Things," in *Sensing and Monitoring Technologies for Mines and Hazardous Areas*, S. K. Chaulya and G. M. Prasad, Eds. Elsevier, Jan. 2016, pp. 279–350. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780128031940000064

[12] M. Albano, P. Barbosa, J. Silva, R. Duarte, L. Ferreira, and J. Delsing, "Quality of service on the arrowhead framework," in *2017 IEEE 13th International Workshop on Factory Communication Systems (WFCS)*, May 2017, pp. 1–8, iSSN: null.

[13] J. Delsing, *IoT Automation: Arrowhead Framework*. Boca Raton: Taylor & Francis Inc, Feb. 2017.

[14] S. Maksuti, M. Tauber, and J. Delsing, "Generic Autonomic Management as a Service in a SOA-based Framework for Industry 4.0," in *IECON 2019 - 45th Annual Conference of the IEEE Industrial Electronics Society*, vol. 1, Oct. 2019, pp. 5480–5485, iSSN: 1553-572X.

[15] "K3s: Lightweight Kubernetes." [Online]. Available: https://k3s.io/

[16] "Kubernetes." [Online]. Available: https://kubernetes.io/

[17] S. Jaksic, E. Bartocci, R. Grosu, and D. Nickovic, "An algebraic framework for runtime verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2233–2243, November 2018.

[18] E. Bartocci, J. Deshmukh, A. Donzé, G. Fainekos, O. Maler, D. Ničković, and S. Sankaranarayanan, *Specification-Based Monitoring of Cyber-Physical Systems: A Survey on Theory, Tools and Applications*. Cham: Springer International Publishing, 2018, pp. 135–175. [Online]. Available: https://doi.org/10.1007/978-3-319-75632-5_5

[19] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, Y. Lakhnech and S. Yovine, Eds., vol. 3253. Lecture Notes in Computer Science, 2004, pp. 152–166.