Industry Paper: Managing Geo-Distributed Stream Processing Pipelines for the IIoT with StreamPipes Edge Extensions

Patrick Wiener
wiener@fzi.de
FZI Research Center for Information Technology
Karlsruhe, Germany

Philipp Zehnder
zehnder@fzi.de
FZI Research Center for Information Technology
Karlsruhe, Germany

Dominik Riemer
riemer@fzi.de
FZI Research Center for Information Technology
Karlsruhe, Germany

ABSTRACT

The industrial IoT and its promise to realize data-driven decision-making by analyzing industrial event streams is an important innovation driver in the industrial sector. Due to an enormous increase of generated data and the development of specialized hardware, new decentralized paradigms such as fog computing arised to overcome shortcomings of centralized cloud-only approaches. However, current undertakings are focused on static deployments of standalone services, which is insufficient for geo-distributed applications that are composed of multiple event-driven functions. In this paper, we present StreamPipes Edge Extensions (SEE), a novel contribution to the open source IIoT toolbox Apache StreamPipes. With SEE, domain experts are able to create stream processing pipelines in a graphical editor and to assign individual pipeline elements to available edge nodes, while underlying provisioning and deployment details are abstracted by the framework. The main contributions are (i) a fog cluster management model to represent computing node characteristics, (ii) a node controller for pipeline element life cycle management and (iii) a management framework to deploy event-driven functions to registered nodes. Our approach was validated in a real industrial setup showing low overall overhead of SEE as part of a robot-assisted product quality inspection use case.

CCS CONCEPTS

• **Information systems → Computing platforms**; • **Computer systems organization → n-tier architectures**; **Data flow architectures**.

KEYWORDS

stream processing, fog computing, industrial internet of things

ACM Reference Format:
Patrick Wiener, Philipp Zehnder, and Dominik Riemer. 2020. Industry Paper: Managing Geo-Distributed Stream Processing Pipelines for the IIoT with StreamPipes Edge Extensions. In *The 14th ACM International Conference on Distributed and Event-based Systems (DEBS '20), July 13–17, 2020, Virtual Event, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3401025.3401764

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DEBS '20, July 13–17, 2020, Virtual Event, QC, Canada
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8028-7/20/07…$15.00
https://doi.org/10.1145/3401025.3401764

1 INTRODUCTION

The steady increase in digitalization in industrial domains such as manufacturing, energy, or logistics has lead to a deluge of generated data with the industrial internet of things (IIoT) as a key enabler bridging the physical and virtual worlds. This offers great opportunities for companies to harvest new data sources and deduce meaningful data-driven insights. Typical use cases include improvements in product and process quality, as well as collaborative human-machine scenarios that enable companies to generate competitive cost advantages. For instance, in visual quality inspection, readings from a camera sensor are used for quality predictions by leveraging edge-based machine learning models to classify product anomalies such as deviations in terms of size, shape or component skew, consequently triggering a robot to sort out that product for rework or scrap as depicted in Figure 1. In recent years, in order to satisfy this insatiable need for computing capability, while still being able to face all the challenges associated, specialization in hardware (heterogeneity) has been massively adopted [26]. Together with the decrease in hardware size and costs in conjunction with the development of new decentralized computing paradigms such as fog and edge computing attracts businesses to invest thereby overcoming

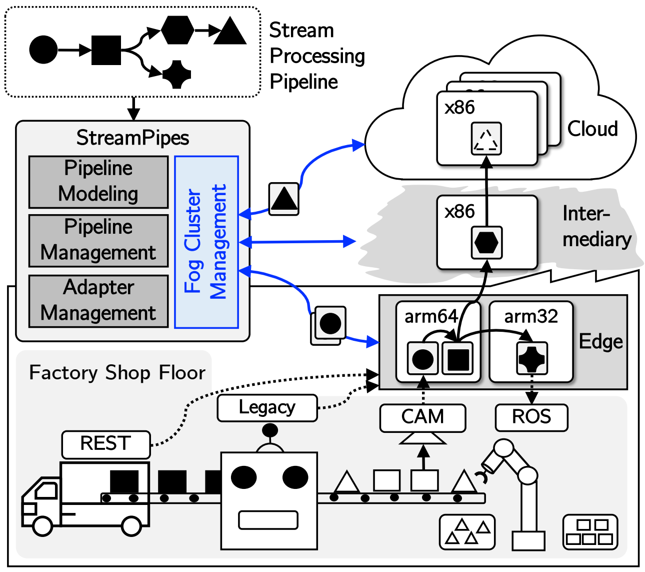


Figure 1: A stream processing pipeline deployed over heterogeneous resources from edge to cloud to analyze factory shop floor data to derive time-sensitive insights.

the shortcomings of previous cloud-only approaches in terms of latency, privacy or available bandwidth. Thus, this offers new possibilities for a new generation of stream processing applications that are centrally modeled, and deployed over a pool of geographically distributed compute resources.

However, this induces a variety of challenges: (i) unlike cloud resources, intermediary fog and edge compute nodes are highly heterogeneous with regard to their hardware, such as differences in CPU architectures (e.g., x86, arm32, arm64, etc.), (ii) with the large deployment of compute nodes, the managing middleware becomes more responsible for exposing the right level of abstraction to the upper level stacks, including pipeline orchestration and management, as well as algorithm selection (e.g., edge optimized machine learning model), (iii) due to fog/edge nodes being exposed to the physical world, we cannot always assume a stable connection between or within the compute layers that affects messaging between geo-graphically deployed pipeline elements, (iv) the technical complexity concerning the deployment and provisioning of pipeline elements needs to be hidden from (non-technical) domain experts who are defining the application logic of pipelines.

This paper presents a conceptual model and implementation supporting the management of geo-distributed stream processing pipelines in fog and edge environments. Our approach is based on an extension to the Industrial IIoT toolbox Apache StreamPipes (incubating)[1], which was originally created by the authors of this paper and is now an incubating project of the Apache Software Foundation. StreamPipes includes an adapter library to connect industrial data sources and a graphical editor to allow non-technical users to create stream processing pipelines based on a repository of reusable stateless or stateful functions. This paper highlights a major contribution to StreamPipes, namely *StreamPipes Edge Extensions*, which add the capability to register edge computing nodes in StreamPipes and allows users to automatically deploy pipeline elements to registered edge nodes. Our implementation is completely open source[2] and has been validated in real industrial settings. The first core innovation presented in this paper is concerned with a fog cluster management model that aims at representing capabilities of nodes at data-, software- and hardware-level. Second, we introduce a node controller architecture for pipeline element life cycle management. Third, we outline concept and implementation of a management framework that allows users to selectively deploy pipeline elements to registered edge nodes.

The remainder of this paper is structured as follows: In Section 2, we summarize the two foundational building blocks of this paper *flow-based frameworks for IIoT analytics* and *decentralized computing paradigms*. Section 3 presents three real-world motivating scenarios to highlight the need for geo-distributed pipeline management. Afterwards, the fog cluster resource model is introduced in Section 4, followed by the description of the node management framework in Section 5. Finally, the validation and an overview of related work are presented in Sections 6 and 7.

[1]https://streampipes.apache.org/
[2]https://github.com/apache/incubator-streampipes/tree/edge-extensions

2 BACKGROUND

In this section, we introduce flow-based approaches for stream processing as well as fog computing, a decentralized computing paradigm that we built on in this work.

Flow-based Frameworks for IIoT Analytics. Apache StreamPipes is an incubator project of the Apache Software Foundation, that provides a reuseable toolbox to easily connect, analyze and exploit a variety of IIoT-related data streams without any programming skills. Therefore, it leverages different technologies especially from the fields of stream processing, distributed computing and semantic web. Based on the dataflow programming paradigm [21], StreamPipes allows to model new stream processing pipelines as a sequence of pipeline elements from an extensible toolbox and execute them in a distributed environment consisting of multiple, potentially heterogeneous runtime implementations. The decomposition of complex analytical problems into smaller function blocks allows StreamPipes to mitigate the problem of committing to a single stream processing technology. On top, it uses semantics to provide guidance to non-technical domain experts throughout the pipeline creation process. Besides StreamPipes, there exists other solutions for low-code dataflow programming, namely Apache Nifi, or Node-RED.

Decentralized Computing Paradigms. While cloud computing has acted as the de-facto standard computing paradigm over the past decade, the massive increase in the access and deployment of sensors (e.g., machines with MQTT, OPC-UA interfaces), and spezialized hardware (e.g., NVIDIA Jetson) paved the way for new concepts to manage the data growth, especially in the IIoT. This lead to the acute need to move a substantial amount of processing closer to the source thereby complementing the cloud with a decentralized computing paradigm that is often referred to as fog computing [3]. Since there exists no clear consensus on the term itself, neither by researchers nor practitioners, we refer to [9, 32] for a detailed comparison of fog computing and its related computing paradigms such as edge computing, mist computing or cloudlet computing. In our previous work [30], we defined fog computing as a multi-layered, hierarchical and heterogeneous model for enabling access to a shared continuum of compute resources including edge, intermediary fog and cloud nodes and allowing for dynamic reconfiguration.

3 MOTIVATING SCENARIOS

In this section, we present three use cases for the IIoT to illustrate the manifold of different characteristics shaping the application space to highlight the need for geo-distributed pipeline management as exemplified in a typical smart factory (see Figure 1).

AI-assisted Inbound Logistics (**UC1**). Automated shipment recognition and recording are current topics in production and logistics both in industry as well as academia. In the case of inbound logistics, the automated visual recognition of the packing structure, i.e. number/type of load carriers, bears great potential. Thereby, when a new shipment arrives, a camera sensor streams recordings to a machine learning classifier, that detects and identifies the dedicated packing units that are enriched with additional order information and send to the corresponding inventory management system.

Table 1: IIoT characteristics of the presented use cases

N°	Rate	Privacy	Latency	Targeted Receiver
UC1	low	high[1]	moderate	receiving, enterprise level
UC2	high	low	low	robot, factory level
UC3	high	low	low	shop floor, factory level

[1] to comply with the General Data Protection Regulation (GDPR).

Product Quality Inspection (**UC2**). Over the last decade, automation has gained immense popularity especially in the industrial domain. In this regard, collaborative robots are one of the major automation types supporting companies in various applications, e.g., picking/packing, handling materials or for product quality inspection. Especially in the latter case, this is key to ensure high product quality and thus high customer satisfaction. Rather than manually performing quality inspections, collaborative robots can be leveraged to execute this dull task at high accuracy without exhaustion. Here, data from equipped sensors can be used to analyze the event stream for deviations from defined targets by the domain expert. As a result, failed units are automatically sorted out for rework or scrap and process data is stored for traceability or offline training.

Asset Condition Monitoring (**UC3**). In industrial operations, asset condition monitoring captures the state of machines and equipment of manufacturing companies while running. Here, machine health state can be continuously assessed by measurements from various sensors, e.g., acceleration, pressure, or vibration, that are analyzed on the fly to detect abnormal behavior in order to either react in a timely manner or predict the remaining useful life for this asset. This lets manufacturers identify and fix causes for costly unplanned downtime to increase machine utilization and availability throughout the factory shop floor. However, assessing machine health is difficult given the average lifespan of industrial machines, brownfield IIoT deployments are common [2]. On the other hand, sensor data comes in high velocity and volume where hidden problems and guesswork of domain experts can incur extra expenses.

IIoT Characteristics. What the presented use cases have in common is that they all decompose complex problems into smaller analytical subtasks that target various places in the hierarchical infrastructure ranging from edge deployments with fast response times to cloud deployments for overall monitoring and storing of results. We summarize the use cases in Table 1 by putting them into three categories with respect to (i) the used data, (ii) their requirements towards data protection and latency, as well as (iii) their analytics target destination. Given this information, we can identify the necessity for local event processing on the edge layer, either induced by the high frequency at which data is collected (**UC2**, **UC3**), the regulation to comply with GDPR due to possibly collecting sensitive personal data (**UC1**), or the criticality of latency (**UC2**, **UC3**). In addition, computation results of stream processing pipelines are leveraged to update centrally deployed cloud-based systems on enterprise or factory level including business relevant enterprise resource planning as well as quality management systems.

4 FORMAL MODEL DESCRIPTION

This section presents our application and resource model for the IIoT and serves as a foundation for our approach to managing stream processing pipelines over the layered resource pool from edge to cloud (cf. Section 5).

4.1 Stream Processing Application Model

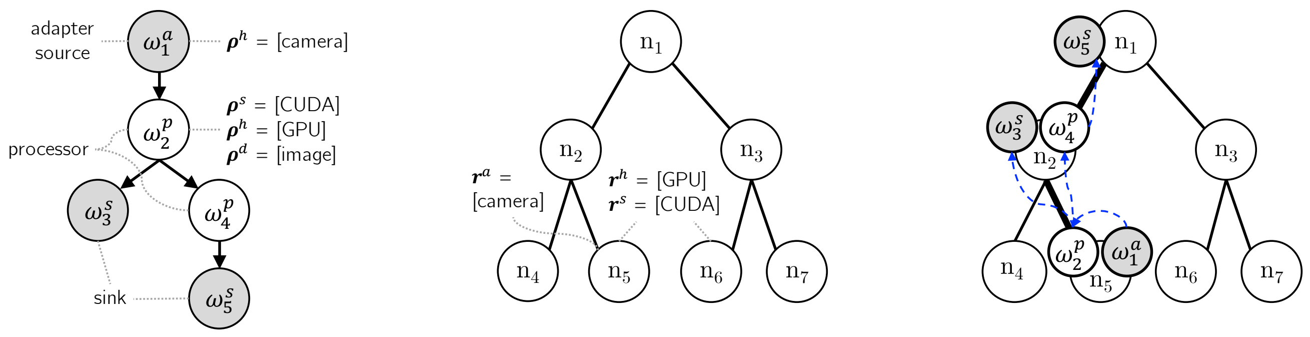
Our stream processing application model is based on the dataflow programming pattern [12, 21] that encapsulates functions in discrete, reusable computational blocks called pipeline elements each of which follow the single responsibility principle and only focus on a specific task. Multiple consecutive pipeline elements are interconnected based on the well-known publish-subscribe pattern [11]. Generally, a pipeline element ω can fall into one of the following three categories:

- **adapter sources** ω^a are runtime instantiations from an extensible pool of adapter templates and expose a data stream consisting of a *semantic description* with information on the schema (event properties), the grounding (transport format such as JSON, transport protocol such as MQTT, or Kafka) and a *runtime implementation* operating on event level [33],
- **processors** ω^p can be stateless/stateful functions applied to a data stream. Similarly to adapter sources, processors also consist of a *semantic description* used to determine compatibility to an input event stream, user input and the definition of the output event stream as well as a *runtime implementation* where the actual logic is implemented.
- **sinks** ω^s are very similar to data processors with the exception that sinks do not produce any output streams and thus mark the end of a pipeline.

We consider the following requirements for pipeline elements that are on (i) *data-level* $\boldsymbol{\rho}^d$, e.g., quality, frequency, or specific domain property types of the incoming data stream, (ii) *software-level* $\boldsymbol{\rho}^s$, e.g., operating system, library or kernel version, as well as (iii) *hardware-level* $\boldsymbol{\rho}^h$, e.g., presence and type of GPU, CPU, memory, or possibility to access dedicated sensor/actuator resources such as camera modules (wired or via IP). For instance, let us consider an image analytics pipeline as exemplified in Figure 2a. Here, the adapter source ω_1^a has a hardware requirement $\boldsymbol{\rho}^h = [camera]$ to be connected to a camera and produces an image stream. The subsequent processor ω_2^p declares a data stream requirement expecting it to contain an image event $\boldsymbol{\rho}^d = [image]$ besides additional CUDA software requirements $\boldsymbol{\rho}^s = [CUDA]$ and GPU hardware requirements $\boldsymbol{\rho}^h = [GPU]$. Lastly, the remaining pipeline elements do not address any special requirements.

This stream processing application model of interconnected pipeline elements are ideal to be easily packaged and deployed as pipeline element container including their application logic, runtime environment as well as configuration on the underlying infrastructure [30].

More formally, let us consider a stream processing pipeline SPP, its dedicated set of pipeline elements Ω as well as their interconnections Ψ. Thus, a pipeline element ω of category adapter source ω^a, processor ω^p or sink ω^s is defined as follows.


adapter source
ω_1^a
$\boldsymbol{\rho}^h$ = [camera]
ω_2^p
$\boldsymbol{\rho}^s$ = [CUDA]
$\boldsymbol{\rho}^h$ = [GPU]
$\boldsymbol{\rho}^d$ = [image]
processor
ω_3^s
ω_4^p
sink
ω_5^s
n_1
n_2
n_3
$\boldsymbol{r}^a$ = [camera]
$\boldsymbol{r}^h$ = [GPU]
$\boldsymbol{r}^s$ = [CUDA]
n_4
n_5
n_6
n_7

(a) stream processing pipeline SPP **(b) fog computing infrastructure** FCI **(c) mapping result** $SPP \mapsto FCI$

Figure 2: Stream processing application model (2a), fog cluster resource model (2b) and best fit mapping result (2c).

Definition 4.1 (Pipeline Element). A pipeline element ω is formally described as a tuple $\omega = (\mathcal{F}, s_{in}, s_{out}, \boldsymbol{\rho}^d, \boldsymbol{\rho}^s, \boldsymbol{\rho}^h, m, \delta)$, where:

- $\mathcal{F}$ is the encapsulated function
- s_{in} is the input data stream
- s_{out} is the output data stream
- $\boldsymbol{\rho}^d$ is a data stream requirement vector
- $\boldsymbol{\rho}^s$ is a software requirement vector
- $\boldsymbol{\rho}^h$ is a hardware requirement vector
- m are semantic metadata information
- δ contains configuration parameters

As a result, we model a stream processing pipeline SPP as a directed acyclic graph $SPP = (\Omega, \Psi)$ where $\Omega = \{\omega_1, \omega_2, \ldots, \omega_n\}$ is the finite set of n pipeline elements deployed along the cloud-edge continuum, Ψ is the set of edges connecting two consecutive pipeline elements of that graph. In addition, a SPP is further characterized by $\boldsymbol{\rho}^u$ that denotes a vector of additional *user*-level requirements w.r.t. pipeline prioritization (low, medium, high), or deployment strategies, e.g., network-oriented (bandwidth, latency), task-oriented (deadline driven), location-oriented (bound to a specific geographical location) [22]. While the before mentioned requirements are generally known and specified by the corresponding developer during the development phase, this requirement category is specific to the designated end user, i.e domain expert, and thus only known at pipeline modeling time. Furthermore, we consider two individual pipeline elements ω_i and ω_j adjacent, if there exists an edge $\psi_{i,j}$ between (ω_i, ω_j) s.t. ω_i is the parent node and ω_j the child node. Let us define a stream processing pipeline as follows.

Definition 4.2 (Stream Processing Pipeline). A stream processing pipeline SPP is formally described as a directed acyclic graph (Ω, Ψ), with additional requirements $\boldsymbol{\rho}^u$ where:

- Ω is a set of pipeline elements
- Ψ is a set of interconnections
- $\boldsymbol{\rho}^u$ is a user-defined requirement vector

4.2 Fog Cluster Resource Model

The physical fog infrastructure consists of set of nodes possessing computational power and/or storage capacity [22]. We consider the fog cluster resource model to be based on a multi-layered, hierarchical heterogeneous pool of shared resources that vary in terms of their (i) geographical distribution and location (e.g., edge, fog, cloud), (ii) proximity to the data sources with regards to latency, (iii) node resources, i.e. hardware and software (e.g., CPU architectures such as arm32, arm64, x86).

Generally, the infrastructure topology of a fog cluster consists of several interconnected compute nodes. Thus, we can model it as follows. Let us consider a heterogeneous fog cluster infrastructure graph $FCI = (N, C)$ where $N = \{n_1, n_2, \ldots, n_m\}$ describes the set of nodes, each of which is a dedicated fog cluster resource along the edge-cloud continuum, and a set of network connections C linking two nodes in the cluster, where an individual connection $c_{i,j} \in C$ connects the node pair (n_i, n_j). As shown in Figure 2b, nodes provide node resources and may also expose specific capabilities that are (i) *hardware resources* $\boldsymbol{r}^h$, e.g., specialized hardware such as GPU, (ii) *software resources* $\boldsymbol{r}^s$, e.g., libraries such as CUDA, as well as (iii) *accessible sensor/actuator resources* $\boldsymbol{r}^a$, e.g., connected industry camera.

Definition 4.3 (Fog Cluster Infrastructure). A fog cluster infrastructure FCI is formally described as an undirected, heterogeneous graph (N, C) where:

- N is a set of nodes from edge, fog, or cloud layer
- C is a set of network connections

The general research area of application placement, often referred to as fog application assignment problem, is broadly studied in literature [4, 15, 22] also in terms of multi-component application placement [1, 28]. The placement problem defines a mapping pattern by which application components and links are mapped onto an infrastructure graph. Figure 2c shows a mapping result of the exemplified SPP graph (Figure 2a) on a given FCI graph (Figure 2b). Typically, this involves finding available node resources that (i) satisfy the pipeline element and user requirements, (ii) satisfy the given constraints (e.g., locality constraints), as well as (iii) optimize an objective function (if any).

For the sake of this paper, we focus on the actual pipeline element management on geo-distributed nodes where we assume an a priori placement as a result of a manual user selection.

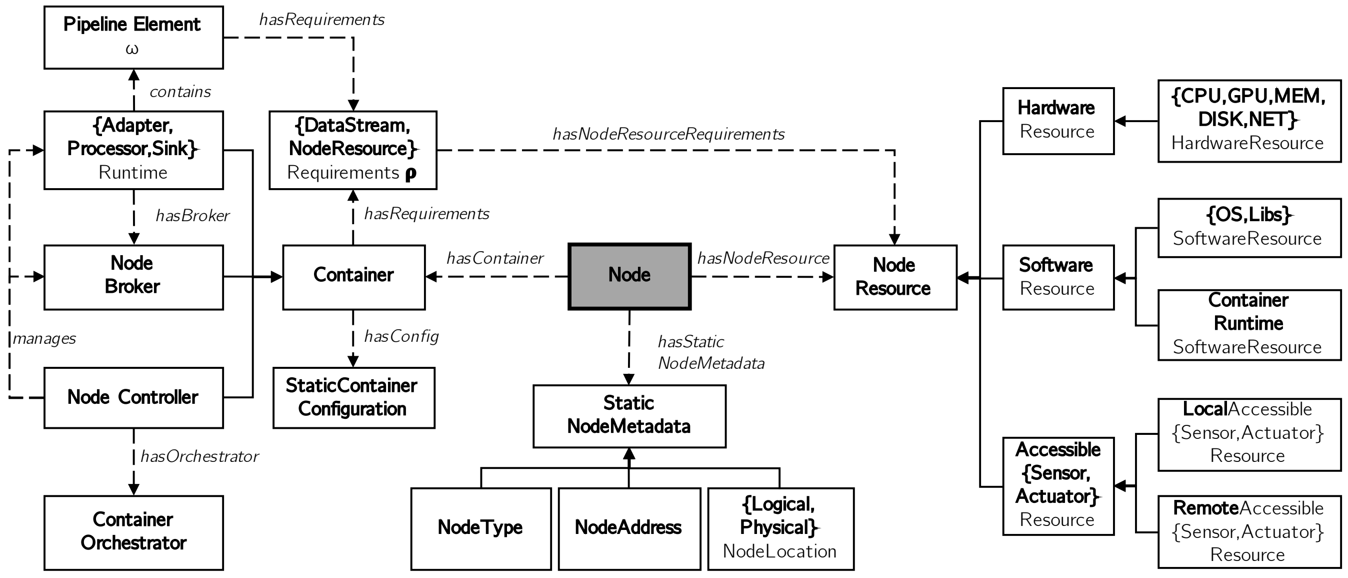

Pipeline Element ω
hasRequirements
contains
{Adapter, Processor,Sink} Runtime
{DataStream, NodeResource} Requirements ρ
hasNodeResourceRequirements
hasBroker
hasRequirements
Node Broker
Container
hasContainer
Node
hasNodeResource
Node Resource
manages
hasConfig
hasStatic NodeMetadata
Node Controller
StaticContainer Configuration
Static NodeMetadata
hasOrchestrator
Container Orchestrator
NodeType
NodeAddress
{Logical, Physical} NodeLocation
Hardware Resource
{CPU,GPU,MEM, DISK,NET} HardwareResource
{OS,Libs} SoftwareResource
Software Resource
Container Runtime SoftwareResource
Accessible {Sensor, Actuator} Resource
LocalAccessible {Sensor,Actuator} Resource
RemoteAccessible {Sensor,Actuator} Resource

Figure 3: Conceptual node model in a fog cluster infrastructure showing inheritance (⟶) and relationships (- ->).

5 NODE MANAGEMENT

In this section, we give a detailed descriptions of our node management approach. For the sake of simplicity, our node model was explicitly kept at a high granularity, but still providing a good level of abstraction w.r.t. describing the relevant components.

5.1 Node Model

The central resource of a fog cluster infrastructure for hosting pipeline elements of stream processing pipelines is a node. Thus, we give a detailed description of our node model as illustrated in Figure 3.

Static Node Metadata. Each node in the cluster provides static node metadata about itself. Thereby, a node can either be of type virtual node, for instance a virtual machine in the cloud, or of type bare-metal that is more typical for edge/fog nodes such as a Raspberry Pi, Intel NUC, or NVIDIA Jetson. Besides, nodes can be reached by their node address, i.e. via IP or DNS. Concerning the node location, we distinguish between logical and physical node location. Here, the logical node location is a simple tag attached to node related to the hierarchical layers of the fog infrastructure (e.g., cloud, fog, edge), whereas the physical node location[3] expresses the actual position on a more fine-grained level. While this location type could also be modeled as absolute geographical coordinates in terms of latitude and longitude pairs, this is not practicable in the context of IIoT, as nodes are deployed either directly on the factory shop floor level itself or on companies premises with a reference to machines, assembly lines, buildings or factory sites. Thus, we represent the physical node location as set of *semantic tags*, e.g., `factory=<A>, building=<B>, machine=<C>`.

[3]Only relevant for fog/edge nodes

Node Resources. A node (of any type), provides dedicated resources to be used by stream processing applications. For our purpose, we consider the following resource types (i) hardware resources, (ii) software resources, and (iii) accessible sensor/actuator resources. Since we operate in a geo-distributed environment along the cloud-edge continuum, the resources are to be considered highly heterogeneous. Hardware resources subsume any type of resources that are necessary for processing (compute), transferring (network) and storing data (storage). This includes typical resources such as CPU, memory, disk and network, but also newly prominent ones such as GPU. Especially the latter is beneficial in regards of edge AI deployments, where a specific pipeline element contains a pre-trained AI model leveraging the hardware acceleration of the onboard GPU. Besides, a node has software resources that fall into the category of operating system related with regards to the type, e.g., Debian, specific libraries, e.g., NVIDIA CUDA drivers[4], but also the installed container runtime. Even though we consider the container runtime to be based on Docker for the sake of convenience, our model is not exclusively restricted on it. Lastly, a node can also have access to sensors/actuators, either locally, e.g., a camera connected to a USB port, or remotely, e.g., via resolvable IP address of a dedicated endpoint (OPC-UA server, or programmable logical controller).

Container. Leveraging the container technology has many benefits and well-addresses the demands in resource-limited and heterogenous fog computing infrastructures. Especially, when it comes to managing stream processing pipelines in this geo-distributed setup, this is where container technology alleviates the deployment of individual pipeline elements. As the runtime instantiation of its corresponding container image, a container inherits certain properties that allow it to run on heterogeneous node architectures (arm32,

[4]Essential to run GPU accelerated pipeline elements in containers.

arm64, amd64, etc.). A container per se has certain static configurations that enables it to be parametrized when instantiated, and thus adapted for the given use case. Containers can address node resource requirements in order to satisfy the application running inside.

The **node controller** container is part of a overall fog cluster management concept and is a locally deployed instance responsible to manage pipeline elements throughout their application life cycle and hence triggers pipeline element runtimes to start/stop dedicated adapters, processors or sinks according to the invocation request of the pipeline management backend. Besides, the node controller manages a node broker container instance responsible to enable messaging between locally deployed pipeline elements. Furthermore, the node controller implements interfaces to a container orchestrator, e.g., Docker, or Kubernetes, to do the actual container deployment and supervision. We provide a more detailed description of the node controller architecture in the following (cf. Section 5.2).

The **adapter, processor, sink runtime** containers provide lightweight standalone runtime wrappers for hosting individual pipeline elements such as adapters (e.g., OPC-UA), processors (e.g., numerical filter, AI models) and sinks (e.g., databases, interfaces to Robot Operating System). Once deployed by the node controller, they register themselves at it while the node controller further propagates their availability to the central node management. As described in Section 4.1, pipeline elements formulate certain requirements on the data stream as well as the node resources that are considered by the placement algorithm.

As mentioned, the **node broker** container enables communication between individual pipeline elements running on the same node thereby leveraging a publish-subscribe pattern. We refer to this pattern as intra-node data flow. In the case of network outages the node broker acts as an event buffer and allows for disconnections of nodes by temporarily persisting the events. On the other hand, inter-node data flow can be realized by relaying local node broker events to the node broker instance hosting the adjacent (child) pipeline elements of a given stream processing pipeline.

5.2 Node Controller Architecture

Overall, the fog cluster management is following the master-slave pattern [11], thus once started on a cluster node, the node controller (slave) registers itself as a new available node in the backend (master) as shown in Figure 4. For this purpose, we use a high available key-value store for node metadata storage and discovery of active nodes. Now, when a new stream processing pipeline is modeled and deployed by the domain-expert, the pipeline management asks the fog cluster management for available nodes and provides the specified requirement vectors (see Section 4.1). Next, individual pipeline elements are assigned to dedicated nodes while satisfying the overall requirements and constraints. On the node itself, it is a crucial task to manage pipeline elements throughout their life cycle, from the actual deployment step and resource monitoring for changes to removing and cleaning up. Hence, we provide a detailed description of the underlying node controller architecture and its interplay with the corresponding pipeline element runtimes and the container orchestrator.

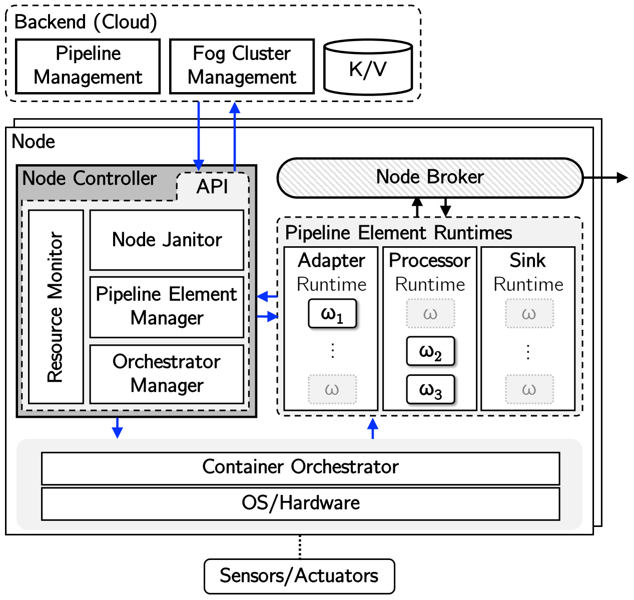


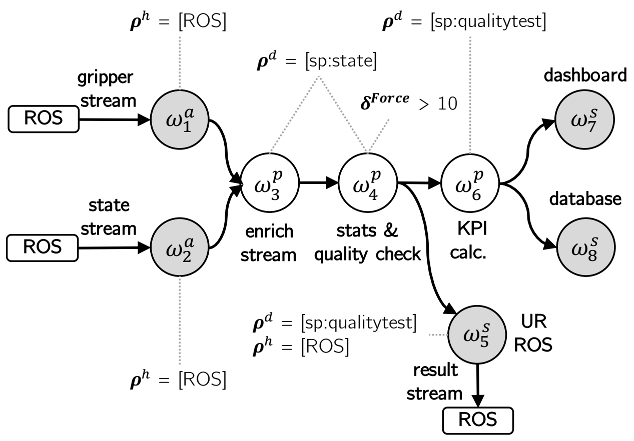
Figure 4: Node architecture with node controller, pipeline element runtimes (adapter, processor, sink) and local node broker instance showing internal and external control flow (→) to the cloud-based backend, as well as data flow (→) of running pipeline elements ω_1, ω_2 and ω_3 via node broker including optional access to sensors/actuators.

Node Controller API. Overall, we distinguish between data flow and control flow. While the first denotes to the actual flow of events from event sources, over processors to sinks, the latter refers to all system-side messages that are essential to managing stream processing pipelines in a geo-distributed deployment. Similarly to the data flow, we differentiate between intra-node control flow, meaning all local communication, e.g., to the pipeline element/adapter runtime or container orchestrator, and inter-node control flow to the backend. Thus, the node controller exposes a set of API endpoints based on the REST protocol in order to enable external and internal communication as summarized in Table 2.

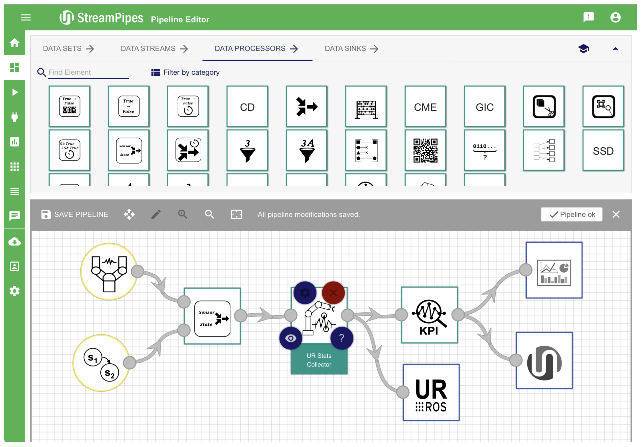
Table 2: Node controller API endpoints.

Endpoint	Description
`/info`	Provides static node metadata^{1,3,G}
`/status`	Provides the current resource status^{1,3,G}
`/deploy`	Deploys PE^{4} runtime container^{2,3,P}
`/invoke`	Starts dedicated PE in PE runtime container^{2,3,P}
`/register`	Registers PE's in PE runtime container^{2,3,P}
`/detach`	Stops dedicated PE in PE runtime container^{2,3,P}
`/remove`	Removes PE runtime container^{2,3,D}

[1] Full endpoint: `/node/<endpoint>`
[2] Full endpoint: `/node/container/<endpoint>`
[3] HTTP GET (G), POST (P), DELETE (D) method
[4] Pipeline element



(a) Quality inspection and KPI analytics pipeline $SPP_{Q,KPI}$



(b) User-modeled $SPP_{Q,KPI}$ in StreamPipes

Figure 5: Product quality inspection and KPI analytics pipeline $SPP_{Q,KPI}$ as abstract definition (5a) as well as the implementation within StreamPipes (5b). $SPP_{Q,KPI}$ consists of eight pipeline elements: instantiated ROS adapter sources (ω_1^a, ω_2^a), processors (ω_3^p, ω_4^p, ω_6^p) as well as sinks (ω_5^s, ω_7^s, ω_8^s) including the individual requirements ρ and user-defined configurations δ.

Orchestrator Manager. Since we leverage container technology for the ease of orchestration and deployment, we rely on a dedicated container orchestrator to handle the heavy lifting of actually starting/stopping the runtime container instances containing the dedicated pipeline elements. Thereby, the orchestrator manager implements a generic container orchestrator interface with dedicated methods to handle incoming HTTP requests on the `/deploy` and `/remove` endpoints from the backend. Additionally, the interface can easily be extended to address additional functionality. This allows the orchestrator manager to be implemented for any state-of-the-art container orchestration technology.

Pipeline Element Manager. The pipeline element manager is responsible for pipeline element life cycle management. Once the pipeline element runtime containers are deployed, they register their availability including semantic descriptions about contained pipeline elements at the pipeline element manager (`/register`), that further propagates it to the central backend. Now, when a stream processing pipeline is deployed, the node controller API gets an invocation request (`/invoke`) for a specific pipeline element to be started that the pipeline element manager forwards alongside the invocation graph containing information about the corresponding stream processing pipeline, the event schema and event grounding (transport format, protocol, topic to subscribe/publish) to the dedicated runtime container. In Figure 4, we see pipeline elements ω_1, ω_2 and ω_3 in running stage, while others were not yet invoked. Only when the user stops the corresponding stream processing pipeline, the respective pipeline elements are stop their job. Similarly as before, the pipeline management sends a detach request (`/detach`) containing the running element id that is further distributed to dedicated runtime containers to stop the pipeline element. Lastly, when a pipeline element runtime container is removed from a node, the pipeline element manager deregisters them from the backend such that they are not available anymore (`/remove`).

Resource Monitor and Node Janitor. The node controller also contains a resource monitor to observe the current node resource consumption, that can be assessed by calling the `/status` endpoint. Besides, the ease of deployment that the container technology brings, it also introduces the challenge of not littering the host with partly downloaded (dangling) images and volumes. In this respect, this can easily lead to large amounts of disk storage to be used for local image cache or old host volumes of previously mapped containers. Especially on resource-constrained industrial edge nodes that are oftentimes specialized hardware of Raspberry-Pi-like devices with storage in the low gigabyte range, this can be a serious issue. Hence, the node janitor is a scheduled service that gets triggered at predefined intervals and cleans the space on the node.

6 EVALUATION

To demonstrate how to manage pipeline elements of decomposed stream processing pipelines over a heterogenous, geo-distributed infrastructure, we implemented the product quality inspection use case (see **UC2** Section 3) in combination with a continuous calculation of specific key performance indicators (KPI) for an overall process and quality assessment.

6.1 Collaborative Robot based Product Quality Inspection and KPI Analytics Pipeline

In our validation setup, a Universal Robots UR5e [27] (flexible collaborative robot arm) is used to autonomously assemble an industrial product consisting of various parts including a mainboard, a housing and a housing cover. In the last step of the product assembly, the UR5e picks a housing cover out of a load carrier box to finalize the assembly process thereby putting it onto the housing itself where the mainboard has already been inserted. In order for the housing cover to stay in place it is equipped with a spring whose actual force is of importance for the cohesion of the end product.

Fortunately, testing the spring force can be realized by leveraging a specific gripper module for the UR5e where the effective gripping force (while picking the housing cover) is continuously measured via a force sensor deployed in the gripper's fingers. Thereby, the sensor measurements of the gripper module (**gripper stream**) as well as the robot's internal state machine (**state stream**) are accessed via the Robot Operating System (ROS) to be analyzed as depicted in Figure 5a. While the first is a continuous stream of gripper events the latter represent discrete state change events to retrieve knowledge about the current process step. Thus, the overall goals are as follows.

1. **quality assurance**: assess wether a housing cover is ok/not ok based on the aggregated force measurement when performing the quality check and inform the robot about the test result.
2. **process assessment**: calculate relevant process/quality KPI (e.g., scrap rate, first pass yield, average quality check duration) for general monitoring and potential adjusting levers for process optimization.

To realize these goals, we decomposed the product quality inspection and KPI analytics pipeline $SPP_{Q,KPI}$ into eight individual pipeline elements (see Figure 5a).

Adapter sources. First, two instantiated ROS adapter sources ω_1^a and ω_2^a with a requirement to be able to access ROS connect to the UR5e control server to gather both the gripper stream as well as state stream.

Processors. Next, the input streams need to be merged, s.t. the gripper stream is enriched by the current robot state in ω_3^p needing a valid robot state (e.g., "picking start/end", "quality check start/end") as a data stream requirement. The enriched stream is then used by processor ω_4^p, where it is buffered in memory for the state duration deriving descriptive statistics (e.g., min, max, average, stddev, variance) of the force measurements as well as performing the actual quality check based on a user-defined threshold that is compared against the average force during the quality check. For the given use case, the domain experts determined a threshold value of 10 Newton to differentiate between "ok" or "not ok" checked parts.

$$Q(x) = \begin{cases} ok & \text{if } \bar{x}_{force} > 10N \\ nok & \text{otherwise} \end{cases}$$

where $\bar{x}$ is the mean value of force observations $\{x_1, \ldots, x_n\}$ within the quality check time interval ΔT^{qc} as shown in Figure 6. Lastly, based on the statistics, the quality check results as well as the time durations, we can calculate relevant KPI's, e.g., completed cycles, scrap total/rate, first pass yield total/rate and average process times, where we also formulate a data stream requirement, s.t. we expect the input stream to contain a valid quality test result.

Sinks. Lastly, the pipeline consists of three sinks. The quality test results are fed back to ROS in ω_5^s on the one hand also formulating a data stream requirement towards a valid quality test result, and on the other hand the requirement to be able to access the ROS running on the UR5e control server. Besides, the calculated KPI's are visualized in a central management dashboard ω_7^s and stored in a database ω_8^s.

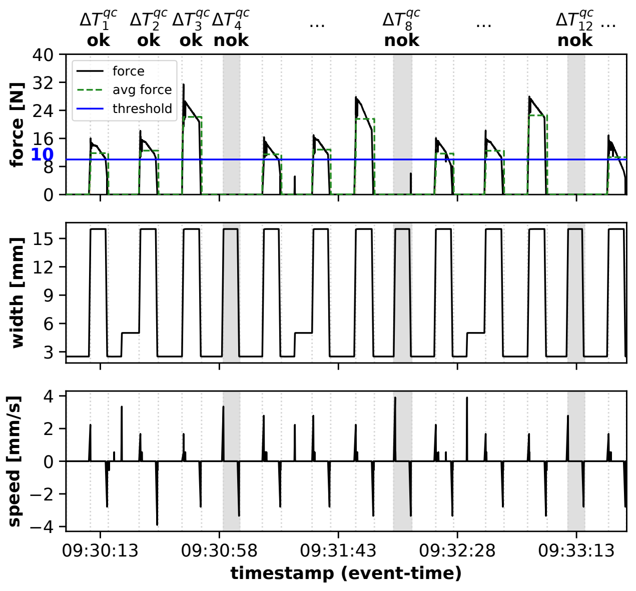


Figure 6: Excerpt of the first 12/99 performed quality checks executed by an UR5e with a gripper producing event streams (gripper force, width and speed). Defined quality criteria for the housing cover to pass is an average $\bar{x}_{force} > 10$ (see line) within the active time period of the quality check. Highlighted time intervals show "not ok" results.

6.2 Validation Setup

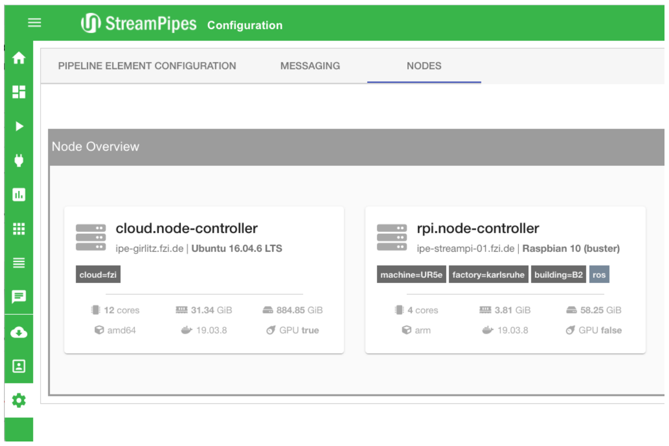
Data. To create a reproducible dataset while still embracing real-world shop floor data, we set up a configurable test bed in the context of collaborative robot-based parts assembly and quality inspection thereby repeatedly testing 4 parts in a rows, where the 4th part contains a quality defect, i.e. a missing spring in the housing cover as can be seen in force profile in Figure 6.[5] We were able to gather real measurements from a manufacturing plant over a significant time period, which were subsequently stored in a so-called ROS bag. For the sake of the evaluation, we cut out a slice of roughly 27 minutes of the original ROS bag containing 99 performed quality checks including gripper and state machine trigger event as shown in Table 3.

Table 3: UR5e+Gripper ROS bag measurements.

Topic	Events	Event Rate	Event Size
`/gripper`	45.458	28/sec	285 byte
`/trigger`	200	every 7sec	200 byte

Implementation. We implemented the proposed node model and architecture and integrated StreamPipes Edge Extensions in the open source framework Apache StreamPipes (incubating)[6]. Thus, every node controller that is deployed as a containerized StreamPipes

[5] Varying force profiles are due to differences in 3D printed gripper fingers.
[6] https://github.com/apache/incubator-streampipes/tree/edge-extensions

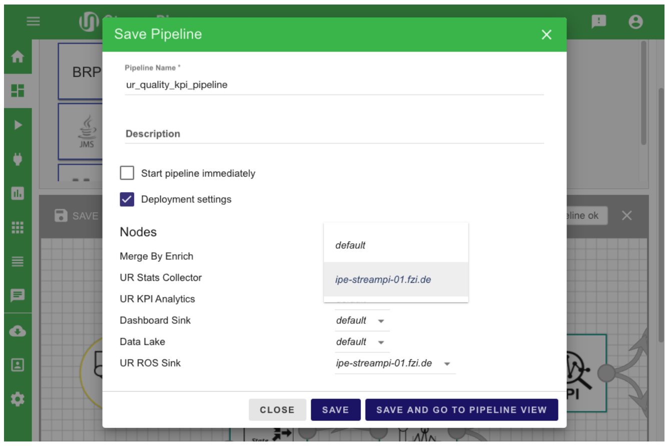


Node Overview

cloud.node-controller
ipe-girlitz.fzi.de | **Ubuntu 16.04.6 LTS**
cloud=fzi
12 cores | 31.34 GiB | 884.85 GiB
amd64 | 19.03.8 | GPU **true**

rpi.node-controller
ipe-streampi-01.fzi.de | **Raspbian 10 (buster)**
machine=UR5e | factory=karlsruhe | building=B2 | ros
4 cores | 3.81 GiB | 58.25 GiB
arm | 19.03.8 | GPU **false**

(a) Node overview in StreamPipes



Pipeline Name *
ur_quality_kpi_pipeline

Description

[ ] Start pipeline immediately
[✓] Deployment settings

Nodes

Merge By Enrich	default
UR Stats Collector	ipe-streampi-01.fzi.de
UR KPI Analytics	
Dashboard Sink	default
Data Lake	default
UR ROS Sink	ipe-streampi-01.fzi.de

(b) Deployment settings in StreamPipes

Figure 7: Implementation of our node model as part of the node overview (7a) in StreamPipes showing a two node cluster consisting of a cloud (server) node and one edge node (Raspberry Pi 4) including hardware and software related resource descriptions, semantic tags, e.g., `machine=UR5e`, as well as accessible sensor/actuator resources, e.g., `ros` as well as the deployment settings for target node selection (7b).

core service on target compute nodes registers itself as a new available node at the central StreamPipes backend in combination with its node metadata description. Relevant hardware resource information are gathered by using OSHI[7] and further complemented by using standard linux commands. Besides, we implemented a specific container orchestrator (`DockerOrchestratorManager`) to interface with the local Docker daemon to realize pipeline element life-cycle management (see Section 5.2), periodically clean dangling volumes and/or images, and retrieving relevant software information such as the availability of NVIDIA docker container runtime to also know wether to run containerized GPU workloads on a specific node. Other than that, we provide possibilities for system operators to attach semantic tags about the location or accessible sensor/actuator resources in the form of environment variables, with the result depicted in Figure 7a showing a three node cluster consisting of a cloud node and two edge nodes with corresponding semantic tags. In addition, we realized the pipeline $SPP_{Q,KPI}$ in StreamPipes as shown in Figure 5b. For instance, a configurable adapter was added to easily replay persisted data streams inside the ROS bag for various experimental settings.

Setup and Scenarios. For our validation, we use the following setup. We have a two node cluster consisting of one **cloud node** (x86_64), that is running Ubuntu 16.04, with 32GiB memory, 12 cores with 885GiB useful disk storage. Further, we use Raspberry Pi Model 4 (arm32) as an **edge node** running Raspbian Buster 10 with 4GiB memory, 4 cores with 59GiB useful disk storage. Besides the two nodes forming the cluster, we use a Raspberry Pi 3 Model B+ to replay our ROS bag thus simulation the native ROS control server unit. StreamPipes core container such as the pipeline management backend and UI were pre-downloaded and installed on the cloud node. The individual pipeline element runtime container of $SPP_{Q,KPI}$ are both downloaded on cloud node and the edge node. As part of the current development in StreamPipes we also integrated support for the MQTT message protocol. Hence, we use Eclipse mosquitto[8] as a node broker. For our experimental settings, we consider the following mapping of the $SPP_{Q,KPI}$ pipeline on the two node cluster:

Table 4: Mapping of $SPP_{Q,KPI}$ on two node cluster.

Node	Pipeline Elements
edge node	**adapter sources** (ROS gripper, state stream) **processors** (enrich, stats & quality check) **sink** (ROS)
cloud node	**processor** (KPI calculation)[1] **sinks** (dashboard, datalake)

Table 5: Experimental scenarios.

Name	Scenario
cluster-local	local network
cloud-edge	200 ± 10ms delay
cloud-edge++	cloud-edge + KPI processor on edge node

For our tests, we define 3 scenarios (see Table 5). In the *cluster-local* scenario we setup a local cluster deployment, where both the edge and cloud node reside on the (company) network with an average round trip time of 0.29ms over 50 ping runs. Both in the *cloud-edge* as well as *cloud-edge++* scenario, we add a delay between the two nodes of 200ms with random ±10ms uniform distribution

[7]https://github.com/oshi/oshi

[8]https://mosquitto.org/

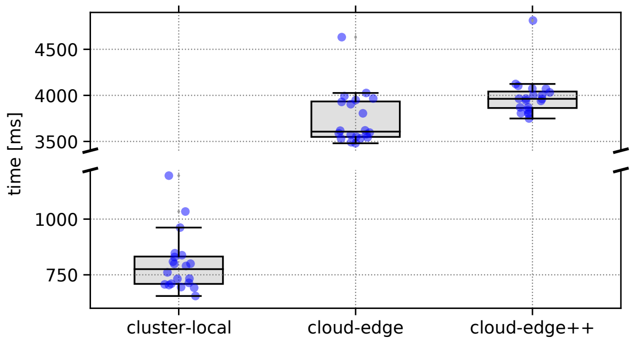


Figure 8: $SPP_{Q,KPI}$ **pipeline starting times.**

by using Linux traffic control leading to a realistic setup, while in the latter as opposed to the initial mapping we now also run the KPI calculation on the edge node leaving only the dashboard and datalake sink on the cloud node.

In all three scenarios, StreamPipes core container are running on the cloud node. Thereby, pipeline modeling and deployments are triggered manually by respecting the mapping requirements of Table 4 and selecting the dedicated target nodes for pipeline element invocation as shown in Figure 7b. We evaluate two central questions:

- How does the delay affect pipeline starting times (here based on the $SPP_{Q,KPI}$)?
- How much load in terms of memory and CPU consumption is put on the nodes in total and how much is consumed by the node controller container?

6.3 Results and Discussion

We measured the time difference between triggered deployment requests and the succeeding response messages from all involved pipeline elements over 20 runs. Thereby, an element specific invocation graph containing detailed information such as the transport protocol (here MQTT) and topic name to subscribe/publish to is sent to each pipeline element from the pipeline management, where the runtime wrapper for the element is running. In our evaluation setup, this graph represents an average payload of 21kB. Results for the pipeline starting times are shown in Figure 8.

As suspected, we see an increase in overall starting times due to the introduced delay. In the ideal world of the *cluster-local* scenario, we measured a median starting time of 775ms, we calculated a median of 3606ms in *cloud-edge* and 3962ms in *cloud-edge++*. In the latter, the slight increase occurs due to the additional network request to invoke the KPI processor on the edge node. Over all scenarios, there are some outliers being disproportionately higher than other measurements that are representing the first test run, where the complex invocation graph structure is built and then cached for subsequent uses. While the current implementation of StreamPipes and the node controller only accounts for static pipeline element deployment, the results provide valuable estimates as the foundation for extensions towards dynamic pipeline element relocation to other target node locations during runtime in future work.

Further, for each scenario we used the ROS bag and replayed it thereby collecting current resource status information in terms of memory and CPU usage on (i) node-level by calling the `/status` endpoint of the resource manager built-in node controller every 5 sec, as well as (ii) container-level using Docker stats[9] feature in order to gain insights on the dedicated node controller resource consumption on a more fine-grained level. The results are respectively shown in Figure 9a as well as Figure 9b. On node level, we clearly see a separation of consumed CPU and memory between cloud and edge node due to the varying hardware capabilities, however, between the evaluated scenarios there is merely a difference noticeable.

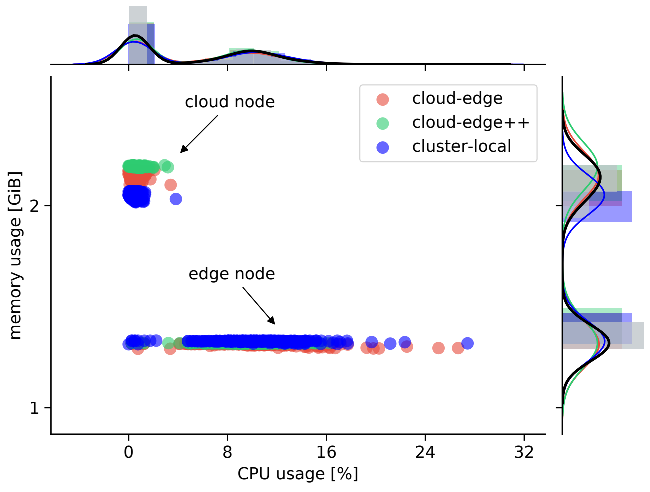
The average CPU consumption on the cloud node (0.5%) as well as its average memory consumption (2.13GiB) remains relatively stable throughout pipeline execution. In addition, the average memory usage on the edge node is also staying at a steady level (1.32GiB). Only the CPU consumption is varying (up to 27.4%) with an average at around 10%. The main influencing factor are the source adapters that constantly stream data from ROS as well as the processing in terms of calculating the descriptive statistics over the buffered events (495 byte buffer containing 200 − 300 gripper events) at the end of each quality check.

The node controller was designed to be lightweight in terms of consumed resources in order to be also well-suited for resource limited edge nodes that reflects in Figure 9b. Both, the CPU and memory footprint on cloud node and edge node (Raspberry Pi) are at a low level with a median CPU consumption of 0.86% on the edge node and 0.22% on the cloud node. Besides some outliers on the edge node, median memory usage remains stable over all scenarios for the edge (194.5MiB) and cloud node (144.4MiB). We consider the actual load to increase slightly in the course of integrating more functionality into the node controller itself that also allow for decentralized pipeline element management when the central pipeline management is not available due to network partitions.

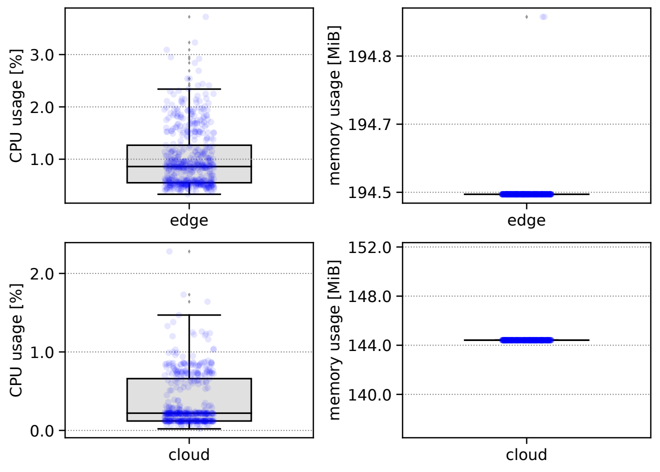
6.4 Industry Adoption

Apache StreamPipes (incubating) was initially developed by the authors of this paper in multiple research as well as industry projects over the last couple of years. Recently it was handed over to the Apache Software Foundation as an incubating project with the goal to grow a bigger and diverse community. The project has over 3.300 installations and is already used in multiple manufacturing companies to monitor the production process. New and innovative ideas and features (as presented in this paper) are still developed in academia with the goal to integrate them into to the production grade tool once the developed concepts are properly evaluated. The basic idea for the contribution of this paper emerged from requirements that arose in multiple industrial projects, where a flexible solution for the management of stream processing pipelines in edge processing scenarios was required. Those companies often have many different machines and production lines which are distributed all over the globe. This makes it very hard to manage the deployment of analytics tasks. A flexible semi-automated solution for dynamic edge deployments is required which is capable to adapt to changes in the IIoT infrastructure as well in the performed analyses.

[9]Currently, the node controller does not collect stats exposed by the Docker daemon.



(a) cloud and edge node



(b) node controller container

Figure 9: Consumed resources (CPU, memory usage) on the two node cluster during $SPP_{Q,KPI}$ execution for the scenarios *cluster-local, cloud-edge, cloud-edge++* (9a) as well as the consumed resources by the node controller container as an aggregate over all scenarios (9b).

7 RELATED WORK

Distributed stream processing have been broadly studied in both the industrial as well as the academic research community with emerging novel application areas, such as the IoT [8]. In [12], an extension of distributed dataflow programming paradigm for Node-RED is proposed thereby providing mechanism to deploy defined flows over heterogeneous compute resources by specifying static requirements and constraints. In [31] a framework for dynamic resource provisioning and automated container-based application deployment is proposed, presenting a more fine-grained description of requirements including prioritization to enable preemption as well as privacy constraints in terms of the actual placement in the infrastructure hierarchy. A container-based architecture for supporting autonomic data stream processing applications on fog computing infrastructure is shown in [5], yet only exploiting native Docker features to scale and migrate application containers. In [20], an edge-based programming framework that allows users to define how data streams are processed based on the content and the location of the data is introduced. In addition, in [24] a programming infrastructure for the geo-distributed computational continuum is presented, that manages the application components in a situation-aware manner. While generally applications are also deployed and managed using a container orchestrator, the proposed approach lacks managing multi-component applications as it is the case in our presented application model of stream processing pipelines. In [16] an approach based on the declarative TOSCA standard for the automated deployment of distributed applications on heterogeneous target environments consisting of public and private clouds is shown, thereby tackling the issue of deploying components in environments having restricted inbound communication capabilities. While the presented approach greatly addresses the accessibility and security aspects especially in IIoT application deployments, it does not account for dependent multi-component application deployments regarding stream processing pipelines. Further, [18] investigates complex event processing over fog infrastructures and present ProgCEP, a programming model facilitating the development of operator placement algorithms. In our previous work we presented a conceptual architecture to support context-aware, dynamic management of stream processing pipelines in the fog [30]. Besides, there exists numerous other works in the area of application orchestration and deployment such as FogFrame [25], Foggy [23], FogFlow [6], Fogsy [29].

Besides the research community, several companies offer fog computing services and tools for application orchestration and deployment, such as Google Cloud IoT, Amazon Greengrass, Microsoft Azure IoT Edge, IBM Edge Computing, Crosser [7], Foghorn [10] or Nebbiolo [19]. While these services provide the possibility to realize edge deployments, they are not targeted towards flexible deployment of stream processing pipelines. Existing open source container cluster management orchestrators that are designed to address challenges evolving around edge to cloud deployments either based on/or extending Kubernetes such as kubeEdge [17], k3s [14] or ioFog [13] focus mainly on the infrastructure level and are not tailored to streaming applications.

Overall, the aforementioned approaches neglect providing domain experts with a solution to easily deduce meaningful insights within their industry domain. To the best of our knowledge, existing approaches lack a generic and conceptual node model describing node resources and metadata information and do not account for the unique challenges of managing industrial stream processing pipelines over the underlying heterogeneous infrastructure. In our case, these pipelines consist of multiple decomposed but dependent analytical functions that need to managed in a holistic manner thereby still allowing domain experts to easily model and deploy applications in a self-service manner while abstracting them from the orchestration, deployment and monitoring of the containerized pipeline elements.

8 CONCLUSION

In this paper, we presented StreamPipes Edge Extensions, a novel contribution to the open source IIoT toolbox Apache StreamPipes that allows domain experts to create stream processing pipelines and assign individual pipeline elements to available, geo-graphically distributed nodes. We introduced an application model for stream processing pipelines as well as a fog cluster resource model representing capabilities of compute nodes. Besides, we presented the node controller architecture of SEE for pipeline element life cycle and node management to realize edge deployments of individual pipeline elements. We validated our approach in a real industrial setup involving a collaborative robot and demonstrated the feasibility of edge deployments with the result of low overall resource overhead for the node controller.

As part of our future work, we will switch from manual node selection upon startup of a pipeline to an advanced deployment approach by realizing certain deployment strategies, e.g., minimum-latency, or maximum-edge utilization, and also focus on allowing to offload pipeline elements to other suitable cluster nodes at runtime based on changes in the requirements or infrastructural context (e.g., mobile edge nodes, node maintenance, etc).

REFERENCES

[1] Tayebeh Bahreini and Daniel Grosu. 2017. Efficient Placement of Multi-Component Applications in Edge Computing Systems. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing (SEC '17)*. Association for Computing Machinery, San Jose, California, 1–11. https://doi.org/10.1145/3132211.3134454

[2] S. Bhattacharjee. 2018. *Practical Industrial Internet of Things Security: A Practitioner's Guide to Securing Connected Industries*. Packt Publishing. https://books.google.de/books?id=ZO1mDwAAQBAJ

[3] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. 2012. Fog Computing and Its Role in the Internet of Things. *Proceedings of the first edition of the MCC workshop on Mobile cloud computing* (2012), 13–16. https://doi.org/10.1145/2342509.2342513 arXiv:1502.01815v3

[4] Antonio Brogi, Stefano Forti, Carlos Guerrero, and Isaac Lera. 2019. How to Place Your Apps in the Fog – State of the Art and Open Challenges. *Software: Practice and Experience* 50, 5 (Nov. 2019), 719–740. https://doi.org/10.1002/spe.2766 arXiv:1901.05717

[5] Antonio Brogi, Gabriele Mencagli, Davide Neri, Jacopo Soldani, and Massimo Torquati. 2018. Container-Based Support for Autonomic Data Stream Processing Through the Fog. *Euro-Par 2017: Parallel Processing Workshops* (2018), 17–28. https://doi.org/10.1007/978-3-319-75178-8_2

[6] Bin Cheng, Gürkan Solmaz, Flavio Cirillo, Ernö Kovacs, Kazuyuki Terasawa, and Atsushi Kitazawa. 2018. FogFlow: Easy Programming of IoT Services Over Cloud and Edges for Smart Cities. *IEEE Internet of Things Journal* 5, 2 (April 2018), 696–707. https://doi.org/10.1109/JIOT.2017.2747214

[7] Crosser.io. 2020. *Official crosser website*. Retrieved May 28, 2020 from https://crosser.io/

[8] Miyuru Dayarathna and Srinath Perera. 2018. Recent Advancements in Event Processing. *Comput. Surveys* 51, 2 (Feb. 2018), 33:1–33:36. https://doi.org/10.1145/3170432

[9] Koustabh Dolui and Soumya Kanti Datta. 2017. Comparison of Edge Computing Implementations: Fog Computing, Cloudlet and Mobile Edge Computing. In *2017 Global Internet of Things Summit (GIoTS)*. IEEE, Geneva, Switzerland, 1–6. https://doi.org/10.1109/GIOTS.2017.8016213

[10] FogHorn. 2020. *Official FogHorn website*. Retrieved May 28, 2020 from https://www.foghorn.io/

[11] Erich Gamma, Richard Helm, Johnson Ralph, and John Vlissides. 1995. *Design Patterns : Element of Reusable Object Oriented Software.*

[12] Nam Ky Giang, Michael Blackstock, Rodger Lea, and Victor C.M. Leung. 2015. Developing IoT Applications in the Fog: A Distributed Dataflow Approach. In *Proceedings - 2015 5th International Conference on the Internet of Things, IoT 2015.* IEEE, 155–162. https://doi.org/10.1109/IOT.2015.7356560

[13] ioFog. 2020. *Official ioFog website*. Retrieved May 28, 2020 from https://iofog.org/

[14] K3s. 2020. *Official k3s website*. Retrieved May 28, 2020 from https://k3s.io/

[15] Amir Karamoozian, Abdelhakim Hafid, and El Mostapha Aboulhamid. 2019. On the Fog-Cloud Cooperation: How Fog Computing Can Address Latency Concerns of IoT Applications. In *2019 Fourth International Conference on Fog and Mobile Edge Computing (FMEC)*. IEEE, Rome, Italy, 166–172. https://doi.org/10.1109/FMEC.2019.8795320

[16] Kálmán Képes, Uwe Breitenbücher, Frank Leymann, Karoline Saatkamp, and Benjamin Weder. 2019. Deployment of Distributed Applications Across Public and Private Networks. In *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*. 236–242. https://doi.org/10.1109/EDOC.2019.00036

[17] KubeEdge. 2020. *Official KubeEdge website*. Retrieved May 28, 2020 from https://kubeedge.io/

[18] Manisha Luthra and Boris Koldehofe. 2019. ProgCEP: A Programming Model for Complex Event Processing over Fog Infrastructure. In *Proceedings of the 2nd International Workshop on Distributed Fog Services Design - DFSD '19*. ACM Press, Davis, CA, USA, 7–12. https://doi.org/10.1145/3366613.3368121

[19] Nebbiolo. 2020. *Official Nebbiolo website*. Retrieved May 28, 2020 from https://www.nebbiolo.tech/

[20] Eduard Gibert Renart, Javier Diaz-Montes, and Manish Parashar. 2017. Data-Driven Stream Processing at the Edge. In *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*. IEEE, Madrid, Spain, 31–40. https://doi.org/10.1109/ICFEC.2017.18

[21] Dominik Riemer, Ljiljana Stojanovic, and Nenad Stojanovic. 2014. SEPP: Semantics-Based Management of Fast Data Streams. In *Proceedings - IEEE 7th International Conference on Service-Oriented Computing and Applications, SOCA 2014*. IEEE, 113–118. https://doi.org/10.1109/SOCA.2014.52

[22] Farah Ait Salaht, Frédéric Desprez, and Adrien Lebre. 2019. An Overview of Service Placement Problem in Fog and Edge Computing. *[Research Report] RR-9295, Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, LYON, France. 2019, pp.1-43. ffhal-02313711v2ff* (2019), 47.

[23] Daniele Santoro, Daniel Zozin, Daniele Pizzolli, Francesco De Pellegrini, and Silvio Cretti. 2017. Foggy: A Platform for Workload Orchestration in a Fog Computing Environment. In *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*, Vol. 2017-Decem. 231–234. https://doi.org/10.1109/CloudCom.2017.62 arXiv:1810.00179

[24] Enrique Saurez, Kirak Hong, Dave Lillethun, Umakishore Ramachandran, and Beate Ottenwälder. 2016. Incremental Deployment and Migration of Geo-Distributed Situation Awareness Applications in the Fog. In *Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems - DEBS '16*. ACM Press, New York, New York, USA, 258–269. https://doi.org/10.1145/2933267.2933317

[25] Olena Skarlat, Vasileios Karagiannis, Thomas Rausch, Kevin Bachmann, and Stefan Schulte. 2018. A Framework for Optimization, Service Placement, and Runtime Operation in the Fog. In *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 164–173. https://doi.org/10.1109/UCC.2018.00025

[26] Olivier Terzo, Karim Djemame, Alberto Scionti, and Clara Pezuela. 2019. *Heterogeneous Computing Architectures: Challenges and Vision*. CRC Press.

[27] Universal Robots. 2020. *Official Universal Robots website*. Retrieved May 28, 2020 from https://www.universal-robots.com/

[28] Shiqiang Wang, Murtaza Zafer, and Kin K. Leung. 2017. Online Placement of Multi-Component Applications in Edge Computing Environments. *IEEE Access* 5 (2017), 2514–2533. https://doi.org/10.1109/ACCESS.2017.2665971

[29] Patrick Wiener, Zehnder Philipp, Heyden Marco, Philipp Patrick, and Riemer Dominik. 2020. *Fogsy: Towards Holistic Industrial AI Management in Fog and Edge Environments*. [Technical Report] KuVS-Fachgespräch Fog Computing 2020. Wien, Essen.

[30] Patrick Wiener, Philipp Zehnder, and Dominik Riemer. 2019. Towards Context-Aware and Dynamic Management of Stream Processing Pipelines for Fog Computing. In *2019 IEEE 3rd International Conference on Fog and Edge Computing (ICFEC)*. 1–6. https://doi.org/10.1109/CFEC.2019.8733145

[31] Emre Yigitoglu, Mohamed Mohamed, Ling Liu, and Heiko Ludwig. 2017. Foggy: A Framework for Continuous Automated IoT Application Deployment in Fog Computing. In *Proceedings - 2017 IEEE 6th International Conference on AI and Mobile Services, AIMS 2017*. 38–45. https://doi.org/10.1109/AIMS.2017.14

[32] Ashkan Yousefpour, Caleb Fung, Tam Nguyen, Krishna Kadiyala, Fatemeh Jalali, Amirreza Niakanlahiji, Jian Kong, and Jason P. Jue. 2018. All One Needs to Know about Fog Computing and Related Edge Computing Paradigms: A Complete Survey. (Aug. 2018). arXiv:1808.05283 http://arxiv.org/abs/1808.05283

[33] Philipp Zehnder, Patrick Wiener, Tim Straub, and Dominik Riemer. 2020. StreamPipes Connect: Semantics-Based Edge Adapters for the IIoT. In *The Semantic Web*, Andreas Harth, Sabrina Kirrane, Axel-Cyrille Ngonga Ngomo, Heiko Paulheim, Anisa Rula, Anna Lisa Gentile, Peter Haase, and Michael Cochez (Eds.). Springer International Publishing, Cham, 665–680.