Managing Event-Driven Applications in Heterogeneous Fog Infrastructures

Zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften
(Dr.-Ing.)

von der KIT-Fakultät für Wirtschaftswissenschaften
des Karlsruher Instituts für Technologie (KIT)

genehmigte
DISSERTATION

von

M.Sc. Patrick Wiener

Tag der mündlichen Prüfung:	14. Februar 2022
Referent:	Prof. Dr. York Sure-Vetter
Korreferent:	Prof. Dr. Jens Nimis

Karlsruhe 2022

Abstract

The steady increase in digitalization propelled by the Internet of Things (IoT) has led to a deluge of generated data at unprecedented pace. Thereby, the promise to realize data-driven decision-making is a major innovation driver in a myriad of industries. Based on the widely used event processing paradigm, *event-driven applications* allow to analyze data in the form of event streams in order to extract relevant information in a timely manner. Most recently, graphical flow-based approaches in no-code event processing systems have been introduced to significantly lower technological entry barriers. This empowers non-technical *citizen technologists* to create event-driven applications comprised of multiple interconnected event-driven processing services. Still, today's event-driven applications are focused on centralized cloud deployments that come with inevitable drawbacks, especially in the context of IoT scenarios that require fast results, are limited by the available bandwidth, or are bound by the regulations in terms of privacy and security. Despite recent advances in the area of *fog computing* which mitigate these shortcomings by extending the cloud and moving certain processing closer to the event source, these approaches are hardly established in existing systems. Inherent fog computing characteristics, especially the heterogeneity of resources alongside novel application management demands, particularly the aspects of geo-distribution and dynamic adaptation, pose challenges that are currently insufficiently addressed and hinder the transition to a next generation of no-code event processing systems.

The contributions of this thesis enable citizen technologists to manage event-driven applications in heterogeneous fog infrastructures along the application life cycle. Therefore, an approach for a holistic application management is proposed which abstracts citizen technologists from underlying technicalities. This allows to evolve present event processing systems and advances the democratization of event-driven application management in fog computing. Individual contributions of this thesis are summarized as follows:

1. A model, manifested in a geo-distributed system architecture, to semantically describe characteristics specific to node resources, event-driven applications and their management to blend application-centric and infrastructure-centric realms.
2. Concepts for geo-distributed deployment and operation of event-driven applications alongside strategies for flexible event stream management.
3. A methodology to support the evolution of event-driven applications including methods to dynamically reconfigure, migrate and offload individual event-driven processing services at run-time.

The contributions are introduced, applied and evaluated along two scenarios from the manufacturing and logistics domain.

Contents

Figures

Tables

Listings

Abbreviations

EC	Event consumer
EDA	Event-driven architecture
EP	Event producer
EPA	Event processing agent
EPN	Event processing network
IIoT	Industrial Internet of Things
IoT	Internet of Things
LAEDS	Locality-aware event dissemination strategy
MAPE	Monitor-Analyze-Plan-Execute
SPR	Sense-process-respond

Part I

Introduction

1 Introduction

The steady increase in digitalization efforts has led to a deluge of generated data with the *Internet of Things* (IoT) acting as a key innovation driver, in particular in domains such as smart city, manufacturing, energy, or logistics. Never before have deployed sensors and smart devices been able to produce new real-time data as prolifically as experienced today. Especially in industrial environments, the adoption of the *Industrial Internet of Things* (IIoT) which refers to a network of interconnected sensors, devices, and other industrial assets [Jeschke et al. 2017], offers substantial opportunities and attracts businesses to invest in order to harvest new data sources. Alongside the promise to realize data-driven decision-making, companies have great expectations in achieving added value through improvements in product and process quality, the development of new digital business models and establishing competitive advantages. Yet, the added business value does not result from the raw data itself but from the information gained from a real-time data analysis process.

In this regard, event processing is a widely used processing paradigm to analyze data in the form of event streams to extract relevant information in a timely manner. To lower the rather high technological entry barriers into event processing, graphical flow-based approaches employed in no-code event processing systems have been proposed [Kleinfeld et al. 2014; Riemer et al. 2014; Noor et al. 2019]. The visual modeling approach makes event processing accessible for non-technical domain specialists and allows them to create applications in a self-service manner. Such *event-driven applications* are described as processing pipelines which consist of multiple, loosely-coupled and self-contained event-driven processing services. Based on the conceptual event processing network model, the individual event-driven processing services represent a collection of event producers, event processing agents and event consumers which shape the pipeline structure [Sharon and Etzion 2007]. In this matter, distributed event-based systems have been prevalent for realizing event-driven applications at large-scale [Carzaniga et al. 1998] which has led to the general concepts and principles of the event-driven architecture [Bruns and Dunkel 2010]. The presented characteristics make event-driven applications well suited for a broad application in a myriad of IIoT-related scenarios. Typical use cases include, among others, continuous monitoring, complex event processing or anomaly detection. However, today's event-driven applications are static and focused on centralized cloud deployments. Over the past decade, the cloud has acted as the de facto computing environment

for multiple reasons, including the ubiquitous, on-demand access to network-centric services such as virtualized resources that can be provisioned and released with minimal management effort to allow for dynamic scalability [Tai et al. 2010; Baun et al. 2011]. However, there are some inevitable drawbacks to the centralized approach in view of emerging IoT and IIoT scenarios that require time-sensitive decisions, are limited in available bandwidth [Satyanarayanan et al. 2009], or are bound by regulations with respect to privacy and security [Zhou et al. 2017]. Unclear development of costs and concerns about third-party handling of sensitive data further intensify the shortcomings of the centralized cloud model [Vaquero and Rodero-Merino 2014; Sunyaev 2020]. These limitations have driven the development of a novel decentralized computing paradigm referred to as *fog computing* [Bonomi et al. 2012]. By extending the cloud, fog computing envisions to move certain processing steps and their management closer to the edge of the network, i.e., closer to the event source. Despite these advantages, fog computing so far finds little adoption in practice [Bermbach et al. 2018].

From an infrastructure perspective, some of the reasons are the inherent fog computing characteristics [Bonomi et al. 2012; Iorga et al. 2018]. Apart from mobility aspects and the exposure to the physical environment, the aspects of resource heterogeneity in particular are poorly addressed in modern event processing systems. From an event processing perspective, managing geo-distributed event-driven-applications is complex and requires deep technical knowledge from multiple domains. Moreover, the static nature of current event-driven applications is not suitable to account for the dynamics in fog computing environments and to evolve the application to new analytical and business needs. On the one hand, this includes topological changes which characterize *where* along the cloud-edge continuum the processing is executed. On the other hand, this also involves the modification of processing semantics describing *how* events are processed. These problems are further exacerbated by the recent paradigm shift in enterprises towards establishing a data-driven culture that requires appropriate platform support. Here, the term *citizen technologists* is used to describe a set of newly emerging organizational roles that make data-driven decisions in industrial business processes by combining profound domain knowledge with basic IT and data analysis skills [Gröger 2018]. Technical problems have been investigated in various domains, including (complex) event processing, distributed event-based systems, or application orchestration. Still, there lacks a holistic management approach for geo-distributed event-driven applications with fog computing as an enabler to usher the next generation of no-code event processing systems to be leveraged by citizen technologists.

By combining both application-centric and infrastructure-centric worlds into a holistic approach, this embraces citizen technologists to go beyond existing boundaries which are hindered by the capabilities of today's systems. This not only provides citizen technologists with the ability to centrally model event-driven applications, but to self-reliantly manage them in heterogeneous fog infrastructures along the applicational life cycle without much effort.

In the course of this thesis, we propose enhancements to the next generation of no-code event processing systems in fog computing which allow to drive the democratization movement from IT experts to citizen technologists. In view of this, a holistic approach to managing geo-distributed event-driven applications in heterogeneous fog computing infrastructures is developed which pursues the following objectives: First, creating heterogeneity-awareness for the event-driven application to adequately exploit computational resources independent from the underlying infrastructure. Second, deploying and operating event-driven applications in geographically dispersed fog environments. Third, adapting event-driven applications to accommodate both the dynamics of fog computing and the changes in domain requirements.

1.1 Research Questions

Our research aims to provide a holistic application management approach to a next generation of no-code event processing systems in fog computing. This empowers citizen technologists with little or no software engineering and infrastructure management knowledge to perform the following tasks: (1) deploy geo-distributed event-driven applications on heterogeneous computational resources spanning across the cloud-edge continuum while freely configuring deployment targets, (2) operate individual event-driven processing services in geo-distributed fog computing infrastructures along their application life cycle and (3) modify employed application-specific logic or execution targets of event-driven processing services at run-time to account for changing requirements. Consequently, this leads to the following principal research question:

How can citizen technologists be enabled to manage event-driven applications in geographically distributed fog computing infrastructures?

The principal research question comprises three key aspects that need more elaboration: *event-driven application*, *citizen technologist*, and *fog computing*.

Event processing allows to continuously process event streams by applying application-specific logic while data is in motion. Often, this follows an event-driven architecture with individual, loosely-coupled event-driven processing services communicating in a publish/subscribe manner. The composition of these event-driven processing services forms the *event-driven application*, also referred to as processing pipeline. More recently, no-code development platforms have started to democratize the development of event-driven applications by means of graphical flow-based modeling approaches [Riemer 2016]. This has significantly lowered rather high technological entry barriers making it particularly appealing for organizational roles such as *citizen technologists* to create new business applications for consumption by others [Wong et al. 2015; Gröger 2018]. Finally, *fog computing* describes a decentralized compute infrastructure organized in a layered,

hierarchical architecture from the local edge layer closest to real-world IoT devices, over a regional fog layer to the global cloud layer [Bonomi et al. 2014]. We argue that fog computing serves as an enabler for the next generation of no-code event processing systems as it allows to move certain processing in closer proximity to the event source by extending the cloud towards the edge of the network. Such event processing systems not only provide citizen technologists with the ability to create event-driven applications, but also assist them in managing these applications in geo-distributed fog environments, so that they can focus on the actual domain problem [Fischer et al. 2009].

The principal question itself can be further broken down into three sub-questions, each targeting different aspects of the main research question. Each sub-question is motivated in Section 3.3.2.

Research Question 1 (Exploit)**.** *How can event-driven applications exploit heterogeneous computational resources in fog computing infrastructures?*

The first research question investigates resource management matters of event-driven applications in fog computing infrastructures. As fog computing extends the cloud resource layer towards the network edge, event processing systems are faced with constrained and heterogeneous resources driven by specialization and dispersion of custom hardware [Terzo et al. 2019]. The main challenge of this research question involves the development of a generic and extensible model which allows to link infrastructure-related aspects (e.g., resource offers) and application-related aspects (e.g., resource requirements) alongside relevant deployment and operation considerations for event-driven applications. The model is incorporated in a holistic application management approach and serves as a foundational building block to support the platform-independent deployment and operation of event-driven processing services in a heterogeneity-aware manner. The research question is answered in Chapter 6 and evaluated in Chapter 9.

Research Question 2 (Deploy)**.** *How can we deploy and operate event-driven applications that span multiple geographically distributed nodes?*

The second research question covers aspects of deployment and operation of individual event-driven processing services which we refer to as *pipeline elements*. A composition of logically connected pipeline elements form the actual processing pipeline that may span multiple geographically distributed computational resources within the fog infrastructures. Hereby, occurring challenges are manifold: On the one hand, citizen technologists must be given some degree of freedom in selecting appropriate deployment locations for pipeline elements in order to incorporate their domain and application knowledge. On the other hand, apart from sufficient support for citizen technologists in configuring deployment locations, concepts for geo-distributed deployment and operation are required. Further, a management service on node level needs to be developed evolving around the pipeline element life cycle which also allows the coordination of complex data flows in fog infrastructures. The research question is answered in Chapter 7 and evaluated in Chapter 9.

Research Question 3 (Adapt)**.** *How can we reconfigure and relocate existing event-driven processing services at run-time?*

Due to the nature of event streams, event-driven applications typically execute for an indefinite amount of time. As a result, an application's ability to adapt to changing conditions or requirements is crucial for its successful operation [Andrade et al. 2014]. The research question addresses two main adaptation aspects. The first part deals with the modification of design-time decisions for user-provided pipeline element configurations at run-time to reconfigure the processing behavior. The second part works towards run-time displacements of individual pipeline elements from their original to a new node execution target. Developing a generic methodology that covers both adaptation aspects while incorporating human interaction is beneficial for the next generation of no-code event processing systems to facilitate fast information retrieval at minimal effort. The research question is answered in Chapter 8 and evaluated in Chapter 9.

1.2 Research Methodology

The underlying research methodology in this thesis follows the design science paradigm in information system research [Hevner et al. 2004]. The conceptual framework of the design science paradigm derives business needs from a specific environment, e.g., people, organizations, and technologies, defining the problem domain. The objective behind information system research is to build and evaluate purposeful IT artifacts which satisfy articulated needs by systematically applying existing foundations and methodologies from a knowledge base [Hevner et al. 2004]. Thereby, the notion of artifacts is not strictly limited to software or hardware artifacts, but also includes constructs, models and methods utilized in the course of the development and use of information systems. Hevner et al. suggest a systematic problem solving approach for research conducting the design science paradigm which relies on seven key guidelines, namely **(G1)** *Design as an Artifact*, **(G2)** *Problem Relevance*, **(G3)** *Design Evaluation*, **(G4)** *Research Contributions*, **(G5)** *Research Rigor*, **(G6)** *Design as a Search Process* and **(G7)** *Communication of Research*.

In this thesis, all seven key guidelines are considered. Various artifacts in the form of models, concepts and methods apart from software artifacts are discussed in Chapters 6 to 8 to address related research questions (**G1**). In Chapter 3, we conduct thorough studies of the problem domain, identify relevant business needs and state current limitations in view of the problem relevance (**G2**). We provide extensive design evaluation focusing on different aspects of the developed artifacts (**G3**). Therefore, the research contributions of this thesis are models, concepts and methods which are integrated into a software artifact realizing a holistic application management system as a middleware for no-code development environments in order to deploy and operate event-driven applications in heterogeneous fog computing infrastructures (**G4**). Moreover, we present

foundations in Chapter 2, discuss existing methodologies as part of related work in Chapter 4, and derive requirements associated with constructing the artifact in Chapter 5 which are complemented in Chapter 9 by providing evaluation results (**G5**, **G6**). Both research contributions and evaluation results (**G7**) have been presented and discussed at various venues both to expert academics and practitioners of technology-oriented and management-oriented audience which we further detail in the next section.

1.3 Contributions and Impact

The contributions of this thesis follow the outlined research questions in Section 1.1 and provide both conceptual and methodological foundations for managing event-driven applications in heterogeneous fog infrastructures.

(C1) **Model and Architecture**. We present a generic model describing node-specific resource characteristics complemented by additional metadata containing in-depth information such as the dedicated node type or the location. As we discuss, there are various heterogeneity dimensions to be considered regarding resource management matters in fog computing. Thus, the goal is to provide a lightweight and extensible description explicitly expressing node resource offers that builds the foundation to assist citizen technologists through the actual deployment process by internally verifying selected deployment configurations with dedicated node resource requirements from the event-driven application side. In this regard, we re-use and extend vocabularies of previous work [Riemer 2016; Zehnder et al. 2020] to incorporate additional concepts in view of platform and technology-agnostic geo-distributed deployment and operation. Moreover, we suggest a geo-distributed system architecture according to derived requirements that builds on top of state-of-the-art application management approaches and extend them in terms of heterogeneity-awareness. This contribution is related to Research Question 1.

(C2) **Concepts**. We propose deployment and operation concepts for geo-distributed event-driven applications in fog infrastructures. This combines both notions of central orchestration as well as local execution and coordination along the life cycle of respective pipeline elements. To this extent, we instantiate the aforementioned node model as part of the design of a local node controller service which is capable to automatically extract and expose node-specific resource information at startup while providing the possibility to modify or add metadata at any later stage. Further, the node controller service applies a flexible locality-aware event dissemination strategy for event streams between any two adjacent but dislocated pipeline elements. This represents a crucial task to realize a geo-distributed operation as it avoids expensive network round trips to remote message brokers. Moreover, citizen technologists are offered to chose among different deployment and operation options that best align with their use case. This contribution is related to Research Question 2.

(C3) Methodology. We introduce an adaptation methodology which uses side-inputs to pipeline elements as a key abstraction to inject special control directives in the form of adaptation events. This facilitates the run-time evolution of deployed event-driven applications in terms of two specific adaptation types. On the one hand, this includes the run-time reconfiguration to alter pipeline element configurations from design-time. On the other hand, this denotes the run-time relocation of pipeline elements. In the latter case, this includes migration actions that are actively performed by citizen technologists in view of changing requirements apart from autonomic offloading actions that are triggered by the system to account for contextual changes. This contribution is related to Research Question 3.

Research Projects and Publications. The main contributions of this thesis result from the work in several German-funded research and industry projects that were presented in peer-reviewed publications at international top-tier venues which we describe in the following.

BigGIS (Scalable Geographic Information Systems for Predictive and Prescriptive Analytics, 04/2015-03/2018, BMBF)

The goal behind BigGIS was to develop a scalable geographic information system that supports decision-making in a multitude of IoT-related use cases including disaster management using airborne data from unmanned aerial vehicles. This required processing of large and heterogeneous spatio-temporal data from (potentially) unreliable sources. Thereby, we initially discovered the importance to incorporate the knowledge of domain experts in the analytical decision-making process. In BigGIS, the adaptation methodology allowing domain experts to continuously refine and evolve event-driven applications has been developed and instantiated in a flexible event-driven architectural design.

Publications

- Patrick Wiener, Manuel Stein, Daniel Seebacher, Julian Bruns, Matthias Frank, Viliam Simko, Stefan Zander, and Jens Nimis. **BigGIS: A Continuous Refinement Approach to Master Heterogeneity and Uncertainty in Spatio-Temporal Big Data (Vision Paper)**. Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPACIAL '16). 2016, New York, NY, USA. **CCC Blue Sky Ideas Award, Best Vision Paper Runners Up Award**. (see [Wiener et al. 2016]).
- Patrick Wiener, Viliam Simko, and Jens Nimis. **Taming the Evolution of Big Data and Its Technologies in BigGIS - A Conceptual Architectural Framework for Spatio-Temporal Analytics at Scale**. Proceedings of the 3rd International Conference on Geographical Information Systems Theory, Applications and Management (GISTAM). 2017, Porto, Portugal. (see [Wiener et al. 2017]).

LieferBot-E (Automated Supply and Disposal of Urban Neighborhoods through Autonomous Electrified Vehicles, 01/2018 - 06/2021, BMWi)

The major part of this thesis results from the work in the research project LieferBot-E. Hereby, the project investigated the utilization of electrified delivery robot platforms to autonomously pursue package delivery tasks whereby the benefits are twofold: (1) relieve courier, express, and parcel service providers of cost-intensive last-mile delivery and (2) mitigate issues for cities regarding high traffic density, noise and pollution. The goal was to realize a delivery robot monitoring use case while alleviating dispatcher and fleet operator from any technical burdens when deploying and operating respective event-driven monitoring applications in geo-distributed fog infrastructures. Therefore, the challenge was to develop concepts facilitating flexible cloud to edge deployments and realizing geo-distributed event stream management. Hence, individual pipeline elements are either executed in the cloud or can be moved to the delivery robot platform itself. Concepts and methods for supporting citizen technologists in managing geo-distributed event-driven applications are inspired and derived from the key idea of this application scenario. Besides, we conducted evaluations in a real-world setup.

Publications

- Patrick Wiener, Philipp Zehnder, and Dominik Riemer. **Towards Context-Aware and Dynamic Management of Stream Processing Pipelines for Fog Computing**. 2019 IEEE 3rd International Conference on Fog and Edge Computing (ICFEC). 2019, Larnaca, Cyprus. (see [Wiener et al. 2019]).
- Patrick Wiener, Philipp Zehnder, Marco Heyden, Patrick Philipp, and Dominik Riemer. **Fogsy: Towards Holistic Industrial AI Management in Fog and Edge Environments**. KuVS-Fachgespräch Fog Computing (KuVS). 2020, Wien, Essen. (see [Wiener et al. 2020a]).
- Patrick Wiener, Philipp Zehnder, and Dominik Riemer. **Managing Geo-Distributed Stream Processing Pipelines for the IIoT with StreamPipes Edge Extensions**. Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems (DEBS). 2020, New York, NY, USA. (see [Wiener et al. 2020b]).

Moreover, the key ideas and research challenges were thoroughly discussed at the Doctoral Symposium of the 19th International Middleware Conference (Middleware), 2018, Rennes, Britanny, France.

Research and Developer Talks. Parts of the work were also presented in a tutorial at the 19th Annual IEEE/ACM International Symposium in Cluster, Cloud, and Grid Computing (CCGrid), 2019, Larnaca, Cyprus, as well as at international top venues such as Flink Forward, ApacheCon, ApacheCon Asia besides several Meetups around the world. These talks at developer conferences bridge the gap between research and practice and support the adoption of our contributions in industry.

Open-Source Contributions. In addition, the presented contributions are integrated into the open-source project *Apache StreamPipes*[1], an IIoT analytics tool which enables non-technical domain experts to create event-driven applications in a self-service manner. Apache StreamPipes is part of the IoT and IIoT open-source ecosystem within the well-known and renowned Apache Software Foundation, an American nonprofit corporation supporting numerous open-source software projects. This further aids in distributing our contributions on a global scale by providing transparency and accessibility to target interest groups.

1.4 Guide to the Reader

The remainder of this thesis is structured as shown in Figure 1.1. First, we introduce underlying terms, concepts and architectures with regard to event processing, distributed event-based systems and fog computing in **Chapter 2**. Readers familiar with these domains may also skip this chapter to fast-forward. Building on the theoretical foundations, **Chapter 3** motivates our research from an application and user point of view by deriving relevant needs from two IIoT-related scenarios used to formulate a problem statement. **Chapter 4** reviews related work from the field of distributed event-based systems and application management in fog computing with similar problems evolving around geo-distributed event processing. In **Chapter 5**, essential requirements for this thesis are derived by performing a requirements elicitation process. Therefore, the requirements elicitation process is based on the outlined research questions, further complemented by identified needs and problems from the motivation as well as the knowledge obtained from a systematic review of related work. Afterwards, in **Chapter 6** we introduce our generic node model encompassing concepts from the application and infrastructure realms and propose a geo-distributed architecture framing the overall system design. Building on top of the fundamental node model, we state geo-distributed pipeline deployment and operation concepts, propose a strategy for flexible event stream management and introduce the node controller for local run-time management in **Chapter 7**. Thereafter, we first present a generic adaptation methodology alongside relevant key abstractions to support pipeline evolution at run-time prior to suggesting methods for three specific adaptation types in **Chapter 8**. **Chapter 9** presents evaluation results and thorough discussions which is followed by a conclusion and an outlook for future work in **Chapter 10**.

[1] https://streampipes.apache.org/

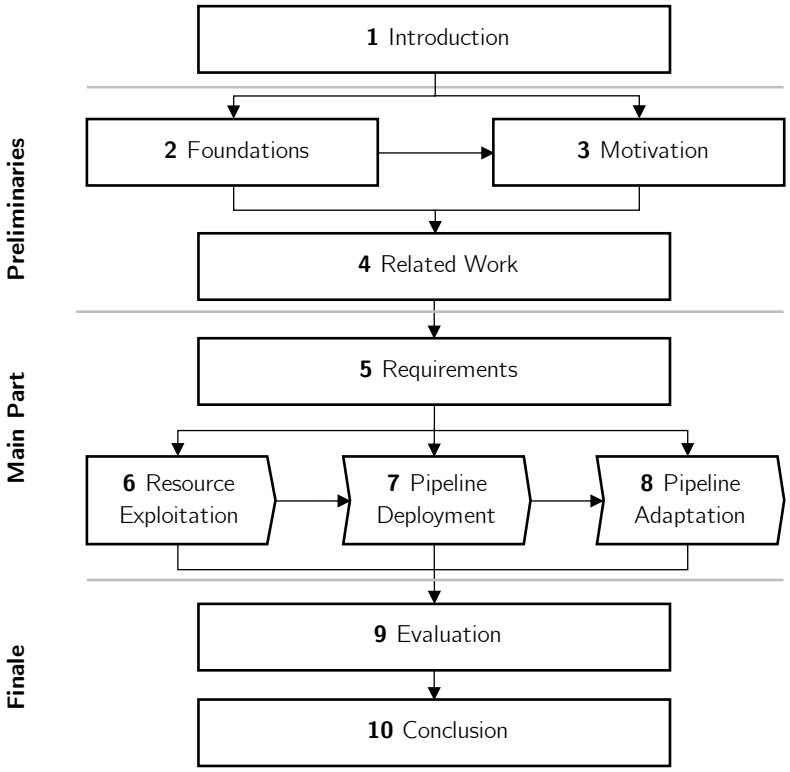


Figure 1.1 Structure of the thesis

Part II

Preliminaries

2 Foundations

In this chapter, we introduce concepts and theoretical foundations needed for the remainder of this thesis. First, we introduce *event processing*, relevant terminologies and a conceptual model for event-driven applications in Section 2.1 before presenting fundamentals of *distributed event-based systems* in Section 2.2. Lastly, we introduce *fog computing* in Section 2.3 and elaborate on related computing models.

2.1 Event Processing

The imminent need to continuously process large amounts of data and to deduce actionable insights under very short delays catalyzed the evolution of models and systems that are capable in dealing with these requirements. Ever since, the field of event processing has gained significant attention in disparate research communities. Unsurprisingly, this has led to the evolution of different terminologies related to event processing in general. In principle, most of them can be categorized into one of the two terms: *data stream processing* [Babcock et al. 2002] and *complex event processing* [Luckham 2002]. Following, we briefly highlight cornerstones and fundamental developments along the historical evolution of both models. Then, we introduce the notion of *events* and *event streams* before presenting *event processing networks*, a conceptual model for event-driven applications.

2.1.1 Background

Data stream processing (DSP) emphasizes the modeling and efficient processing of transient data streams opposed to persistent relations of traditional *database management system* (DBMS). Active DBMS [McCarthy and Dayal 1989] were proposed to extend traditional DBMS in allowing users to specify actions to be automatically triggered for reacting to continuous changes in the data based on event-condition-action rules [Dayal et al. 1988]. More specifically, these systems are classified as closed or open databases depending on whether rules are only applied on changes to internally stored data or also allowed to incorporate external event sources respectively [Cugola and Margara 2012]. In contrast to traditional DBMS that execute one-time queries over point-in-time dataset snapshots, the

idea of continuous queries was proposed and incorporated in the Tapestry system [Terry et al. 1992]. Hereby, queries are deployed once and continually evaluated over the data. *Data stream management systems* (DSMSs) were introduced to deal with data-intensive streaming applications by executing continuous queries on transient and unbounded data streams. Thereby, users are actively provided with updated answers in a timely manner, potentially sacrificing some accuracy if necessary [Babu and Widom 2001; Babcock et al. 2002; Abadi et al. 2003]. This is also referred to as the DBMS-Active, Human-Passive model [Carney et al. 2002]. A DSMS must ensure to produce deterministic and repeatable output which is crucial for fault tolerance and recovery mechanisms [Stonebraker et al. 2005]. However, this is in contrast to the non-deterministic behavior of a DSMS where accuracy is sacrificed by load shedding mechanisms to minimize the degradation in system utility in high load situations [Abadi et al. 2003]. Most popular representatives of DSMSs include STREAM [Babu and Widom 2001], Aurora [Abadi et al. 2003] or TelegraphCQ [Chandrasekaran et al. 2003], commonly considered as the first generation *stream processing engines* (SPEs). The introduction of the MapReduce paradigm [Dean and Ghemawat 2004] marked the beginning of subsequent generations of SPEs for massively-parallel DSP with prominent examples originating both in industry (Storm at Twitter, Samza at LinkedIn) and academia (Spark Streaming at UC Berkley, Flink at TU Berlin).

Parallel to DSMSs, the research area of *complex event processing* (CEP) was introduced as a way to distill meaningful information from simple events into fewer, yet more useful, complex events [Luckham 2002]. The causality between events is modeled by reactive behaviors of involved processing components by the use of event patterns [Luckham and Vera 1995], whereby providers and receivers of information are decoupled [Buchmann and Koldehofe 2009]. Information between producers and consumers are disseminated in the form of events from a large number of distributed sources, e.g., machine sensory in manufacturing. The asynchronous communication and the decoupling form the basis for high interoperability in CEP [Bruns and Dunkel 2010]. Accordingly, CEP allows to continuously process events through aggregation and composition to generate complex events by using event pattern rules that are matched against the incoming events to detect situations of interest [Cugola and Margara 2012].

Nowadays, the separations between CEP and DSP more and more dissolve [Hirzel 2012; Luckham 2020]. Hence, we subsume both models under the more general and abstract term of *event processing* as defined by Etzion and Niblett [Etzion and Niblett 2010]:

> **Definition 1 (Event Processing).** *Event processing* refers to performing operations on events including reading, creating, transforming and deleting events.

The supremacy of event processing over traditional DBMS in use cases such as continuous data analytics has led to a wide adoption in various industries and further provides fundamentals for emerging event-driven applications [Hueske and Kalavri 2019] with recent advancements in the context of the *Industrial Internet of Things* (IIoT).

2.1.2 Events

Events are the most elementary and fundamental entity with regard to event processing. The term *event* was coined by Luckham in the context of CEP and signifies an activity in a system that has happened and can be subjected to computer processing [Luckham 2002]. An event comprises three additional aspects that further describe its characteristics [Luckham 2002]: (1) *form* (attributes contained in an event), (2) *significance* (activity an event signifies), (3) *relativity* (relationship among events, e.g., by time, causality, and aggregration). However, the term system may be misleading in the sense that it is not strictly enforcing to be only limited to computer systems. In fact, any happening of interest leading to a state change in a given domain that can be observed is considered an event [Mühl et al. 2006; Etzion and Niblett 2010]. In that sense, events can occur in the form of physical events such as sensor measurements in a machinery, or as generally arbitrary system inherent events. This is reflected in the definition by Etzion and Niblett [Etzion and Niblett 2010].

> **Definition 2 (Event).** An *event* is an occurrence within a particular system or domain; it is something that has happened or is contemplated as having happened in that domain.

Besides widening the scope for origins of events, it also addresses cases in which events are created without an actual occurrence (i.e., *activity*) having happened—that is, when the observation of the activity led to a false positive [Etzion and Niblett 2010]. Events are considered to be programmatic representations, so-called *event objects* or *event instances* which pertain to a dedicated *event type*, that is, a specification for a group of event objects of the same structure and semantics intent [Etzion and Niblett 2010]. An event type has a set of associated constituent attributes (i.e., *form*) which define the logical event structure [Etzion and Niblett 2010]:

- *Header*—describes meta-information about an event, e.g., its timestamp.
- *Payload*—contains the actual information itself.
- *Open content*—may contain additional information on the event instance.

Finally, there is another term to distinguish that is often used in the context of events. A *message* (or *notification*) is a datum that conveys a serialized form of events [Etzion and Niblett 2010]. Thus, a message can be viewed as a data container on the network level in order to transmit events between endpoints of the underlying communication mechanism, i.e., between event producers and event consumers [Mühl et al. 2006]. Depending on the designated communication mechanism, a message may also contain several events, e.g., when retrieving batches of events from an event log [Etzion and Niblett 2010]. In this regard, distributing events in the system requires for the participating processing entities to agree on a common data format referring to a specific event representation, e.g., *JavaScript Object Notation* (JSON).

2.1.3 Event Streams

The primary abstraction associated with event processing is the notion of an *event stream*. In this matter, data that are to be processed are not available for random access from disk or memory, but arrive in the form of one or more continuous event streams [Babcock et al. 2002]. Thereby, the event stream model varies from the conventional relation model and accommodates a set of characteristics associated with continuous processing, that are (1) events arrive online, (2) the event order cannot be influenced by the system, (3) event streams are potentially unbounded, (4) once an event is processed it is either discarded or archived and cannot be retrieved easily [Babcock et al. 2002]. Especially the last point stresses the fact that event streams often come at high frequency and volume and thus cannot be stored efficiently. Consequently, this poses the real-time requirement on the processing part as it oftentimes can only pass over the data once. Event streams can be classified as *base event streams*, i.e., event streams are external to the application, or *derived event streams*, i.e., intermediary event streams produced internally [Arasu et al. 2006]. Etzion and Niblett define the term as follows [Etzion and Niblett 2010]:

> **Definition 3 (Event Stream).** An *event stream* (or stream) is a set of associated events. It is often a temporally totally ordered set (that is to say, there is a well-defined timestamp-based order to the events in the stream). [...]

Thus, event streams can be understood as a countably infinite sequence of event objects which are continuously generated and made available over time as exemplified in Figure 2.1.The notion of *time* is a central concept in event processing as it allows events to be associated with a certain timestamp (or time step) from a given time domain marking the point when the event was generated, e.g., creation time [Andrade et al. 2014]. In particular, this mapping step (event to time domain) is crucial to evaluate the timeliness of the event as the potential value of the information content decays over time. As an event stream reflects the occurrence of an activity that has happened it commonly adheres an append-only logic [Babu and Widom 2001]. Events are considered immutable and thus are generally no subject of updates or deletions, even though approaches exist that do allow future changes by means of revision flags and reevaluation [Abadi et al. 2005].

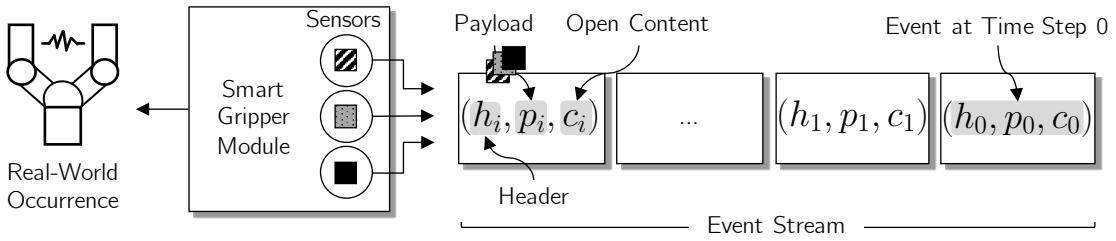


Figure 2.1 Exemplified event creation and resulting event stream

2.1.4 Event Processing Networks

An event-driven application can be logically represented as an event processing network [Perrochon et al. 1999; Luckham 2002; Sharon and Etzion 2007]. The *event processing network* (EPN) is a conceptual model to describe the structure of an event-driven application at an abstract level without the notion of any technical details. Based on the event-driven architecture, an architectural pattern defining principles for the event-driven processing behavior, EPNs aim to express the event-based interactions and processing specifications among components that serve as a basis for realizing event processing architectures in general [Sharon and Etzion 2007]. An EPN can be viewed as an ensemble of platform-independent components: *event producer*, *event processing agent*, *event consumer* and *event channel*. In general, EPN components are considered distributed, loosely coupled while exchanging events asynchronously, mostly based on a push-based event distribution (fire-and-forget). An EPN can also be viewed as a graph as defined in [Sharon and Etzion 2007]:

> **Definition 4 (Event Processing Network).** An *event processing network* is a graph, where vertices are represented by a collection of event processing agents, event producers, event consumers, and edges denote to interconnecting event channels.

Any event object that flows between interconnected components must be transferred through a channel which is a directed link from emitting to receiving components [Sharon and Etzion 2007]. Besides, an EPN may also contain feedback loops where certain outputs of downstream event processing agents are fed back to an upstream event processing agent in the network. While EPNs can form arbitrary topologies, it always describes the flow of events through the network from producer to consumer and specifies intermediate event processing (if any) [Etzion and Niblett 2010].

Next, we briefly introduce each constituent of an EPN as depicted in Figure 2.2 which provides an overview over the graphical notation of an EPN, with arrows indicating the flow of events.

Event Producer. An *event producer* (EP) is an entity located at the edge of an EPN that observes its environment and emits *raw* events into the EPN via channels to be consumed by any party of interest, i.e., either event processing agents or event consumers [Etzion and Niblett 2010]. Thereby, producer do not mutate any state of the system they are observing [Luckham 2002]. Within the EPN, producers represent source nodes, i.e., only directed edges exist that originate from the producer. As the EPN describes an abstract model, producers are only seen as a proxy to an actual event producer typically in the real-world. Therefore, Etzion and Niblett propose three categories of real-world event producer: hardware, software and human interaction [Etzion and Niblett 2010]. The latter refers to events that are generated in the course of a human interaction such as manual acknowledgement of good parts and bad parts in an end of line quality check.

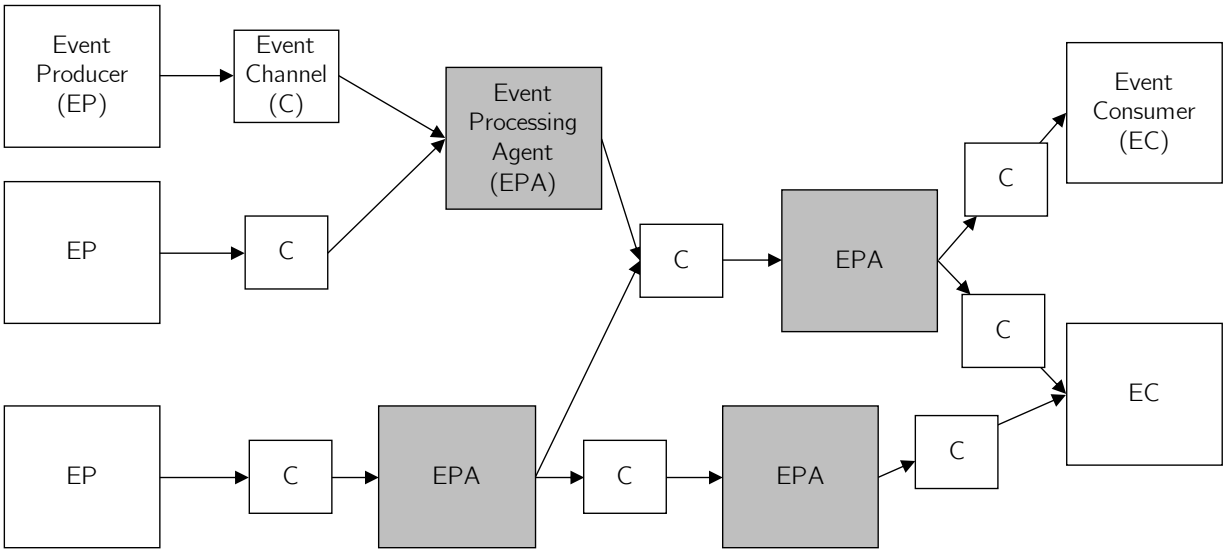


Figure 2.2 Graphical notion of an event processing network

Event Processing Agent. An *event processing agent* (EPA) is responsible to mediate between producers and consumers as a consumers expectations may not always match what is provided [Sharon and Etzion 2007]. For instance, one is generally not interested in raw sensor readings but would like to be informed when a certain threshold value is exceeded. For that matter, EPAs allow to apply intermediary event processing logic "on-the-fly" while events are in transit. Although there are typically different types of EPAs, they all follow the same procedure: first, EPAs receive input events over one or more channels, then they process them, and finally output new derived events to one or more channels. EPAs can be categorized in three base classes [Etzion and Niblett 2010]:

- *Filter*—discard events based on a given criteria
- *Pattern detection*—examine a collection of events for matching patterns
- *Transformation*—modify event properties or structure, e.g., split or aggregate events

Furthermore, a distinction can be made between stateless and stateful EPAs. *Stateless* EPAs (e.g., filtering) process each event entirely independent from the preceding events. On the contrary, *stateful* EPAs (e.g., aggregation) keep "state" between events and thus past events influence the behavior of how current events are processed. Moreover, Etzion and Niblett formulate an EPA hierarchy based on the aforementioned base classes that we explicitly exclude from this work [Etzion and Niblett 2010].

Event Consumer. An *event consumer* (EC) can be viewed as the logical counterpart of a producer and represents sink node that receives events from its preceding components [Etzion and Niblett 2010]. Once a consumer receives events it will perform a certain task based on its predefined internal logic. Similar to producers, consumers can be broadly categorized in any of these three classes: hardware, software, and human

interaction [Etzion and Niblett 2010]. Examples include triggering actions in an actuator, live visualizations of events in dashboards, or sending system-generated notifications over an instant messaging service.

Event Channel. An *event channel* provides a mechanism for distributing events between linked components within the EPN. To this extent, a channel receives events from one or more sources, derives routing decisions, and forwards the input events without changing it to one or more downstream targets in accordance with these routing decision [Etzion and Niblett 2010]. Opposed to simple channels that only mark *one-to-one* mappings between source and target, modeled channels allow sophisticated routing schemes, e.g., *many-to-one*, *one-to-many* or *many-to-many*, including a configurable *quality of service* (QoS) behavior [Chandy and Schulte 2009]. This significantly reduces the overall complexity due to shared channels by decoupling producers, agents and consumers. In this way, any entity in the topology can be dynamically added or removed at run-time without affecting the overall availability of the application. Oftentimes, channels are implemented as a message-oriented middleware to decouple EPN entities [Etzion and Niblett 2010].

2.2 Distributed Event-Based Systems

The dynamics in networked environments, especially in emerging application domains such as IIoT, pose new requirements on the automation of data exchange thus shifting the focus of data and service distribution away from a stationary world to one that is in flux [Mühl et al. 2006]. For many years, distributed event-based systems and their underlying event-based architectural style have been prevalent for realizing event-driven applications at large-scale [Carzaniga et al. 1998]. In the following, we introduce the fundamental concepts and principles of *event-driven architecture* followed by *publish/subscribe*, a core messaging mechanism for asynchronous push-based communication in distributed event-based systems, before presenting *processing pipelines* as a processing topology pattern for distributed event-driven applications.

2.2.1 Event-Driven Architecture

Distributed event-based systems at large-scale imply a high degree of loose coupling and heterogeneity among its components in terms of execution and interoperation. A common architectural style to account for such demands is the *event-driven architecture* (EDA) which is a structure based event observation, generation and notification [Rosenblum and Wolf 1997]. The goal of EDA is to increase efficiency, agility and flexibility of such event-driven system [Bruns and Dunkel 2010]. Accordingly, the event-driven model characterizes the behavior of arbitrary components within an EDA, where their processing

is triggered by the occurrence of events. In this respect, event-driven applications are sometimes viewed as *sense-process-respond* (SPR) applications that react to and generate new events [Schiefer et al. 2007; Chandy and Schulte 2007]. Thus, the event-driven model entails three recurrent steps that characterize the basic principles of EDA [Bruns and Dunkel 2010]:

- *Sense*—events are detected immediately after being created
- *Process*—analyses of detected events, e.g., aggregate, correlate, classify, discard
- *Respond*—reactions to analyses are initiated in a timely manner, i.e., generate derived events or invoke other distributed services in real-time

Consequently, these SPR principles are the foundation for distributed event-based systems that are embedded in the underlying architectural design as well as all constituent components. An EDA possesses key characteristics with regard to its *communication model* and *processing model* [Bruns and Dunkel 2010].

In contrast to conventional request/response (also request/reply) interactions that precipitate tight coupling of communicating parties and thus impair scalability [Franklin and Zdonik 1998], the event-based architectural style facilitates a clear separation of communication and computation by inherently decoupling participating components. This is achieved by using a mediating middleware which conveys events from event producing entities to event consuming entities based on expressed consumer interests without them having any prior knowledge about each other [Carzaniga et al. 1998; Eugster et al. 2003]. As a result, this allows to not only scale-out across multiple nodes but also to dynamically add or remove individual components at run-time without others being affected which leads to high compositionality and reconfigurability of the EDA [Carzaniga et al. 1998]. Further, decoupling individual components reduces dependencies to a minimum, namely to the coordination of the syntactic structure and the semantics of the conveyed events [Bruns and Dunkel 2010]. A key characteristic of this event-based communication model is that it follows a push-based opposed to a pull-based approach as in request/response to provide asynchronous, non-blocking event transfer oftentimes realized by a middleware that implements the publish/subscribe pattern, further detailed in Section 2.2.2. In this regard, the event-based communication model acts inverse to request/response as the initiator of communication, namely the event producing component, actively provides the data [Mühl et al. 2006]. Consuming components immediately react to received events by processing them according to their inherent application logic. As a result, the event-based communication model mitigates elementary disadvantages of request/response in dynamic networked systems [Mühl et al. 2006] that include potential system congestion due to unnecessary requests from short polling intervals as well as increased update latencies and potentially stale data due to long polling intervals which makes it superior in the context of data dissemination [Franklin and Zdonik 1997; Fiege et al. 2002]. Using events as a uniform primitive for information exchange simplifies the construction of complex systems composed of several autonomously operating, distributed components which aids in evolving event-driven applications to cope with new

requirements [Sullivan and Notkin 1990; Bates et al. 1998]. It is noteworthy, that not every software system that deals with events automatically meets the criteria for an EDA. As previously mentioned, events may also occur in response to requests or remote procedure calls. Furthermore, the EPN and its dedicated EPAs (see Section 2.1.4) are considered an integral part of the processing model of a sophisticated EDA [Bruns and Dunkel 2010].

EDA is oftentimes complemented with a microservice architectural style that views components as self-contained, independently deployable and scalable services having a bounded domain context and thus facilitate a clear separation of concerns. Similarly to EDA, event-driven applications built from microservices aim to be as decoupled and as cohesive as possible while exposing interfaces that support lightweight messaging for producing and consuming events [Lewis and Fowler 2014]. Combining both architectural styles aids in obtaining non-functional requirements such as performance, scalability, and availability and allows to realize complex distributed and extensible event-driven applications which are capable of consuming, processing, aggregating, or correlating large amounts of events in real-time. Moreover, advancements in the field of container technology, a lightweight virtualization paradigm at the operating system level, have evolved into the de facto standard for deploying and operating distributed event-driven applications with wide adoption in both academia and industry [Alshuqayran et al. 2016; Kang et al. 2016; Pahl et al. 2020].

2.2.2 Publish/Subscribe

Publish/subscribe (or pub/sub) is a messaging mechanism for push-based communication and a common approach to realize event channels of the mediating middleware in EDAs. In contrast to synchronous point-to-point communication that leads to rigid and static application structures, publish/subscribe is an interaction scheme that provides loose coupling between participating parties, that are *publisher* (also referred to as *producer*) and *subscriber* (also referred to as *consumer*) which are decoupled by a mediating *event service* (also referred to as *broker*). In general, subscribers issue their interest in certain events in the form of a subscription at the broker and are subsequently notified in case any publishers published matching events [Eugster et al. 2003]. The actual event dissemination happens asynchronously without the publishers and the subscribers knowing about each other which leads to a full decoupling in various dimensions [Eugster et al. 2003]:

- *Space*—publisher and subscriber are unaware of each other
- *Time*—publisher and subscriber do not need to simultaneously participate
- *Synchronization*—publisher and subscriber operate in a non-blocking way

Depending on the supported expressivity of subscriptions, publish/subscribe models can be broadly classified as *topic-based* and *content-based* [Eugster et al. 2003].

Topic-based Publish/Subscribe. Inspired by early work in the context of group communication in distributed systems [Birman and Joseph 1987; Birman 1993; Powell 1996], the *topic-based publish/subscribe* scheme (also called *subject-based*) was introduced and allows publishers to administer messages in named logical channels commonly referred to as *topics* (or *subjects*) [Oki et al. 1993]. Thereby, publishers usually annotate messages with a dedicated topic that subscribers can use in order to issue a subscription. In practice, topic-based publish/subscribe maps individual topics to distinct communication channels. Though this approach can be implemented very efficiently, one of the major drawbacks is the usage of static topic schemes that are governed by publishers resulting in a lack of sufficient expressivity with respect to subscribers. To alleviate this shortcoming, additions have been proposed such as permitting topic hierarchies, that is, topics can be organized by specifying containment relationships along the possibility for subscribers to articulate simple expressions based on keyword matchings or wildcards [TIBCO 1999].

Content-based Publish/Subscribe. The *content-based publish/subscribe* scheme mitigates the downsides of the topic-based approach by allowing subscriptions based on actual event properties conveyed in the message rather than a predefined criterion [Rosenblum and Wolf 1997]. In this matter, subscribers specify subscription predicates using constraints to express the interest in only matching events [Mühl 2001]. These predicates can also be logically combined to form complex subscription patterns. As predicates are evaluated on the event content instead of static topic schemes, sophisticated protocols are required that incur higher overhead at run-time [Eugster et al. 2003].

2.2.3 Processing Pipelines

In distributed event-based systems, we often face the notion of *processing pipelines*. A processing pipeline (or *pipeline*) refers to a specific processing topology pattern which describes the structure, organization and interaction of all entities that provide real-time event processing capabilities [Bruns and Dunkel 2010]. The idea of patterns was introduced in the field of building architecture [Alexander et al. 1977; Alexander 1979] and since then has been transferred to several other disciplines, most notably software engineering where Gamma et al. coined the term design patterns [Gamma et al. 1993; Gamma et al. 1994]. In general, patterns serve as a blueprint for recurring design problems in the development of a software system and differ in terms of their granularity thereby ranging from architectural patterns at the highest level of abstraction, over design patterns to idioms at lowest level [Buschmann et al. 1996]. A processing pipeline is an architectural pattern specifying the logical view [Kruchten 1995] of event-driven applications. Event-based systems are usually structured hierarchically, i.e., the most abstract views refeis to the general application logic, which in turn is composed or orchestrated of multiple smaller and independent processing services [Bruns and Dunkel 2010]. Consequently, events are processed in stages as a sequence of multiple interconnected processing

services according to the pipes-and-filters pattern [Mcllroy 1964; Buschmann et al. 1996] which views the set of filters and their connecting pipes as a pipeline. Processing pipelines have characteristics that resemble the ones of EPNs (see Section 2.1.4). Events originating from upstream event sources are forwarded via interconnected stages until they reach downstream application systems for event handling. In between, each processing service represents an event processing agent which operates on incoming events with the goal to derive meaningful insights after multiple stages according to the overall application objective.

In principle, processing pipelines can be realized in various programming styles, however generally structuring the application program as a directed acyclic graph according to the dataflow programming paradigm [Dennis and Misunas 1974], whereby we differentiate between code-based *imperative* and *declarative* as well as graphical *flow-based* programming paradigms as contrasted in Table 2.1.

Imperative and Declarative Programming. Both imperative and declarative programming models are commonly found in modern stream processing engines, e.g., Flink, Spark. These stream processing engines typically provide higher-level APIs as an abstraction on top of a set of core operators which are executed inside the streaming runtime. Oftentimes, these are complemented by domain-specific libraries for CEP, graph processing or machine learning. Internally, data exchange between subsequent operators is done via non-blocking I/O event channels. While imperative programming models allow for more expressiveness and fine-grained topology configuration, declarative programming models based on SQL-like interfaces for relational processing using declarative queries aim at a wider adoption [Armbrust et al. 2015; Begoli et al. 2019]. However, both programming models are aimed at experienced technical users.

Flow-Based Programming. Conversely to imperative and declarative paradigms, flow-based programming models [Morrison 1994] further introduce another abstraction layer. In doing so, this allows to visually describe the logical view of the pipeline topology by means of a *graphical user interface* (GUI) which is receiving enormous attention due to advancements in both academia and industry. We further differentiate between two categories of flow-based programming models that vary fundamentally in terms of their execution environment, namely *technology-specific* [Giang 2019; Mahapatra 2019] and *technology-agnostic* [Riemer 2016]. In technology-specific environments, the graphical user interface is added as an abstraction layer on top of existing APIs to interact with stream processing engines that may involve validation and automated code generation [Mahapatra and Prehofer 2019]. In technology-agnostic environments, the graphical user interface is added as an abstraction layer on top of an EDA where a set of loosely-coupled, standalone event-driven processing services in the sense of EPNs components are operated in combination with a publish/subscribe broker acting as an event channel. In the latter case, the actual EPN or pipeline management, namely the orchestration, is handled by

Attribute	Imperative & Declarative	Flow-based
Interface	code-based (API)	graphical (GUI)
Abstraction	low, moderate	high
EPN-Technology	specific	specific, agnostic
EPN-Management	internal	external
EPA-Runtime	low-level operator	event-driven processing service
Event Channel	non-blocking I/O channel	publish/subscribe broker
Target User	technical	non-technical

Table 2.1 Programming paradigms to realize processing pipelines

an external service. In general, the graphical flow-based paradigm abstracts complexities of underlying runtime environment and management from end users and offers an intuitive entry point to create event-driven applications in a drag and drop fashion which makes it suitable for non-technical users. Hereafter, we focus on processing pipelines that are created using the graphical flow-based programming paradigm with emphasize on technology-agnostic event-driven processing services leveraging a publish/subscribe message broker for event dissemination throughout the EPN.

Flow-based Pipeline Authoring. Based on EPN principles and manifested in an EDA, methodologies for graphical flow-based modeling and authoring of processing pipelines have been proposed that use semantic web services to facilitate the development of distributed event-driven applications [Riemer et al. 2014; Riemer 2016]. Here, an event-driven application is comprised of arbitrary potentially heterogeneous event-driven processing services also subsumed under the term *pipeline elements* for different EPN constituents (EP, EPA, EC). Thus, a processing pipeline can be defined as follows [Riemer et al. 2014]:

> **Definition 5 (Processing Pipeline).** A *processing pipeline* is a composition of arbitrary, potentially distributed, heterogeneous event-driven processing services (or pipeline elements).

Thereby, individual pipeline elements provide a *description graph* that contains a complete specification of the corresponding event-driven processing service, namely information on run-time event streams produced by an EP or requirements and output strategies of an EPA and EC. Furthermore, pipeline elements are considered either *active* or *passive* [Riemer 2016]. For instance, EP pipeline elements are classified as passive as they only provide information on their published event stream. These include descriptions of corresponding event schema, stream quality properties such as frequency and corresponding stream grounding. The stream grounding denotes technical aspects regarding the event representation like JSON, or Thrift[1], as well as the run-time transport protocol to realize

[1] https://thrift.apache.org/

the event channel. Therefore, the transport protocol commonly utilizes a topic-based publish/subscribe messaging mechanism like the *Message Queuing Telemetry Transport* (MQTT) protocol. In contrast, EPA and EC are active pipeline elements. Active pipeline elements are instantiated at run-time using binding information such as static data to configure the inherent functional logic and information about the input and output event streams which are part of an *invocation graph* that is sent to the dedicated processing service when being invoked, i.e., instantiated [Riemer 2016]. Both the declaration of pipeline element specifications in the setup phase and the instantiation via invocation in the execution phase are part of the overall interaction model with a central pipeline management that orchestrates the processing pipelines using a request/response messaging pattern. However, run-time operations of individual pipeline elements are completely detached from the modeling phase. Consequently, logically connected pipeline elements are only loosely-coupled, using a topic-based publish/subscribe middleware between intermittent processing stages.

2.3 Decentralized Computing

In recent years, the rapid increase in digitalization initiatives such as Industry 4.0 in industrial domains have catered the emergence of novel application scenarios evolving around the IIoT which come along new challenges and domain-specific needs. Thus ever since, these circumstances have urged the development of new concepts, system architectures and compute paradigms for decentralization to ensure overall QoS. In the following, we touch upon key milestones along the historical evolution of decentralized computing and introduce *fog computing*. Next, we elaborate on its characteristics to further sharpen the terminology and distinguish it from related computing paradigms, before presenting a conceptual view on the fog computing architecture.

2.3.1 Background

The beginnings of decentralized computing in general can be traced back to the introduction of content delivery networks [Dilley et al. 2002]. A *content delivery network* (CDN) uses nodes at locations geographically closer to end users to provide cached content such as images and videos targeted at saving network bandwidth to reduce service bottlenecks. Noble et al. showed how different types of applications, namely web browsers, video streaming and speech recognition, could run on resource-constrained mobile devices with acceptable performance by offloading dedicated compute intensive tasks to nearby servers in order to relieve the overall system load [Noble et al. 1997] which was further extended to improve battery life [Flinn and Satyanarayanan 1999]. Satyanarayanan generalized these concepts and introduced *cyber foraging* [Satyanarayanan 2001], which was further refined in [Balan et al. 2002] and seen as one of the early works for edge

computing [Mouradian et al. 2018]. In cyber foraging, resource-limited mobile devices temporarily exploit proximate servers, so-called *surrogates*, that are connected to the Internet through high-bandwidth networks. A major cornerstone along the way was the introduction of *cloud computing* that received particular attention when Amazon initially promoted its virtual computing environment, the Amazon Elastic Compute Cloud, or Amazon EC2 [Barr 2006]. Cloud computing opened up a plethora of new opportunities including *Infrastructure-as-a-Service* (IaaS) offerings with on-demand access to virtualized, scalable resources in terms of compute, storage and network [Baun et al. 2011]. However, consolidating data at a central location implies larger average latencies in application scenarios that involve mobile devices. Satyanarayanan et al. were among the first to observe the need of emerging applications that require low latency and thus local data processing to enable timely decision-making. As a result, Satyanarayanan et al. propose a two-tier architecture that includes the cloud on the first tier as well as *cloudlets* on the second tier [Satyanarayanan et al. 2009]. Cloudlets or "data center in a box" are Internet infrastructure components that are decentralized and geographically widely dispersed computational resources that are ideally self-managing and offer proximate compute capability in a single hop communication link to nearby mobile devices [Satyanarayanan et al. 2009]. Besides, cloudlets only store soft states that are cached copies of data or code while hard state exclusively remains in the cloud which makes cloudlets more resilient to failures [Satyanarayanan et al. 2009].

2.3.2 Fog Computing

Fog computing was first introduced by Cisco as a concept that extends the existing cloud computing paradigm to the edge of the network [Bonomi et al. 2012]. According to Bonomi et al., fog computing is a highly virtualized platform that provides computation, storage, and networking services between end devices and cloud servers that are typically, but not exclusively located at the edge of the network [Bonomi et al. 2012]. Thereby, fog computing aims to overcome precedented shortcomings of cloud-centric approaches by moving substantial amounts of computational resources in closer vicinity of the spatially distributed event sources, analogous to the "code-to-data" principle. Additionally, the proposed paradigm still embraces benefits of the existing cloud model in terms of powerful and elastically scalable compute resources. Thus, fog computing serves as an extension and expansion of cloud computing, rather than a substitute [Hu et al. 2017]. While fog computing naturally provides lower latencies and allows for real-time processing of sensitive data, the remaining distilled data are transferred to the cloud and used for running complex compute-expensive big data analytics, training machine learning models, long-term storage, or general application management [Hong et al. 2013; Bittencourt et al. 2015; Hu et al. 2017]. Typical domains include industrial automation, transportation, and networks of sensors and actuators [Stojmenovic and Wen 2014].

As of today, there still exists no common consensus on the term fog computing itself, neither by researchers nor practitioners. There are numerous other computing paradigms that coincide with fog computing such as the previously mentioned cyber foraging and cloudlets as well as more recent compute paradigms, most notably *edge computing* [Shi et al. 2016] and *mist computing* [Preden et al. 2015], Yet, this further hinders the convergence towards a generally agreed-on definition. While cloudlet, edge and mist computing paradigms are more geared towards the things side, fog computing focus more on the infrastructure side in order to better address the necessities in the advent of emerging IoT scenarios [Shi et al. 2016; Satyanarayanan 2017]. As these fields of research are still in their infancy the boundaries between the partially overlapping paradigms are oftentimes fluid. An essential commonality between fog computing and its related paradigms, above all, edge computing, is the attempt to gather, process and analyze data from real-world event sources, e.g., industrial machines in the case of manufacturing, more efficiently compared to the existing cloud computing model by pushing intelligence and processing capability down to where the data originates. Despite this mutual objective there are vital differences with regard to where exactly the actual intelligence is placed [Mahmood and Ramachandran 2018]. In view of this, a commonly used term refers to the *edge* of the network as opposed to the core network which we will discuss in Section 2.3.3 in more detail.

Hence, it is hardly surprising that fog computing is either considered synonymous with other decentralized computing paradigms such as edge computing [Shi et al. 2016; Hao et al. 2017; Mahmood and Ramachandran 2018] or is defined as a combination of edge, cloud and any computational resources in between, so-called *fog nodes* [Hong et al. 2013; Shi and Dustdar 2016; Bermbach et al. 2018]. In this work, we take the latter perspective and refer to the definition of fog computing provided by the National Institute of Standards and Technology [Iorga et al. 2018]:

> **Definition 6 (Fog Computing).** *Fog computing* is a layered model for enabling ubiquitous access to a shared continuum of scalable computing resources. The model facilitates the deployment of distributed, latency-aware applications and services, and consists of fog nodes (physical or virtual), residing between smart end devices and centralized (cloud) services.

Fog computing can be distinguished from other related computing paradigms by the following intrinsic characteristics: (1) contextual location awareness and low latency, (2) geographical distribution, (3) heterogeneity, (4) interoperability and federation, (5) real-time interactions, and (6) scalability and agility of federated fog cluster. These characteristics of fog computing are oftentimes complemented by two additional associated characteristics: (7) predominance of wireless access and (8) support for mobility [Iorga et al. 2018; Bonomi et al. 2012]. In the following, we elaborate on each of these characteristics in detail.

- *Contextual location awareness and low latency*—Location awareness is a relevant factor in fog computing as most applications tend to be location dependent. On the one hand, location awareness aids to attain low latency for mission-critical applications [Iorga et al. 2018; Bonomi et al. 2012]. On the other hand, it allows to account for data ownership and locality needs in order to keep sensitive data in given physical or logical boundaries to preserve privacy [Vaquero and Rodero-Merino 2014]. Therefore, fog nodes must be aware of their logical location in the network. This can be achieved either by static, a priori configuration of the location at setup time or autonomously when joining the cluster by determining their location relative to nearby nodes with known locations [Tammemäe et al. 2018].
- *Geographical distribution*—In contrast to the centralized cloud, fog computing targets applications that require geographically distributed deployments [Bonomi et al. 2012]. Fog computing infrastructures are comprised of widely dispersed computational resources that allow to run processing and analytics tasks where they are best suited anywhere along the cloud-edge continuum to improve the overall QoS of dedicated application scenario. This plays a vital role not only in stationary deployments in the context of the IIoT and smart manufacturing [Chen et al. 2018; Wiener et al. 2020a; Pop et al. 2021], but also in dynamic and mobile deployments such as connected vehicle applications [Hong et al. 2013; Hou et al. 2016].
- *Heterogeneity*—An essential characteristic of fog computing is that computational resources are heterogeneous in nature. Opposed to cloud computing environments that mostly consist of homogenous, powerful resources, the fog computing architecture is typically comprised of a multitude of heterogeneous, volatile and constrained physical resources that collect and process data acquired through different kinds of network communication technologies [Iorga et al. 2018; Chiang et al. 2017]. As indicated by Bonomi et al., fog nodes occur in a variety of form factors and thus offer vastly varying hardware resources in terms of memory, secondary storage, processor types and architectures. Additionally, fog nodes are either virtualized or physical nodes that are deployed in a variety of environments [Bonomi et al. 2012; Bonomi et al. 2014]. Moreover, these platforms run various types of operating systems as well dedicated software packages that lead to an increasingly large set of different hardware and software capabilities [Bonomi et al. 2014]. With the increasing adoption and affordable access to hardware-accelerated compute units tailored for running *artificial intelligence* (AI) applications, heterogeneity in fog computing environments is further exacerbated. Heterogeneity is considered a crucial aspect for any type of systems operating in fog computing environments and needs to be dealt with in the application management middleware, in particular, regarding the deployment of individual application components [Mouradian et al. 2018].
- *Interoperability and federation*—The seamless support of certain services requires the cooperation between different fog service providers. Hence, fog nodes have to be able to interoperate, and services must be federated across different application areas [Bonomi et al. 2012]. Unlike cloud environments that leverage resource

pooling inside large data centers to even out variability in demand, fog and edge environments are strictly limited available computational resources [Bermbach et al. 2018]. As a consequence, new forms of competition and cooperation among different fog service providers will arise [Kai et al. 2016].

- *Real-time interactions*—Unlike other computing paradigms such as cloud computing, real-time interactions are one of the key characteristics of fog computing. Due to the nature of real-world deployments where event sources permanently observe their surroundings and continuously produce streams of events that come at varying frequencies and volumes, applications scenarios are generally concerned with timely decision-making as the conveyed information may decay over time. Consequently, real-time event processing is preferred without any interruptions [Bonomi et al. 2012; Madakam and Bhagat 2018].
- *Scalability and agility of federated fog clusters*— While individual fog nodes are resource-bound, at its core, fog computing allows to adaptively scale when considering clusters of fog nodes or even clusters of clusters. To this extent, these clusters support elastic computing, resource pooling, data-load changes, and volatile network conditions [Iorga et al. 2018]. Fog computing infrastructures can grow with minimal disruption in order to adequately support arising applicational demands which allows organizations to start modestly and seamlessly scale the initial infrastructure to large-scale deployments [Byers 2017]. Additionally, fog computing allows for cooperative offloading and load sharing [Shin and Chang 1989] among neighboring fog nodes to reduce processing delays and thus prevent overloading potentially contented node resources [Yousefpour et al. 2018].
- *Predominance of wireless access*—Although fog computing is used in wired environments such as in stationary industrial settings on the factory shop floor, the large scale of wireless sensors as well as mobile application scenarios demand for distributed computing power in close vicinity [Iorga et al. 2018]. Similar to computational resources, the network infrastructure of fog computing is also heterogeneous ranging from high-speed links in cloud data centers to wireless access technologies such as ZigBee, 3G, WiFi or 5G connecting edge devices [Bonomi et al. 2012; Stojmenovic and Wen 2014].
- *Support for mobility*—As many fog applications require to directly interact with mobile devices, it is mandatory for fog computing to support mobility techniques and protocols [Bonomi et al. 2012]. For instance, in connected vehicles scenarios, moving vehicles are viewed as computational resources that form powerful cluster once they connect with each other [Hou et al. 2016]. Fog computing well supports mobile scenarios of moving nodes and their interactions to nearby stationary resources by means of geo-distributed data replication [Mayer et al. 2017] as well as offloading and migration schemes [Saurez et al. 2016; Puliafito et al. 2018]. The mobility aspect of fog computing is a key characteristic that distinguishes fog computing from edge and cloud computing [Mahmood and Ramachandran 2018].

The described characteristics for fog computing not only aid to sharpen the understanding of the term itself, but allows to contrast it to other computing paradigms in this context. It is worthy to note, that not all general characteristics may be equally present in potential application scenarios leveraging fog computing as an enabling technology.

2.3.3 Fog Computing Architecture

Fog computing extends the traditional cloud computing model by providing computational resources in closer proximity to geographically dispersed end devices, namely *IoT devices* on the ground. Thereby, the decentralized fog computing model differs from related conventional computing models in terms of their proposed architecture [Hu et al. 2017]. In recent year, many different architectures for fog computing have been proposed [Bonomi et al. 2012; Masip-Bruin et al. 2016; Liang et al. 2017; Hong and Varghese 2019; Karagiannis and Schulte 2020]. Most of these architectures are derived from the fundamental structure which views fog computing systems to be organized in multiple, hierarchical resource layers. The hierarchical fog computing architecture generally resembles three logical layers that comprise an *edge layer* at the bottom, closest to the IoT devices, an intermediary *fog layer*, and a *cloud layer* at the top of the hierarchy as depicted in Figure 2.3. Accordingly, fog computing envisions a seamless resource management across the hierarchical architecture that allows to place computing anywhere along the cloud-edge continuum [Bonomi et al. 2014; Chiang et al. 2017].

In the following, we elaborate on the individual layers constituting to fog computing in more detail starting by clarifying the notion of *edge* of the network.

Where is the "edge" of the network? The edge of the network denotes to the area where an IoT device or the local area network interfaces with the core of the network (or Internet). In turn, nearby computational resources or services are provided to those IoT devices to either facilitate enhanced operation or enable to realize data-driven use cases that involve low-latency analytics. While in the telecommunication industry the term edge describes the edge of the mobile network, within the radio access network close to mobile subscribers [Hu et al. 2015], other work in the area of IoT assume the edge to be within the local area network, i.e., on the factory shop floor, where physical assets and field devices such as sensors, actuators, motors are located [Garcia Lopez et al. 2015; Chen et al. 2018]. According to Yousefpour et al., the edge of the network is considered the immediate first hop from real-world event sources where intelligence is placed [Yousefpour et al. 2019]. Hereafter, we refer to the network edge in a more relaxed definition being the resource layer closest to the event source represented by real-world end devices.

The *edge layer* is the layer closest to real-world IoT devices and thus the physical environment. IoT devices are typically small, resource-limited and oftentimes purpose-built for

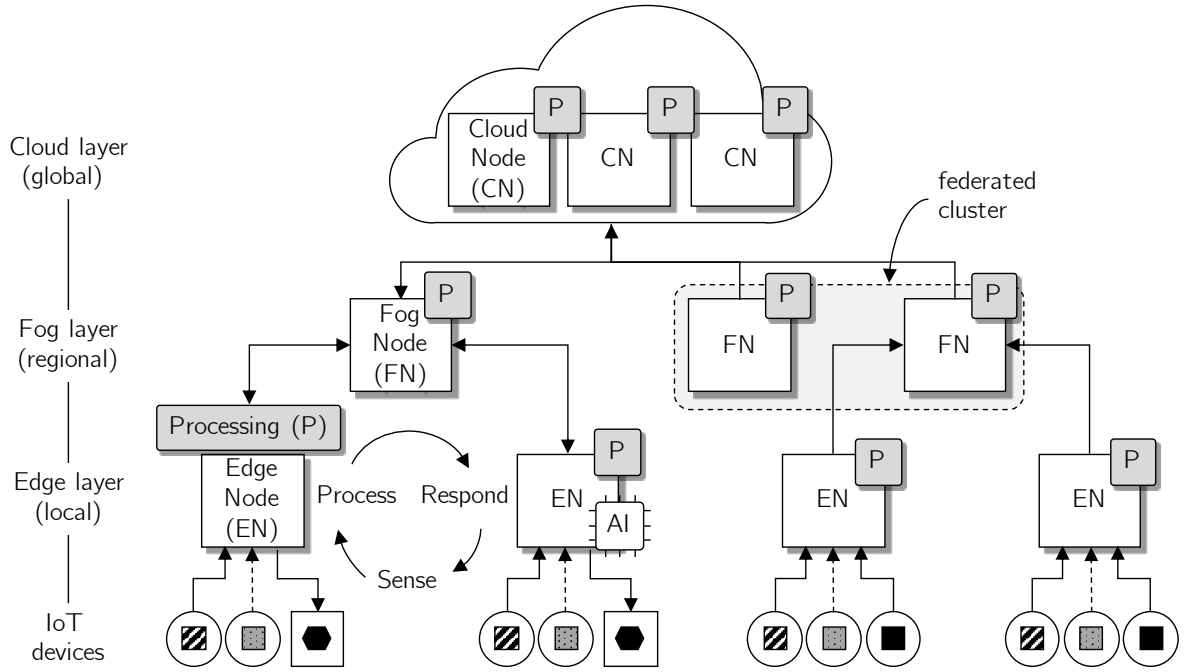


Figure 2.3 Hierarchical fog computing architecture comprised of edge, fog, and cloud layer

a given task such as single sensors and actuators, embedded systems, programmable logic controller, or machines. These devices may be widely geographically distributed and are generally responsible for observing their physical surroundings, sensing relevant features of real-world objects, and transmitting these events to the nearby edge layer for processing or storage [Hu et al. 2017]. Computational resources in this layer are provided by so-called *edge nodes*. Edge nodes are capable to immediately process received events in real-time and autonomously conduct decision-making as they are much closer to the *Internet of Things* (IoT) devices compared to fog nodes [Shi et al. 2016; Varghese and Buyya 2018]. In contrast to cloud data centers, spatial mobility of IoT devices as well as edge nodes is frequent, though not the default [Stojmenovic and Wen 2014]. For instance, while autonomous delivery robots are inherently mobile, connected industrial machines remain stationary [Wiener et al. 2019]. If required, edge nodes may respond their decision to corresponding IoT devices at minimal delay, e.g., to trigger a certain robot operation in case of an automated quality inspection, to close the sense-process-respond loop as discussed in Section 2.2.1. Typical edge nodes range from base stations and routers, to general-purpose single-board computer and industrial PC, to specialized computer systems containing miniaturized AI accelerators. Especially the latter has gained increasing popularity in recent years. With AI applications on the uprise, deployments on such edge nodes not only facilitate more efficient and effective model inferencing, but are also enablers for promising techniques such as federated learning, active learning or transfer learning at the edge, which is subsumed under the umbrella term *edge AI* or *edge intelligence* [Zhou et al. 2019; Rausch and Dustdar 2019; Greengard 2020; Tziouvaras

and Foukalas 2020]. This underpins the fact that computational resources in fog computing architectures are most likely heterogeneous platforms. Similar to IoT devices, edge nodes are exposed to the physical world making them error prone, unreliable in terms of network connectivity and potentially accessible by third-parties, e.g., when installed in public areas.

The *fog layer* introduces an additional intermediary resource layer by leveraging *fog nodes* that are computational resources residing between the edge and the cloud in the network topology [Bonomi et al. 2012]. Fog nodes can either be *physical* nodes, e.g., gateways, switches, servers, or *virtual* nodes, e.g., virtualized switches, virtual machines, cloudlets [Iorga et al. 2018]. In addition, fog nodes provide lower access latencies in both directions (edge to fog and vice versa) and serve edge nodes as an additional nearby resource pool apart from higher and more reliable bandwidth opposed to the far distant cloud data centers [Varshney and Simmhan 2017; Satyanarayanan et al. 2009]. That said, on the one hand the fog layer is seen analogous to a CDN, however located closer to the edge [Vaquero and Rodero-Merino 2014]. On the other hand, it functions as a reversed CDN [Satyanarayanan et al. 2015] allowing data originating from IoT devices to be staged in the fog layer and periodically pushed to the cloud for long-term storage after some additional processing [Bonomi et al. 2012; Dastjerdi et al. 2016]. Therefore, fog nodes are connected with the upper cloud layer via high bandwidth Internet connections to obtain more powerful computing and storage capabilities [Hu et al. 2017]. Still, similarly to edge nodes, fog nodes are heterogeneous by definition with varying resource characteristics. Likewise, the fog layer also manifests fog nodes that are either stationary at a fixed location, e.g., on the factory shop floor, or mobile when installed on moving vehicles or platforms [Luan et al. 2016]. Fog nodes may form clusters in order to pool computational resources to be obtained by edge nodes or end devices when needed [Bonomi et al. 2012]. Clusters can either exist vertically to support isolation, horizontally to support federation, or relative in terms of their latency distance to end devices respectively edge nodes. Moreover, fog nodes are aware of their geographical distribution and logical location within the fog node cluster and provide data management and communication services between the edge layer and the fog computing service or centralized cloud computing resources, if required [Iorga et al. 2018].

The *cloud layer* is at the top of the hierarchical fog computing architecture and consists of mostly homogeneous, powerful computational resources referred to as *cloud nodes* that are centrally located inside operated data centers. Depending on whether cloud refers to either public or private cloud, these resources are available to anyone in an on-demand fashion, or exclusively accessible to private enterprises [Varshney and Simmhan 2017]. To this extent, the cloud layer has the widest geographical coverage, but, at the same time the highest latencies to the end devices at the edge of network [Bonomi et al. 2012]. Nevertheless, the cloud layer offers the illusion of infinite computing and storage capabilities usually in the form of virtualized and managed environments and is commonly used for compute-intensive big data analytics with low delay requirements to

extract meaningful knowledge from preprocessed and aggregated data originating from the lower fog and edge layer besides long-term storage [Hu et al. 2017; Bermbach et al. 2018]. In addition, the cloud layer serves as the central entry point for human-interaction that include visualization and reporting, human-machine interactions as well as general application and cluster management [Bonomi et al. 2012; Bittencourt et al. 2015].

From an application point of view, similar to the physical mobility of edge and fog nodes, logical mobility of application components deployed in the inherently distributed fog computing architecture is common. In this regard, lightweight operating system-level virtualization in terms of *containers* proofs to be a feasible approach to best fit the heterogeneous, constrained and scarce resource pool. Containers are seen an appropriate execution environment providing portability and isolation for individual application components [Pahl and Lee 2015; Ismail et al. 2015; Bellavista and Zanni 2017; Dupont et al. 2017]. Among a variety of application models in the field of fog computing, processing pipelines are widely used for formulating complex distributed event-driven applications due to their real-time processing ability [Shi et al. 2012; Giang et al. 2015; Brogi et al. 2018; Karamoozian et al. 2019]. The application management of such event-driven applications involves orchestrating and coordinating both application components as well as data and control flows. Such an application management can either be centralized or a distributed [Mahmud et al. 2020]. In the centralized case, a single management service has a global view of the fog environment and performs necessary orchestration decisions, therefore, typically located at an accessible location, e.g., the cloud layer. In contrast, the distributed case utilizes several management services spread across nodes on the lower edge or fog layer that generally operate independently and act according to their limited local view of the fog environment, including distributed forms of coordination such as peer-to-peer [Vaquero and Rodero-Merino 2014; Varshney and Simmhan 2017].

Finally, nor the fog and edge layer or the cloud layer are perceived as a mandatory layer as different scenarios might pose different needs on the architecture according to the actual requirements of IoT devices or applications [Iorga et al. 2018; Sunyaev 2020].

3 Motivation

In this chapter, we motivate our research by introducing two scenarios in the context of the Industrial Internet of Things in Section 3.1. We derive related applicational and organizational needs to facilitate the democratization movement towards managing event-driven applications in fog computing in Section 3.2. In Section 3.3, we highlight limitations of current systems alongside problem statements inline with the main research questions of this thesis before concluding in Section 3.4.

3.1 Democratizing Application Management in Fog Computing

We are in the midst of witnessing profound shifts across many industries caused by the confluence of emerging technological breakthroughs in a wide variety of fields such as artificial intelligence, robotics, autonomous vehicles, and the IoT [Schwab 2016]. In particular the proliferation of the IoT and its industrial adoption within the IIoT [Jeschke et al. 2017] has led to a deluge of produced data due to the ubiquitous presence and the progression in the installation of new IoT devices and connected industrial equipment in a myriad of applications. Respective IIoT application domains among others include *Factory 4.0*, which focuses on improvements in product and process quality of industrial assets on the factory shop floor or *Smart Urban Logistics* which targets innovative package delivery concepts leveraging autonomous robot delivery platforms. This offers companies new possibilities for digital transformation and establishing data-driven solutions aside from the development of novel digital products and business models [Gröger 2018]. Yet, most companies are capturing only a fraction of the potential value of the actual data and analytics due to several reasons that hinder the full adoption. From a technical and application perspective the sheer flood of data cannot be stored or efficiently transmitted to the cloud, whereby newly arising IoT deployments require geo-distribution alongside location awareness and low latency [Bonomi et al. 2012]. From a legal perspective, incentive problems and regulatory issues exist which pose additional barriers [Henke et al. 2016]. This has driven the development fog computing enabled by the decrease in hardware size and costs alongside the massive adoption of specialization in hardware [Terzo

et al. 2019] which has resulted in unprecedented availability of computational resources at the edge of the network capable of running even sophisticated AI workloads [Dasher 2019]. The hierarchical fog computing model reflects a staged event processing model where preliminary processing is done at the edge to deliver real-time insights, provide higher availability and account for data locality needs while a subset of the results are moved to regional fog nodes or the cloud to perform additional processing. In this matter, democratizing the management of event-driven applications in fog computing extends the ability to realize IIoT scenarios beyond regular IT experts by empowering business domain specialists to independently perform necessary tasks with the assistance of specialized tools and platforms. Eventually, this aids companies to exploit the full potential towards establishing a data-driven culture.

In the following, we introduce the two aforementioned scenarios and specifically elaborate on various applicational demands and requirements that shape the application space for geo-distributed event-driven applications. Moreover, the Smart Urban Logistics scenario introduces a running example which is used throughout this thesis to exemplify the developed models, concepts and methods. Although the scenario in question targets an emerging field that has so far only been investigated in pilot applications, it nevertheless provides a comprehensible example that covers all relevant aspects to describe the problem domain. Further, we derive needs for flexible, geo-distributed event-driven application management in fog computing environments which we use to identify relevant gaps to be bridged to motivate our research contributions.

3.1.1 Factory 4.0

While the disruption achieved by increasing industrial automation is an immanent topic to date especially holding true in high-cost countries such as Germany, we are on the leap to witness a fourth industrial revolution driven by the IIoT towards cyber-physical systems, a new generation of systems integrating computational and physical abilities to interact with humans using a variety of new modalities [Lee 2008]. Over the years, increasing digitalization has dominated industrial domains initiating the convergence of operational technology and information technology [Pop et al. 2021]. While the former deals with the operation of physical processes and corresponding machinery and conveyer belts on shop floor level, the latter addresses the actual data and information flow. In this context, fog computing has been identified as the decentralized computing paradigm to achieve this convergence [Steiner and Poledna 2016; Alcaraz 2019]. Thus, various applications exist, of which continuous asset monitoring and the use of collaborative robots are typical examples [Wiener et al. 2020a].

Collaborative robots are one of the major automation trends in recent years that support companies in many applications, e.g., assembly, placement, handling or picking [Matheson et al. 2019]. Especially in the latter case, opposed to manually performing repetitive and dull quality inspection tasks, collaborative robots can be leveraged to execute these

tasks at high accuracy without exhaustion which is key to ensure high product quality, and thus enhanced customer satisfaction. Here, data from equipped sensors can be used to quickly assess the product quality by analyzing produced event streams in real-time to detect deviations from specified targets defined by manufacturing or quality engineers. Thereby, a dedicated EPA processing service either containing a conventional rule-based algorithm or more sophisticated AI-based algorithm must be deployed on an edge node in close proximity for timely actions. Depending on the result of the real-time quality check, the robot either continues with its actual assembly tasks or automatically sorts out products with insufficient quality for rework or as scrap. Moreover, aggregated process data are sent to a central location in the cloud where another EPA calculates relevant production *key performance indicators* (KPIs) for assessing the overall quality objectives, parallel to storing the data for general traceability purposes, to support historical analysis or to build up training data. With shortening product life cycles, increasing number of product variants due to customization, or lot-size one production, the changeability of a production system is key to withstand continuous changes and surrounding turbulences [ElMaraghy and Wiendahl 2014]. Similarly, individual event-driven processing services of deployed pipelines must adapt as well, for instance, to facilitate live modifications of dedicated analytics parameters when the robot retools for a new product variant.

Similarly in industrial operations, asset condition monitoring captures the state of machines and equipment of manufacturing companies while running. Machine health state can be continuously assessed by measurements from various sensors, e.g., acceleration, pressure, or vibration. These sensor event streams require timely analysis to detect any abnormal machine behavior to immediately initiate countermeasures. In addition, predictions on the remaining useful life for the asset or tools are among common predictive maintenance applications. This let manufacturers identify and fix causes for costly unplanned downtime to increase machine utilization and availability. However, assessing machine health is difficult. Given the average lifespan of industrial machines, brownfield IIoT deployments are common [Bhattacharjee 2018]. At the same time, sensor data comes at high velocity and volume where hidden problems and guesswork of domain experts can incur extra expenses. To alleviate domain experts and facilitate edge intelligence applications, pre-trained models can be wrapped inside *machine learning* (ML)-EPA and be deployed on local edge nodes located on the shop floor level. By using specialized machine connectors, employed algorithms can either signal anomalous behavior or predict future break down times to schedule enhanced maintenance tasks. Results are also used to update third-party systems on enterprise or factory level including business relevant enterprise resource planning as well as quality management systems, e.g., to optimize production plans or maintenance measures accordingly. Since most industrial providers have large machine parks running many of these processing services, resource pooling and thus sharing of available edge resources is common. In the event of high system load, some EPAs must be offloaded elsewhere, e.g., to nearby fog nodes.

3.1.2 Smart Urban Logistics

A well-known challenge for courier, express, and parcel service providers occurs as part of the so-called last-mile logistics, which comprises all activities that need to be done to deliver goods from distribution hubs to the final customer. Commonly referred to as the *last-mile problem*, this final leg of the supply chain constitutes to the most costs with estimates ranging up to 50% of the total shipping costs [Joerss et al. 2016] while bearing the greatest inefficiencies for many organizations. One of the reasons for this is the fact that the last mile usually involves several stops with a low package drop-off rate, in addition to general waiting times at package handover. As consumers increasingly turn to e-commerce to satisfy all their shopping needs, this leads to a wide variety of problems, especially in urban areas. Increasing noise pollution, rising emission levels as well as high traffic congestion are among the top issues in many cities, which are further exacerbated by concepts such as same-day delivery.

This has driven the development of new innovative concepts and ideas to shape the future of intelligent and economically friendly urban logistics solutions, chief among them are last-mile *delivery robots* [Joerss et al. 2016; Wiener et al. 2019]. Delivery robots are intelligent carrier platforms to autonomously deliver packages to the end consumers to cut high costs, reducing the carbon footprint while ensuring high customer satisfaction. From regional *satellite hubs* close to urban areas, packages might either be delivered at fixed defined handover times or delivered to dedicated *package boxes* to be picked up at any time [Joerss et al. 2016]. To perform the necessary delivery task, delivery robots are equipped with various sensors that constantly produce data at high volume and high velocity. Data include, above all, point cloud data from laser scanners for obstacle detection and avoidance, acceleration data for incident detection, and geolocation information from the *Global Positioning Service* (GPS) module for location tracking. Thereby, expected timely results with low latency, typical bandwidth constraints and network reliability concerns due to the mobile environment do not allow for a traditional cloud-only approach. By using a fog computing approach, computational resources at the edge and fog layer are exploited. In this context, delivery robots themselves act as edge nodes while package boxes can be considered as regional fog nodes. Still, the cloud is used for centralized fleet monitoring, for long-term storage purposes as well as for burst-out cases to run resource-intensive analytics which cannot be satisfied by neither the edge nor fog nodes. Figure 3.1 depicts a theoretical, but representative example of delivery robots operating in the city of Karlsruhe, with two satellite hubs and three package boxes.

Running Example: Location Monitoring for Delivery Robots. As delivery robots operate autonomously, it is crucial to employ monitoring mechanisms and immediately notify the fleet operator in case of any malfunctions or incidents to initiate timely actions. A common application in the field of fleet management deals with geofencing which allows to monitor mobile objects located by GPS [Reclus and Drouard 2009]. Another

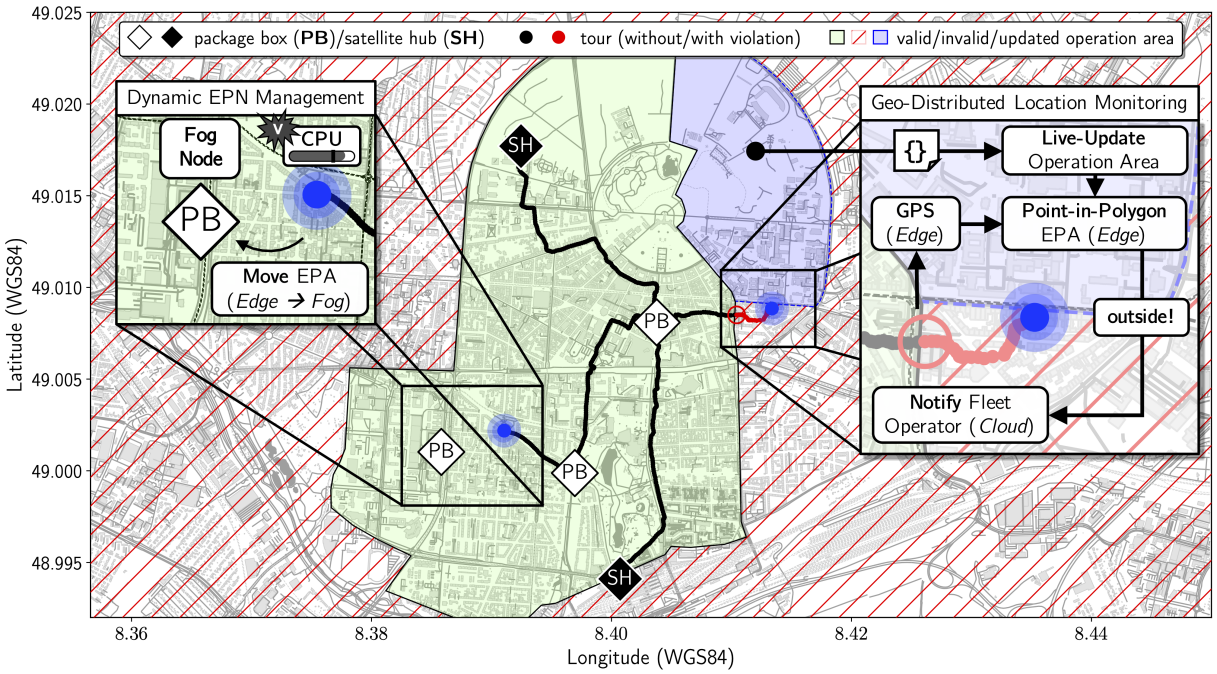


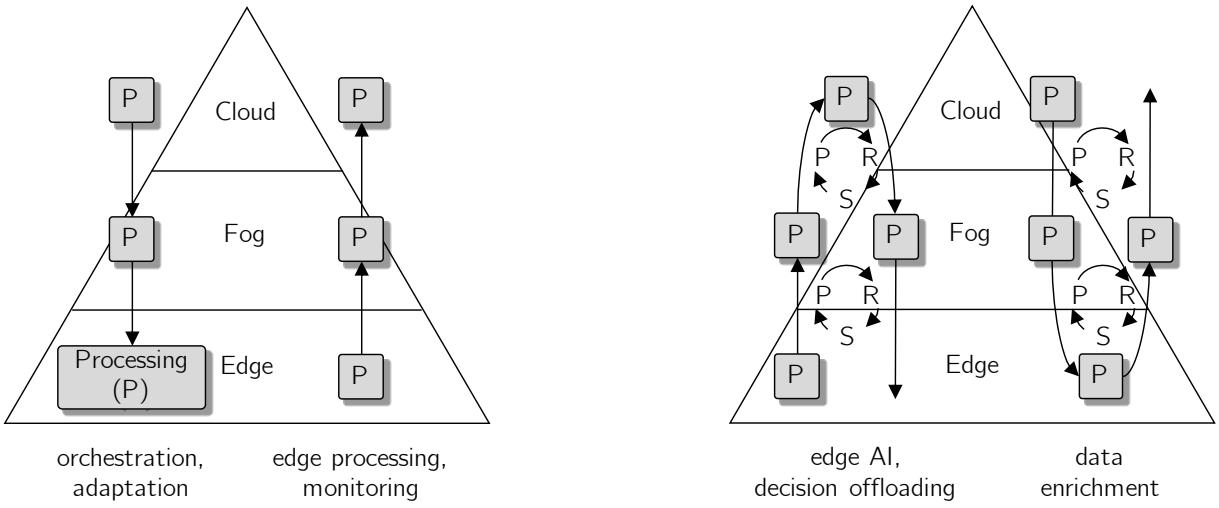
Figure 3.1 Running example: Dynamic EPN management and location monitoring for delivery robots. Map data © OpenStreetMap [OpenStreetMap contributors 2021]

set of geographical coordinates form a virtual boundary for a real-world geographical area, a so-called *geofence*, which is used to decide whether the tracked object is inside or outside this area [Reclus and Drouard 2009]. Consequently, the fleet operator specifies the valid operation area for the provided delivery service offer as shown in Figure 3.1. In order to perform the geofencing task, fleet operators need to create an event-driven application for *location monitoring* which requires to be geo-distributed and executed on respective delivery robots to benefit from the local edge processing capability. The geofence assessment can be realized by using a domain-specific EPA which implements the well-known point-in-polygon algorithm [Shimrat 1962]. Apart from this, the location monitoring pipeline comprises an additional EC component which sends a notification to the central fleet control center in the cloud to alert the fleet operator when a delivery robot leaves the operation area. Figure 3.1 exemplifies the running location monitoring application for two different tours, including a detected violation of the geofence criteria. In addition, as the real world is rather dynamic than static, arising changes might require adaptations to the running components. For instance, as the delivery robot service expands to new urban areas, fleet operators require to update the valid operation area employed in the point-in-polygon EPA. At the same time, it is not possible to stop, change and redeploy the application, so such configuration updates have to be done at run-time. Moreover, due to resource sharing on the delivery robot (edge node), in the event of over-utilization it is necessary to move certain EPAs to regional package boxes (fog nodes) in a burst-out manner which poses the need for a *dynamic EPN management*.

3.2 Needs

In general, the presented scenarios have in common that they decompose complex analytical problems into smaller subtasks, that are a collection of generic or purpose-built EPAs combined as part of a processing pipeline to achieve an overall business goal. These EPAs target different layers of the hierarchical fog architecture, ranging from edge and fog deployments for fast response times ensuring low latency and data locality needs, to cloud deployments for subsequent processing of pre-filtered or pre-aggregated data as well as overall monitoring and storage. This chain of logically interconnected event processing components in the EPN represents a dedicated processing pipeline. Individual pipeline elements, including event producers, event processing agents and event consumers, are executed at geographically dispersed locations along the cloud-edge continuum. Next, we discuss applicational needs in the context of the presented scenarios before elaborating on the democratization movement and derived needs in more detail. Thereby, we introduce and clarify the notion of the *citizen technologist*, a newly emerging organizational role which forms the basis of our research efforts.

Applicational Needs. The previously discussed scenarios and their IIoT-related characteristics ranging from low latency to data locality are proxies for general problem classes that shape the overall event-driven application space in fog computing. In this context, two model categories denoting the logical mobility of applications [Varshney and Simmhan 2017] can be distinguished at the highest level, which we refer to as *directed models* and *sense-process-respond models* as illustrated in Figure 3.2.



(a) directed

(b) sense-process-respond

Figure 3.2 Directed and sense-process-respond model for event-driven applications in fog computing. Adapted from [Varshney and Simmhan 2017].

In the *directed model*, processing pipelines and dedicated event producers, EPAs and event consumers are vertically oriented, either originating from cloud to edge or vice versa.

- *Centralized pipeline orchestration*—Here, the cloud acts as the central control center for orchestrating and managing geographically distributed processing pipelines. Besides the actual initial deployment process, life cycle management of these IIoT applications include adaptations of running processing pipelines in the form of reconfiguration as well as migration intends of individual processing services at run-time (see next).
- *Adaptive pipelines*—As processing pipelines and their underlying event processing paradigm operate on unbounded event streams, they are hypothetically no subject to being finished at any given point in time as it is the case for batch processing. However, as event streams might evolve or contextual circumstances might changes, e.g., when a changeover happens in a production process, it enforces processing pipelines to adaptively reconfigure some of their application logic of dedicated EPA components [Rehman et al. 2019]. Similarly, as surrounding context changes are common, especially on the fog and edge layer as computational resources are subjects to being exposed to the real-world, individual pipeline elements may be migrated from one node to another to support node mobility aspects or perform node maintenance work.
- *Edge processing pipelines*—Edge processing applications in the IIoT mostly deal with big data induced volume and velocity challenges. Hereby, data produced by sensor-rich assets such as machines or field aggregates like delivery robots are preprocessed, filtered and aggregated to remove irrelevant or erroneous data right at its source and only send relevant data to the fog or cloud to reduce overall network costs. This resembles a geographically distributed extract, transform, load process where data from real-world IoT devices ultimately reach the cloud for archival, analytics, or integration into other third-party IT systems.
- *Remote condition monitoring pipelines*—When looking at general deployment scenarios in fog computing, unsurprisingly having a central control center for observing and assessing the current state of industrial assets or field devices is crucial. From a monitoring and management perspective, domain specialists can quickly compare the current health state by using adequate visualizations as well as relevant metrics and KPIs.

The *sense-process-respond model* refers to closed-loop scenarios where commonly real-time responses are required as the result of intelligent analytics EPAs that preferably run at the edge layer. If feasible, the decision-making can also be delegated upwards and thus be offloaded to either the fog or cloud layer. Similarly, event streams coming from the cloud can be leveraged at the fog or edge layer, e.g., for enriching local sensor data with external information. In return, enriched data are sent back to supply other event-driven processing services in the cloud.

- *Edge AI pipelines*—In edge AI processing pipelines, individual ML-EPA pipeline elements are deployed in close proximity to the event source, either on edge or fog nodes, to provide minimal latency and ensure a high QoS even in situations of network outages that can occur in unreliable wireless setups. Thereby, results are used in many contexts and typically lead to a dedicated action. For instance, upon detecting a quality defect in the case of a visual inspection pipeline, a part is automatically sorted out for rework or as scrap. In contrast to stateless filter EPA or stateful aggregation EPA, ML-EPA mark special components of an analytical processing pipeline as they may address certain hardware requirements towards the underlying compute node. That is the case, when a *graphics processing unit* (GPU) is required for improved inferencing speeds.
- *Data enrichment pipelines*—Deployed EPAs at either the fog or edge layer may rely on other information provided by third-party systems to be used to enrich their own processing. For instance, issued delivery tours by a central tour planning system are sent to the delivery robot (edge node). A dedicated EPA keeps track of all stops and once a package delivery is completed, sends an acknowledgement back to the central tour planning system. Such information provide the basis to compute domain-specific process KPIs that are visualized in a central control center.

It is worthy to note, that a combination of these models may be conceivable, if needed to realize complex processing pipelines [Varshney and Simmhan 2017].

Organizational Needs. Throughout the past, we have witnessed recurrent movements that were subject to lowering rather high technological entry barriers by introducing new paradigms such as graphical flow-based approaches. Graphical flow-based approaches are manifested in more user-oriented systems that give business domain specialists technical capabilities outside their area of specialization, mostly by using abstraction as a key technique to hide unwanted complexities. To accommodate these rapid changes of digital transformations and streamline the work of domain specialists in solving mission-critical problems, the previously separate business and IT worlds have started to converge as a new class of individuals rise in today's enterprises. This marks an organizational paradigm shift towards acquiring a data-driven culture where business domain specialists are enabled to make data-driven decisions in industrial business processes themselves by combining their profound domain expertise with general knowledge in software development, engineering and data analysis techniques [Gröger 2018]. In this regard, appropriate tool support is crucial to enable non-expert users in realizing respective event-driven applications in a self-service fashion. Fortunately, advances in research areas such as automated machine learning [Thornton et al. 2013; Hutter et al. 2019], and the increasing prevalence of *no-code* or *low-code* platforms that recently received tremendous traction in industry, make the development of event-driven applications accessible to non-technical end users through graphical flow-based approaches. While the low-code approach reduces the amount of conventional hand coding to a minimum, the no-code

approach further extends this and dispenses any coding and lets non-programmers describe processing pipelines by intuitively configuring individual processing service blueprints. These newly emerging organizational roles can be grouped under the umbrella term *citizen technologists* shown in Figure 3.3, whereas *citizen developers*, *citizen data engineers* and *citizen data scientists* describe three concrete incarnations [Wong et al. 2015; Brinker 2018; Gröger 2018].

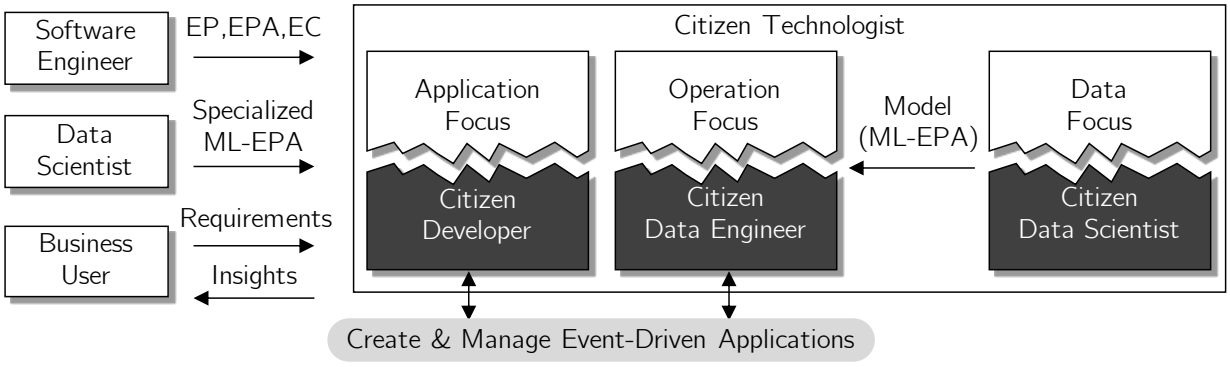


Figure 3.3 Interplay of common organizational roles with citizen technologists

Citizen developers are business domain specialists that are technically enabled to create and manage event-driven applications by using no-code event processing systems. Most notably, they are focused on building event-driven applications solving unique business domain challenges while incorporating requirements by external business user that in turn are provided with meaningful data-driven insights as the result of analytical processing pipelines [Wong et al. 2015]. Moreover, *citizen data engineers* play a vital role as their groundwork fundamentally alleviates citizen data scientists in facilitating the actual model and analytics tasks by providing cleansed, prepared and occasionally labeled training data. Not only are citizen data engineers capable of composing, building and managing processing pipelines along the overall data life cycle. They also know how to operationalize ML models in the form of ML-EPA and see data-driven projects through to production as they understand both domain and model requirements. Thereby, both citizen developer and citizen data engineers are closely interacting with other organizational roles: On the one hand, software engineers that develop and provide generic implementations for EPs, EPAs, and ECs. On the other hand, expert data scientists that create and provide specialized ML-EPAs. Similarly, *citizen data scientists* also possess a deep domain knowledge with clear understanding of corresponding needs where their primary job function is outside the field of data science in enterprise departments such as manufacturing, or process engineering [Tapadinhas and Idoine 2016; Gröger 2018]. Thereby, citizen data scientist are focused on the actual data analytics part.While citizen data scientists are not necessarily involved in creating and managing event-driven applications, they oftentimes provide relevant software artifacts in the form of ML-EPAs as the outcome of other no-code platforms with support for automated machine learning.

Throughout this work, when mentioning the term *citizen technologists*, we equally refer to citizen developers and citizen data engineers while withdrawing citizen data scientists from our consideration, as these roles reflect our focused target audience for managing event-driven applications.

3.3 Problem Statement

The previously described needs are used to identify problems of current solutions. We first outline existing approaches for graphical flow-based pipeline authoring and execution in no-code/low-code event processing systems and present limitations that currently hinder their applicability in fog computing settings. Thereafter, we state major conceptual and technical challenges based on the previous findings that need to be overcome on the way to democratize event-driven application management in fog computing.

3.3.1 Graphical Flow-based Systems

Recent developments of no-code event processing systems for pipeline authoring and execution allow to significantly lower technological entry barriers and provide tool support for non-technical citizen developers and citizen data engineers to model event-driven applications in a self-service manner. Thereby, several tools exist that fall into this category, chief among them is Apache StreamPipes[1]. StreamPipes is an incubator project of the Apache Software Foundation that is built on a graphical flow-based programming approach and provides a reusable toolbox to easily connect, analyze and exploit a variety of industrial event streams without any programming skills as depicted in Figure 3.4. Therefore, it leverages different technologies especially from the fields of event processing, distributed computing and semantic web. Originating from the foundational work of Riemer et al. [Riemer et al. 2014; Riemer et al. 2015; Riemer 2016] which we briefly touched in Section 2.2.3, StreamPipes allows to model 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. Pipeline elements are encapsulated in a service bundle, i.e., a dedicated pipeline element microservice, and contain the application logic that operates on incoming events in an event-driven fashion, once instantiated upon pipeline start. StreamPipes realizes the event-driven architecture using a technology-agnostic EPN design which is not exclusively limited to the EPA level but also reflected in the underlying message transport layer. Here, the message transport layer represents event channels in the EPN offering a topic-based publish/subscribe model for event dissemination between any interconnected pipeline elements with support for a variety of different messaging systems. On

[1] https://streampipes.apache.org/

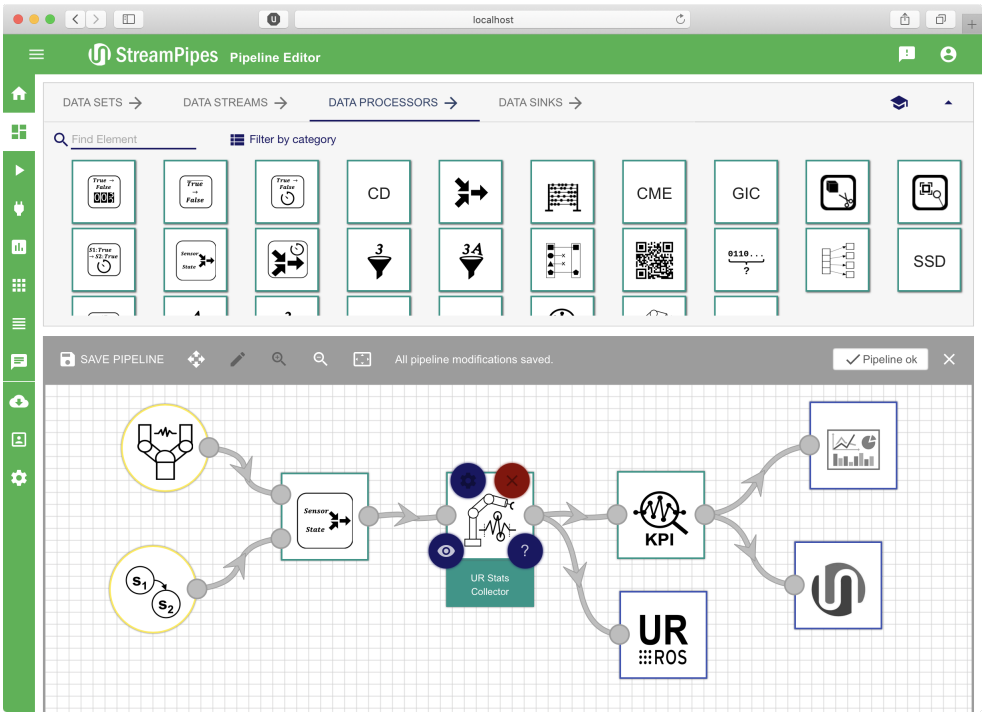


Figure 3.4 Exemplified processing pipeline in Apache StreamPipes

top, it uses semantics to provide guidance to non-technical users throughout the pipeline authoring process. Besides StreamPipes, other solutions for low-code flow-based programming of event-driven applications are available, both commercially, namely KNIME[2], crosser.io[3], and open source, namely Apache Nifi[4], Node-RED[5], or aFlux[6] with similar approaches, but different execution environments ranging from single-node runtimes to specialized stream processing engines in the field of big data.

Limitations. First, today's graphical flow-based systems are focused on static processing pipeline deployments in centralized cloud environments. This comes with inevitable downsides in the context of IIoT scenarios that require fast results, are limited by the available bandwidth, or bound to regulations regarding data sovereignty and data ownership. Second, heterogeneity of computational resources along the cloud-edge continuum is not explicitly taken into account by current systems. It remains unclear if a node is suitable for executing dedicated processing services. For instance, when deploying and executing ML-EPA, in most cases GPU acceleration is required for reasonable fast inferencing speeds. Apart from that, it is oftentimes critical to have a ML framework version installed on the given node that is compatible to the one used for training. In view of this,

[2]https://knime.com/
[3]https://crosser.io/
[4]https://nifi.apache.org/
[5]https://nodered.org/
[6]https://aflux.org/

existing solutions lack finding suitable deployment targets during the node selection process to assist citizen technologists. Third, unlike the cloud, fog and edge resources are physical resources exposed to the real world and thus are more prone to errors like node failures and unreliable network connections, especially in wireless setups with mobile edge nodes such as last-mile delivery robots. Hereby, a challenge is to deal with such intermittent network outages. Lastly, as current solutions run in centralized cloud environments, albeit distributed in data center clusters, it is usually the norm to have the underlying messaging system running inside the central same infrastructure. However, relying on a centralized messaging system is impractical in the context of distributed fog computing environments. Such an approach results multiple edge-cloud round trips in the worst case alongside other non-negligible problems regarding privacy violations and security concerns.

In summary, while the actual pipeline authoring part in state-of-the-art no-code/low-code event processing systems has been extensively studied, geo-distributing processing pipelines in the uprise of fog computing is complex. This introduces a new set of non-trivial conceptual and technical implications that require a holistic management approach to execute and administer event-driven processing services across all layers.

3.3.2 Managing Geo-Distributed Processing Pipelines

Distributing event-driven applications over pool of geo-distributed compute nodes is complex—especially for citizen technologists that lack necessary technical knowledge. Flexible deployment concepts as well as suitable pipeline element life cycle management strategies play a vital role for evolving current event processing systems. However, challenges arise on multiple fronts in view of managing geo-distributed pipeline deployments, especially in transient and heterogeneous fog computing environments. In the advent of data and analytics democratization, technical challenges are further exacerbated by an organizational paradigm shift with emerging functional roles of citizen technologists as the main driver. Ensuring reliable event processing under changing conditions is crucial to realize mission-critical scenarios as discussed.

Apart from the lack of solutions for citizen technologists to administer geo-distributed event-driven applications using no-code event processing systems, we are confronted with underlying technical challenges that arise in view of inherent characteristics of the fog computing model. Altogether, this currently hinders a full adoption of related application management activities by citizen technologists. To this end, the application management including geo-distributed pipeline deployment, operation and life cycle management faces a variety of challenges, including resource heterogeneity across all layers which is paired with use case specific demands. This ranges from the flexible pipeline element allocation on dedicated target nodes to account for data locality needs to the adaptation of pipeline elements in terms of live reconfiguration or migration to encounter business-driven changes. Realizing such needs requires deep technical

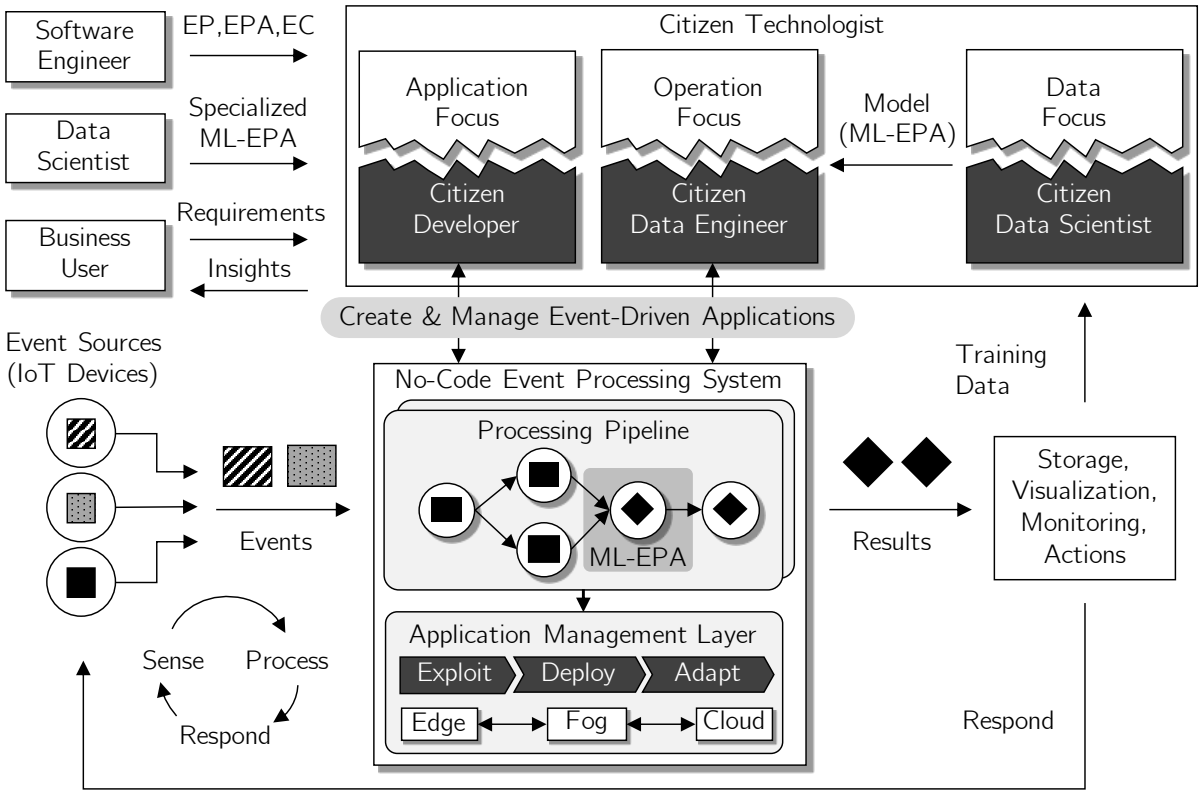


Figure 3.5 No-code event processing system to create and manage processing pipelines in the fog

knowledge in domains such as distributed event-based systems, orchestration, cluster management, which can hardly be stemmed by citizen technologists. Consequently, this poses the following challenges towards a holistic event-driven application management employed in a no-code event processing system as illustrated in Figure 3.5 comprising the following key elements:

- *Exploit*—Fog computing environments are largely heterogeneous in terms of computational resources which not only manifests quantitatively in the amount of memory or primary storage, but also with regard to dedicated chip architectures, operating systems and other capabilities (see Section 2.3.2). To improve the exploitation of resources requires to master the overall heterogeneity which implies the need to first create awareness on *what* specific resource characteristics are available in the first place. This demands having a rich node resource description in a machine-processable manner which is exposed by each of the participating nodes. Such a node description paired with a suitable system architecture facilitates event-driven application management in fog computing and builds the groundwork to realize additional key elements of the holistic management approach, namely geo-distributed pipeline deployment, run-time management and adaptation. Further, this allows to better assist citizen technologists in administrating pipelines along their life cycle, e.g., by proposing eligible deployment targets for individual pipeline elements.

- *Deploy*—Deploying processing pipelines that span over multiple geographically distributed locations from edge to cloud while still being able to cover the application life cycle is non trivial and involves three main building blocks. First, a central distribution mechanism and node management that oversees the pool of available resources which covers required deployment and adaptation aspects. Second, a management service on nodes that is responsible for all administration tasks along the life cycle of a pipeline element. Third, a flexible event stream routing to realize inter-node communication between adjacent pipeline elements hosted on different deployment targets in a technology-independent way.
- *Adapt*—Once the pipeline is deployed and running, contextual changes or new business demands might occur making it necessary to apply changes to individual pipeline elements. Related to the previously presented scenarios, this involves technical aspects of pipeline element reconfiguration to update or refine specific configurations, for instance geofence coordinates, as well as pipeline element relocation that addresses migration activities both vertically (across the layers) as well as horizontally (within a given layer). Therefore, a methodology is needed that reflects the event-driven processing pattern and is incorporated in the design of the application management middleware including the local management services with support for in-flight modifications of running pipeline elements, that are, modifications without redeployment.

In summary, current state-of-the-art no-code/low-code event processing systems lack an appropriate management middleware for fog operations that fully exploits available node resources, allows for geo-distributed deployment, operation and run-time modification of event-driven applications while hiding technical complexities from end users.

3.4 Conclusion

In this chapter, we presented the underlying motivation of our research as part of this thesis. We introduced two scenarios in the context of IIoT upon which we discussed relevant applicational as well as organizational needs and identified gaps of current systems. In conclusion, we are on the verge of a next generation of event-driven applications that are centrally modeled and executed over a pool of geographically distributed computational resources in heterogeneous fog computing environments. The need for a holistic application management approach for geo-distributed event-driven applications in the fog has been identified which targets the demands of citizen technologists. This serves as a crucial prerequisite that requires investigation in order to bridge the gap and allow data and analytics democratization from IT experts to knowledge workers to be further enhanced.

4

Related Work

In this chapter, we review current state of the art in the area of distributed event-based systems in Section 4.1 and application management in fog computing in Section 4.2.

4.1 Related Work on Distributed Event-Based Systems

In the following, we analyze related work in the context of distributed event-based systems and focus on two research directions, namely geo-distributed event processing and related programming models in Section 4.1.1 and geo-distributed publish/subscribe Section 4.1.2.

4.1.1 Geo-Distributed Event Processing

Distributed event processing has been broadly studied in both academia as well as industry with emerging novel application areas, such as the *Internet of Things* (IoT) [Dias de Assunção et al. 2018; Dayarathna and Perera 2018]. Event processing has encountered significant evolutions throughout the years from first-generation SPEs such as STREAM [Babu and Widom 2001], Aurora [Abadi et al. 2003], or Borealis [Abadi et al. 2005], to second-generation SPE for massively-parallel processing in the era of big data such as Apache Flink[1], Apache Spark with Spark Streaming[2] or Apache Samza[3], to more general application frameworks allowing to specify and execute user-defined functions denoting the third generation. In recent years, novel architecture designs induced by the plethora of devices in the context of IoT have been proposed which use decentralized computing paradigms such as edge and fog computing for data stream processing introducing the genesis of a fourth generation of stream processing [Dias de Assunção et al. 2018].

[1] https://flink.apache.org/
[2] https://spark.apache.org/
[3] https://samza.apache.org/

SpanEdge is introduced in [Sajjad et al. 2016] which is a unified approach for stream processing over cloud-based and near-edge data centers aiming to decrease overall latency and bandwidth consumption incurred by communication over wide-area network links in stream processing applications. The authors propose a master-worker runtime environment and introduce a task grouping scheme for Apache Storm[4] enabling developers to assign dataflow graph operators to so-called local-tasks that are executed in close proximity to the event source, e.g., cloudlets [Satyanarayanan et al. 2009], while downstream operators run inside global-tasks in the cloud. In [Renart et al. 2017], an edge-based programming framework is introduced that allows users to define data-driven reactive behaviors by orchestrating stream processing applications to geographically distributed computational resources while hiding low-level communication in an network overlay layer using the content-based publish/subscribe messaging mechanism. An elastic stream processing model for the IoT is presented in [Hochreiner et al. 2015; Hochreiner et al. 2016]. In contrast to static stream processing application deployments in typical SPEs at fixed locations, usually dedicated cloud data center, a resource elasticity mechanism is introduced that allows for cost-efficient cross-regional cloud data center operation to account for constant changes in data load in volatile IoT use cases. Thereby, initial deployment topologies are flexibly reconfigured at run-time by elastically provisioning computational resources to maintain a defined level of QoS to prevent situations of over-/under-provisioning. In follow up work, the Vienna platform for elastic stream processing (VISP) was proposed to realize elastic data stream processing in fog computing environments leveraging virtualized resources provided by containers to host individual processing operators [Hießl et al. 2019].

Similar approaches applying decentralized compute paradigms can be found in the domain of CEP. While CEP is similar to stream processing providing results without undue delay and the processing of input event streams, it uses a rule-based programming model in contrast to exploratory programming model in stream processing [Hießl et al. 2019; Dautov et al. 2018]. Schilling et al. identify the problem of centralized CEP systems that are not suitable for widely dispersed application scenarios such as nation-wide power grids in terms of processing costs and communication overhead [Schilling et al. 2010]. To this end, the authors present a distributed heterogeneous event processing system that interconnects a set of common heterogeneous centralized engines within network of nodes to enable geographically distributed processing while providing managing capabilities such as inter-node communication as part of the distributed system. [Hong et al. 2013] introduce Mobile Fog, a high-level, cloud-based programming model for developing and distributing fog applications by providing a simplified programming abstraction to dynamically scale applications at run-time using on-demand fog or cloud resources. Further, [Luthra and Koldehofe 2019] investigates CEP over fog infrastructures and present ProgCEP, a programming model facilitating the design and development of placement algorithms that dictate the geo-distributed deployment of CEP operators such as filters, joins for distributed query processing. To this end, ProgCEP builds on top of the

[4] https://storm.apache.org/

AdaptiveCEP query language [Weisenburger et al. 2017] and reuses a standard set of CEP operators which allow to specify QoS demands that must be fulfilled by the underlying fog computing infrastructure. In addition, it exposes commonly used resource metrics, e.g., latency, bandwidth, or CPU load on a central broker while also providing access to decentralized monitoring algorithms such as Vivaldi [Dabek et al. 2004], a lightweight algorithm to predict communication latencies between nodes.

Apart from code-based approaches, graphical flow-based programming models and systems which follow an EDA design are actively researched [Issarny et al. 2016; Ravindra et al. 2017]. In [Giang et al. 2015], a distributed dataflow programming model for IoT application development is proposed and validated as an extensions for IBM's Node-RED[5], called Distributed Node-RED [Blackstock and Lea 2014], which provides a mechanism to deploy processing pipelines called *flows* over heterogeneous compute resources by specifying static requirements and constraints, however lacking the support for dynamic adaptations. These shortcomings are investigated in the DDFlow declarative programming abstraction with support for dynamic adaptation in case of node failures or unstable network [Noor et al. 2019]. Despite the similarities to our work from an operations point of view, the proposed adaptation scheme lacks support to update processing semantics at run-time. Riemer et al. propose a semantic model and methodology for discovery and binding of real-time processing services from arbitrary stream processing engines that allow to author event processing pipelines using a graphical editor [Riemer et al. 2014; Riemer 2016]. The proposed ontologies for EPN constituents, e.g., semantic EPA, comprise both functional and non-functional aspects to abstract from heterogeneous execution environments in order to automatically verify processing pipelines on a semantic level at design-time. While the EDA principles are inherent to the suggested architecture facilitating geo-distributed event processing in principal, it lacks a resource model for the heterogeneous fog computing infrastructures and disregards application life cycle management aspects to account for dynamic adaptations which we aim to support with our work.

Besides academic efforts, several companies offer fog and edge computing platform services such as general-purpose platforms by major cloud vendors, namely Google Cloud IoT, Amazon Greengrass, Microsoft Azure IoT Edge, IBM Edge Computing, or more specific ones oriented towards IIoT in the manufacturing domain such as Crosser[6], FogHorn[7] or Nebbiolo[8]. While these offerings provide the possibility to distribute event processing capabilities to edge and fog nodes, they are generally statically configured and thus lack support in dynamic environments. In contrast, in our work we aim for a flexible deployment of processing pipelines and provide adaptation mechanisms such as reconfiguration or migration of individual processing services at run-time.

[5]https://nodered.org/
[6]https://crosser.io/
[7]https://www.foghorn.io/
[8]https://www.nebbiolo.tech/

4.1.2 Geo-Distributed Publish/Subscribe

While message-oriented middlewares, publish/subscribe systems in particular, have enjoyed immense research interest [Eugster et al. 2003] with systems such as Hermes [Pietzuch and Bacon 2002], the emergence of IoT and decentralized computing paradigms such as fog computing have led to new research challenges and opportunities, especially in the area of geo-distributed publish/subscribe.

Enhancements to the traditional publish/subscribe models in mobile operating environments are discussed in [Huang and Garcia-Molina 2004]. In addition to centralized broker architectures, several distributed approaches are presented for improved scalability and fault tolerance. Centralized broker architectures in particular are typically impractical in dynamic environments due to excessive network round trips, although they use existing techniques such as quenching [Segall and Arnold 1997] to discard events at the event source when there are no active subscriptions. More recent research investigate IoT-related challenges and incorporate concepts of QoS such as latency-awareness in the design of geo-distributed publish/subscribe systems. For instance, EMMA is distributed QoS-aware publish/subscribe messaging for edge computing applications that considers brokers on arbitrary edge resources and dynamically reconfigures client-broker connections at run-time based on their latency [Rausch et al. 2018]. Similarly, PubSubCoord has identified the need for edge-cloud integration in the realm of IIoT and proposes an autonomous, coordination and discovery service that provides a topic-based publish/subscribe model for operating over wide area networks [An et al. 2017]. The proposed architecture consists of a two-tier broker hierarchy deployed over a publish/-subscribe overlay network to achieve low-latency local area network and scalable wide area network event dissemination. Additionally, it exploits and extends the well-known Apache Zookeeper[9] distributed coordination service to dynamically create dissemination paths when participating publishers and subscribers leave and join the system. Unlike costly gossip-based approaches that use probabilistic broadcasting for event and query dissemination [Costa et al. 2005], TinyMQ was introduced targeting the challenges of wireless sensor networks [Shi et al. 2011]. TinyMQ is a content-based publish/subscribe middleware using an overlay network constructed on top of the sensor network that logically connects sensor nodes independent of their geographical position while providing virtual addresses as unique identifiers. Based on these virtual addresses, a hash-based mechanism for content-related message and routing mapping determines the corresponding subscription and notification paths for given queries and events to meet at dedicated rendezvous nodes along the network topology from publisher to subscriber. Moreover, a selective event routing strategy for inter-broker event dissemination in geo-distributed publish/subscribe systems is proposed in [Hasenburg and Bermbach 2020]. Thereby, geo-context information related to either the publisher or subscriber location are considered to select suitable rendezvous points. Apart from efficient event routing strategies,

[9] https://zookeeper.apache.org/

Zehnder et al. introduce edge-based event stream reduction strategies for distributed topic-based publish/subscribe systems to dynamically adapt events at run-time based on common data reduction and transformation strategies adhering to specific subscriber requirements. [Zehnder et al. 2019]. The authors propose the concept of virtual events, i.e., events that are reconstructed on subscriber-side without physical event transfer, by leveraging a semantic schema registry that stores information about subscriber and their individual data requirements as *Resource Description Framework* (RDF) triples in a machine-processable way. Publishers use this additional background knowledge to automatically decide how to preprocess events and what events to actually send. To this end, the proposed approach is a relaxation of the typical publish/subscribe paradigms that foresees strict decoupling between publishing and subscribing entities.

Numerous application messaging protocols in the context of IoT have been proposed, chief among the MQTT protocol. MQTT is an open OASIS standard which implements the topic-based publish/subscribe interaction scheme [Al-Fuqaha et al. Fourthquarter 2015]. Due to its lightweight nature and capability to support unreliable networks, MQTT is a designated candidate for IoT-related scenarios, especially in constrained environments such as edge and fog nodes, yet only providing general-purpose messaging capabilities. To this end, several research propose extensions of the standard implementation, e.g., MQTT for wireless sensor networks called MQTT-SN (formerly known as MQTT-S) [Hunkeler et al. 2008], or GeoMQTT which provides spatio-temporal filtering capabilities [Herle and Blankenbach 2016]. Besides, various open-source solutions exist that implement core functionality of publish/subscribe in general, mostly targeting the more simplified topic-based model such as Eclipse Mosquitto[10]. This also includes concepts of broker bridging where multiple broker instances deployed in different networks internally forward published messages on static topic schemes. Moreover, many commercial offerings exist most notably provided by major cloud vendors that are integral parts of current event processing solutions as previously discussed. However, these approaches are generally limited in their operational capability and not flexible as they rely on static configurations.

4.2 Related Work on Fog Application Management

Application management in fog computing includes several research topics which have been actively studied in the last years. Chief among them are the fields of orchestration and deployment of event-driven applications which we cover in Section 4.2.1, besides work on dynamic adaptation and migration schemes relevant to fog applications which we examine in Section 4.2.2.

[10]https://mosquitto.org/

4.2.1 Orchestration and Deployment

Advancements in distributed systems in terms of cluster management with early work on systems such as Omega [Schwarzkopf et al. 2013], Borg [Verma et al. 2015] or more recently Kubernetes[11] [Burns et al. 2016] are major enablers that serve as the foundation of today's orchestration and deployment concepts for event-driven applications in fog and edge computing environments. As such, these approaches have evolved around operating system-level virtualization technologies such as containerization [Bernstein 2014; Pahl and Lee 2015]. Container technology including the most prominent representatives such as Docker[12] or LXC[13], has become the de facto gold standard for application orchestration in many fog-related scenarios. In addition to code portability and isolation capabilities where computing resources are split up and dynamically shared, the overall lightweight character of containers with minimal resource footprint have led to a wide adoption in both academia and industry, in particular of importance for edge deployments in constrained setups [Ismail et al. 2015; Bellavista and Zanni 2017].

LEONORE is introduced which is a cloud-based framework for elastic provisioning of application components on resource-constrained edge devices for large-scale IoT deployments within a service-oriented infrastructure [Vögler et al. 2015]. Thereby, both push-based and pull-based approaches for application deployments are supported by making use of a local device agent service which either reacts on requests from the central provisioning service or independently schedules provisioning tasks to off-peak times. GeeLytics is an edge-based analytics platform to perform real-time stream processing based on container virtualization for task distribution and dynamic topology execution [Cheng et al. 2015]. In contrast to static deployments as part of conventional SPE, run-time adaptations of processing topologies are considered in order to flexibly react to changing demands on the consumer side (here actuators). Foggy, a framework for dynamic resource provisioning and automated container-based application deployment is proposed in [Yigitoglu et al. 2017], 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. Container virtualization is used to enable to maximize resources efficiency and task-sharing while ensuring isolation. The authors propose a policy-driven procedure to distribute multi-component applications by orchestrating corresponding containers within the fog infrastructure. FogFrame is another application orchestration framework for building and maintaining fog landscapes which provides necessary communication mechanisms for instantiating and maintaining service execution in the fog [Skarlat et al. 2018]. Therefore, FogFrame provides general functionality to pool computational resources at cloud or edge level in order to form a so-called fog landscape. Apart from fog landscape configuration mechanisms and application management [Skarlat et al. 2016], FogFrame uses heuristic

[11] https://kubernetes.io/
[12] https://www.docker.com/
[13] https://linuxcontainers.org/

algorithms for service placement on potential nodes based on resource constraints and QoS criteria in addition to mechanisms for recovering from overloaded nodes and specific failures. Moreover, numerous other related work exist in the area of microservice-based application orchestration and deployment [Santoro et al. 2017; Chang et al. 2017; Cheng et al. 2018; Wöbker et al. 2018], among some introducing new terminologies such as osmotic computing [Villari et al. 2016], or capillary computing [Taherizadeh et al. 2018].

Another related research direction is the model-driven orchestration and deployment which addresses challenges with regard to automating the provisioning and operational management of applications while ensuring portability and interoperability based on declarative deployment models [Endres et al. 2017]. According to a review by Bergmayr et al., the OASIS standard *Topology and Orchestration Specification for Cloud Applications* (TOSCA) is among the most prominent modeling approaches with implementations such as Alien4Cloud[14], Cloudify[15], and OpenTOSCA [Binz et al. 2013]. To this end, TOSCA uses the concept of service templates to declare application topologies and orchestration of services, while specifying corresponding components, i.e., infrastructure components (e.g., a Raspberry Pi), middleware components (e.g., a message broker), or application components (e.g., a Java software artifact) and their relations among each other using defined semantics [Wurster et al. 2020]. Generally, the described services are provisioned in designated infrastructures while their management behavior must comply with the given constraints or policies. Originally solely focused on cloud deployments, recent research investigates the usage of TOSCA in the context of IoT in geographically distributed infrastructures. In [Képes et al. 2019] 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 accessibility and security aspects, particularly for IIoT application deployments, it is focused on static configurations and lacks support to adapt deployments at run-time. Therefore, Tsagkaropoulos et al. propose to extend TOSCA for edge and fog deployments to support generic use case patterns ranging from traditional distributed execution paradigms such as hybrid cloud, or multi-cloud deployments to more recently emerging paradigms such as serverless computing including *Function-as-a-Service* (FaaS) [Baldini et al. 2017], or edge-based deployments [Tsagkaropoulos et al. 2021]. Therefore, semantic enhancements for two TOSCA flavors are provided, namely type-level and instance-level TOSCA templates, and integrated in previous work in the context of PrEstoCloud [Verginadis et al. 2017]. These TOSCA flavors are the main constituents of the suggested sequential deployment approach which consists of a design-time modeling phase and an execution phase. In the design-time modeling phase users define their applications including requirements, optimization and placement criteria based on an extension of the YAML specification for TOSCA. Once this specification of the type-level template is provided and validated, the

[14]https://alien4cloud.github.io/
[15]https://cloudify.co/

execution phase is triggered where an optimized instance-level description is generated and used to initiate the automated deployment procedure. Similar work extend TOSCA either for serverless computing and FaaS-based applications [Casale et al. 2020], or in combination with other modeling efforts such as the *Open Cloud Computing Interface* (OCCI) [Challita et al. 2021] aiming at the standardization of a common API for cloud offerings, primarily Infrastructure-as-a-Service. However, all of these approaches are not explicitly focused on geographically distributed deployments and do not focus on specifics regarding multi-connected event-driven applications in the sense of processing pipelines. Apart from TOSCA, an analytical pipeline definition and deployment language, referred to as PADL, is introduced which allows to model and deploy distributed analytical pipelines over edge and fog computing environments in a technology-independent and infrastructure-agnostic fashion [Díaz-de-Arcaya et al. 2020]. Therefore, PADL is focused on operationalizing AI pipelines with support for predictive model interchange formats such as *Predictive Model Markup Language* (PMML) [Guazzelli et al. 2009], or *Portable Format for Analytics* (PFA) [Pivarski et al. 2016]. PADL allows users to define deployment flows by specifying input and output channels for different models, resource requirements, or node constraints using a domain-specific language based on the YAML format. While all above approaches allow to declare orchestration manifests and enact fog deployments, they are text-based making it generally impractical for our targeted audience to define such deployments and reason about. In contrast, our approach aims to automatically generate node resource information using an extensible node description at setup time to assist citizen technologists in the deployment and administration process at design-time.

Moreover, several open-source container orchestrators exist that evolved around the idea of unified edge-cloud deployments either extending Kubernetes such as kubeEdge[16], k3s[17] or providing similar functionality ioFog[18], or fog05[19]. However, their focus lies on cluster management aspects such as resource management, networking, besides high-availability of service deployments, yet neglecting needs induced by geo-distributed event-driven applications and our non-technical target audience (see Section 3.2).

4.2.2 Reconfiguration and Migration

Statically configured deployments of processing pipelines that span the hierarchical fog computing architecture are not suitable to commensurate the dynamics of such environments and support continuously evolving analytical needs. Dynamic adaptation strategies and mechanisms have been proposed over the years in the context of application reconfiguration and migration.

[16]https://kubeedge.io/
[17]https://k3s.io/
[18]https://iofog.org/
[19]https://fog05.io/

Early work in dynamic query modification in stream processing facilitates the online modification of continuous queries by introducing the notion of *control lines* that carry messages with revises parameters and functions in order to update individual operator parameters as well as the operator behavior themselves [Abadi et al. 2005]. Similar ideas about dynamic operator adaptation can be found in [Andrade et al. 2014], where individual operators may exchange dedicated messages via control directives in order to alter the modus operandi of another operator. Moreover, hierarchical control patterns for self-adaptive elastic stream processing over fog computing environments by means of scaling and migration actions are proposed in [Cardellini et al. 2018] in accordance to the *Monitor-Analyze-Plan-Execute* (MAPE) loop as a pattern for self-adaptive systems [Kephart and Chess 2003], yet only taking hardware-related node resources into consideration with strong focus on query-driven stream processing.

Cloud4IoT is a platform which offers automatic microservice-based container orchestration and deployment and allows for dynamic configuration in terms of horizontal and vertical container migration at run-time [Dupont et al. 2017]. Therefore, it provides a discovery mode to autonomously detect new bluetooth low energy devices in the proximity of an IoT gateway. This enables a plug-and-play integration of new sensor devices while using signal strength to deploy and undeploy containers on device join/leave actions. In addition, it leverages state-of-the-art container and orchestrator technology, namely Docker and Kubernetes. In contrast to our work, we assume processing pipelines that are comprised of multiple, event-driven processing services as our application model while Cloud4IoT considers standalone, request/response services that are stateless. Further, the discussed migration approach is limited by the capabilities of the underlying technology while relying on a centralized cloud orchestrator to initiate any migration action which is a hindrance in unreliable network environments. We argue that such offloading decisions can be decentralized to a node-level management service that can observe its context and self-reliantly trigger such relocation actions in combination with a centralized coordination. Preliminary work on a container-based approach discussing an infrastructure-level and application-level controller for autonomic data stream processing applications in fog computing infrastructures is presented in [Brogi et al. 2018]. To this end, the authors propose a two-level adaptation approach introducing a fog node controller to manage node resources and application controller to manage the actual application and motivate this by an intra-node and inter-node scenario. While the intra-node scenario depicts how application containers can be autonomously scaled based on resource requests issued by the application controller, the inter-node scenario investigates how application containers can be migrated between nodes upon migration requests issued by the fog node controller. Yet, the presented approach is technology-dependent relying on native Docker features for elastic resource assignment besides using the project checkpoint/restore in userspace[20] for snapshotting and migrating application containers as a whole following a pause-and-resume approach [Heinze et al. 2014]. Thereby, the application is first paused, with their internal state being persisted, transferred and restored at the new destination

[20] https://criu.org/

to resume operation. Similar approaches to support mobility aspects are investigated in other work, e.g., in terms of virtual machine-based migration [Bittencourt et al. 2015], or container-based migration in companion fog computing [Puliafito et al. 2018]. Another related work is Foglets, a programming model for managing geographically distributed situation-aware event-driven applications in the fog [Saurez et al. 2016]. Therefore, Foglets exposes designated APIs to cater the application development and deployment process. Moreover, it allows to place individual processing tasks at different levels of the hierarchical fog architecture while providing algorithms for migrating individual application components to support device mobility.

4.3 Conclusion

Albeit existing research employs similar management capabilities for event-driven applications as our approach, either an application-centric or infrastructure-centric view is taken. Yet, in order to fully exploit the potential both aspects need to be equally reflected in a holistic application management approach for geo-distributed event-driven applications. Besides geo-distributed event processing and publish/subscribe which, taken in isolation, have proven their applicability in resource-constrained fog computing environments, such a holistic application management approach also considers deployment and adaptation aspects and provides an abstraction layer to mitigate any heterogeneity issues induced by the manifold of node types. Current approaches still rely on partly complex domain-specific languages and notations for describing dedicated deployment topologies in the model-driven engineering domain. In addition, most of the presented related work focuses on declarative programming models that are not suited for non-technical programmers who require graphical approaches to model EPNs that only few work address—yet either lacking support for users to select suitable deployment configurations or not facilitating mechanisms to dynamically adapt event-driven application deployments. Current approaches lack adequate modeling efforts towards capturing relevant resource information beyond commonly used hardware-related aspects in a machine-processable way which is crucial to enact heterogeneity-awareness in the application management layer to better assist citizen technologists along the application life cycle. Consequently, a holistic management approach for the next generation of event processing systems in fog computing is required.

Part III

Main Part

5 Requirements

This chapter presents various requirements this work builds upon. Thereby, requirements are derived from the outlined research questions which are further complemented by identified needs and problems elaborated within the motivation while reflecting observations from related work. In the following, we introduce the requirements elicitation process in Section 5.1 before describing the resulting model and architecture-specific requirements in Section 5.2 and system-specific requirements in Section 5.3.

5.1 Requirements Elicitation

As an integral part within requirements engineering, requirements elicitation refers to the process whose objective is to seek, uncover, acquire, and elaborate requirements for computer-based systems [Zowghi and Coulin 2005]. As such, requirements elicitation involves a set of activities that include (1) understanding the application domain, (2) identifying the sources of requirements, (3) analyzing the stakeholders, (4) selecting the techniques, approaches, and tools to use and (5) eliciting the requirements from stakeholders and other sources [Zowghi and Coulin 2005].

In this thesis, the required activities to derive relevant requirements in the course of the elicitation process are presented and discussed in various chapters and sections. Gaining understanding of the *application domain* marks a crucial success factor in the early stages of the overall procedure. Our work targets application domains such as the IoT and IIoT that require a holistic management approach for event-driven applications in fog computing environments to facilitate the imminent need for data and analytics democratization, as described in Chapter 2 and Chapter 3. *Sources of requirements* are manifold, however in information systems design, requirements are mostly induced by stakeholders such as our target audience, namely citizen technologists. Our work analyzes potential *stakeholders* and shows an interplay of various organizational roles in the context of our targeted application domain and prioritizes our intended target audience of citizen technologists, yet still reflecting concerns of other related roles. *Techniques, approaches and tools* presents another relevant activity to gather requirements. A common approach in research relies on deep analysis of existing approaches on a given subject of investigation. We follow this

principle and discuss shortcomings of current no-code/low-code solutions for graphical flow-based systems, in addition to state-of-the-art approaches which we reviewed and assessed from different viewpoints within Chapter 4. Identified gaps are further taken into consideration when deriving requirements.

Following, we present derived requirements according to *model-specific* and *architecture-specific* aspects focusing on Research Question 1 as well as *system-specific* aspects related to Research Question 2 and Research Question 3 as summarized in Figure 5.1. At this point, it is worthy to note that even though we discuss these requirement categories independently they are strongly interrelated as each category represents an essential pillar in pursuing our overall research objective.

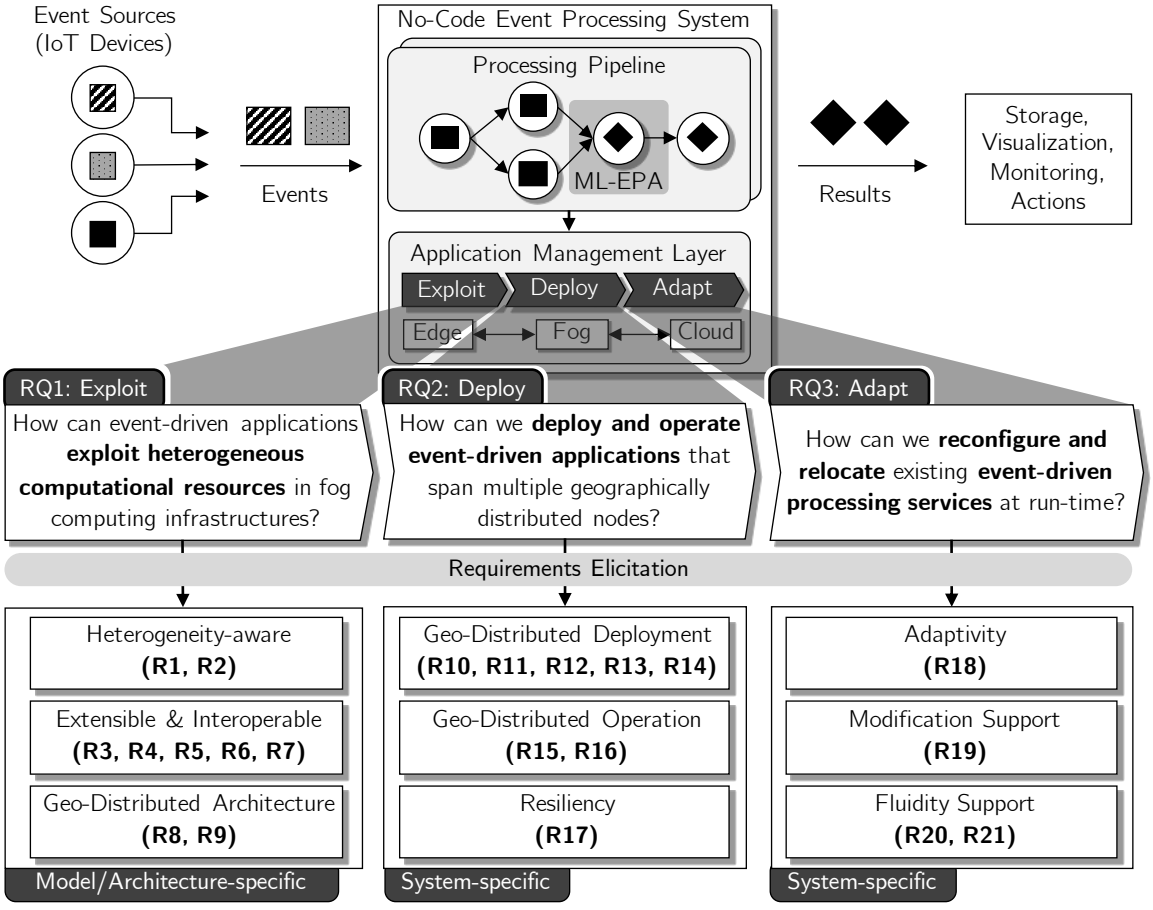


Figure 5.1 Mapping of research questions to requirements

5.2 Model/Architecture-specific Requirements

Within this category of requirements, we cover aspects related to the *node model* and the geo-distributed *system architecture* regarding Research Question 1. The first requirement

entails matters regarding one of the crucial points in application management in fog computing, namely the diverse resource capabilities of edge, fog and cloud nodes subsumed under the term heterogeneity. Single-board computers such as Raspberry Pi's (e.g., a Raspberry Pi Model 4 with 4-core CPU, up to 8 GB memory) or more recently ARM64-based NVIDIA Jetson models (e.g., Jetson Xavier with 8-core CPU, 32 GB memory and GPU support) are widely considered popular edge nodes [Bellavista and Zanni 2017; Yigitoglu et al. 2017], while special-purposes industrial PC as fog nodes in addition to on-demand virtual machine offerings in the cloud further complement the resource landscape. In Section 3.3.2, we have argued that operating event-driven applications in ever-changing compute environments is enhanced by explicitly dealing with such heterogeneity aspects. This implies creating *heterogeneity-awareness* and transparency for the managing middleware over available resources. In this regard, node resources go beyond traditional hardware-related resource types, e.g., CPU, memory, or non-volatile storage devices in order to also considering additional capabilities that are relevant to meet the application demands. This leads to the following requirement:

Requirement R1: Heterogeneity-aware
Each node should expose a node description encapsulating available resources and capabilities.

While the former requirement addresses aspects towards the exposure of the node description to the outside world, the following one contemplates to how these resource information are obtained. Therefore, this requires the instantiation of the node description to be *platform-agnostic*, i.e., independent of underlying infrastructure specificities.

Requirement R2: Platform-agnostic
Instantiating the node description should be agnostic to underlying infrastructure specificities.

As specialization in hardware has been massively adopted to satisfy the insatiable need for computing capability [Terzo et al. 2019], the *extensibility* of such a node model plays a crucial role in the given context. As applicational resource demands increase, in particular due to advancements in AI applications moving towards the edge of the network, novel resource types appear with new capabilities such as hardware-accelerated computer chips (see Section 2.3.2). Moreover, additional *domain-specific knowledge* in the form of metadata must be added to a node description in order to better assist citizen technologists in selecting eligible deployment target nodes. For instance, such domain-specific knowledge may indicate certain privacy zones, affiliations to a given logical location on factory or shop floor level or arbitrary other information that facilitate to structure and organize nodes according to the respective applicational use case.

Requirement R3: Extensibility
The model can be extended to employ new resources characteristics.

Requirement R4: Domain Knowledge
The model supports additional domain-specific metadata which can be altered at run-time.

The next requirement is *interoperability*. In order to ensure that the exposed model description is interoperable, the model representation should build on existing standards.

Here, RDF is as a well-established concept in web technologies to describe data models. This allows us to integrate and reuse existing vocabularies for many domains.

Requirement R5: Interoperability
RDF is used as a data model to represent the node description.

Our goal is to allow citizen technologists to focus on building and deploying event-driven applications to create an added value. In this context, the EPN and thus the dataflow model (see Section 2.2.3) is well suited to address needs of IoT applications and alleviate the distribution of individual processing services on remote nodes. Hence, as software engineers provide implementations for individual EPN components (see Section 3.2), they must be provided with model primitives and concepts for extending the existing vocabularies to address specific resource requirements as they can best estimate how much resources are essential to execute the event processing logic. Here, existing vocabularies for EPN components [Riemer 2016; Zehnder et al. 2020] should be obtained and extended to consider resource requirement declarations as in the case of an ML-EPA requiring GPU support.

Requirement R6: Dataflow Composition
The application model forms a geo-distributed EPN of different event-driven components.

Requirement R7: Requirement Declaration
Individual EPN components can declare resource requirements.

Apart from the actual model view, there is also an *architecture* view which deals with requirements to structure the geo-distributed management architecture which builds the foundation for system-related considerations discussed in the subsequent section. To orchestrate and operate event-driven applications over a pool of geographically dispersed resources, this enforces the requirement of having a *two-level application management* approach. This approach is required to comprise both mechanisms of centralized management and local management. While the former oversees the node cluster and acts as the core entry point for users, the latter is engaged in management activities on node level ensuring smooth operation along the application life cycle in a self-aware manner.

Requirement R8: Two-level Management
The general architecture comprises a central coordinator and local management entities.

Further, while cloud resources are commonly considered vastly scalable, edge and fog resources typically imply limited scalability as they are mostly physical nodes. Consequently, resources are likely to be shared between different types of services that are executed on a given node. This requires for lightweight resource *isolation* approaches that are best addressed by leveraging current state-of-the-art virtualization technologies such as containerization to accommodate event-driven processing services and their dedicated event processing logic.

Requirement R9: Isolation
Management and event-driven processing services are bundled and provisioned as containers.

5.3 System-specific Requirements

In this section, we present requirements related to the *system* associated with Research Question 2 and Research Question 3. These requirements are influenced by non-negligible fog computing characteristics (see Section 2.3.2) apart from applicational needs (see Section 3.2). In addition, the requirements cover technical aspects along the application life cycle to ensure an application management which incorporates necessary deployment and adaptation concepts. The first requirement deals with *geo-distribution*. With the ever-growing interest in data-driven decision making and emerging applicational needs tied to specific requirements in terms of latency or privacy considerations, it is essential for a managing middleware to support event-driven application deployments in geo-distributed environments such as fog computing. Therefore, the system needs to be capable of proving geo-distributed deployment and execution concepts to deploy arbitrary pipeline topologies. To realize this deployment step, participating nodes are required to inhabit a management entity which registers its capabilities as part of the exposed node description (see Requirement R1: *Heterogeneity-aware*) at the central coordinator and handle management matters along the application life cycle in an *autonomous* fashion. Moreover, all management tasks should be handled system-internally which requires *abstraction*.

Requirement R10: Geo-Distribution
Arbitrary pipeline topologies can be deployed and executed on geographically dispersed nodes.

Requirement R11: Node Autonomy
Nodes manage local deployments, continuously observe their context and trigger decisions.

Requirement R12: Abstraction
The system should abstract from low-level management details.

This typically leads to a trade-off between hiding technical details from citizen technologists while providing enough flexibility to adequately express deployment and operation preferences. To this extent, the compatibility between pipeline element requirements and node resource offers must be validated such that only suitable deployment targets are shown to the user, where suitability is assured according to a *matching* mechanism. To further support citizen technologists along the pipeline deployment process, an approach is required which provides flexible deployment options and gives freedom of choice to fulfill use case-specific demands. Thus, configurable and extensible pipeline *deployment* and *operation* options should be provided.

Requirement R13: Matching
The system can match node resource requirements and node resource offers and only select eligible deployment targets for event-driven processing services.

Requirement R14: Deployment Support
The system supports citizen technologists by providing pipeline deployment options.

Requirement R15: Operation Support
The system supports citizen technologists by providing pipeline operation policies.

As processing pipelines potentially span across multiple layers along the cloud-edge continuum, event streams between any two interconnected pipeline elements must be flexibly managed. This includes leveraging location information for intra-node communication and providing a mechanism for inter-node communication.

Requirement R16: Event Routing
The system is capable of ensuring flexible intra-node and inter-node event routing.

In dynamic environments such as fog computing where a subset of nodes are exposed to real world, component failures can lead to unanticipated behavior and unavailability [Cristian 1991]. In particular, unreliable network and temporary node unavailability are more likely to occur than in the cloud. This also affects the previously explained event routing as target nodes for inter-node communication might not be reached. Thus, this poses the requirement to handle situations of potential node unavailability or unintentional node restarts in a *resilient* and robust manner to ensure business continuity.

Requirement R17: Resiliency
The system can deal with unreliabilities in fog computing infrastructures in a resilient manner.

Next, we regard requirements that evolve around aspects of *adaptivity*. In Section 4.2.2, we mentioned that statically configured pipeline deployments are infeasible to account for both the dynamic nature of fog computing as well as evolving analytical and business needs. The same is true if edge nodes are mobile or computational resources are heavily utilized. In any case, this requires pipeline *evolution support* by having a generic adaptation methodology for both *reconfiguration* and *migration* that facilitates an adaptive behavior of processing pipelines at run-time with minimal interruption. Lastly, the latter point forms the basis to incorporate capabilities of *context-awareness* that, in principle, enable systems to detect and respond to changes in their situated environment [Schilit et al. 1994]. Consequently, nodes should be able to continuously observe their system context and assess when certain resource capacities are exhausted to conduct offloading decisions.

Requirement R18: Adaptivity
The system employs a generic adaptation methodology allowing event-driven applications to evolve.

Requirement R19: Reconfiguration Support
The system supports changes in the processing logic of pipeline elements.

Requirement R20: Migration Support
The system supports to migrate pipeline elements between nodes.

Requirement R21: Context-aware Offloading
Nodes can self-reliantly initiate offloading decisions based on observed context changes.

6

Resource Exploitation

In this chapter, we present results with regard to Research Question 1 and demonstrate how event-driven applications can *exploit* heterogeneous resources in fog computing. Therefore, we first categorize and describe key *heterogeneity dimensions* for administrating geo-distributed event-driven applications in Section 6.1. Next, we formalize the *pipeline application* and *fog infrastructure model* in Section 6.2 and Section 6.3. Afterwards, we propose a generic and extensible *node model* in Section 6.4 and detail related concepts which build the foundation for the subsequent investigations within this work. In Section 6.5, we present an *architecture* for the overall management approach which integrates the node model to create heterogeneity-awareness. Lastly, we briefly highlight provided *tool support* in Section 6.6 and summarize the chapter in Section 6.7.

6.1 Heterogeneity Dimensions

A crucial factor for the management of event-driven applications in fog computing relates to dealing with heterogeneity aspects of the underlying compute infrastructure. Contrary to virtual machines in the cloud, edge and fog nodes are typically physical, inherently heterogeneous and come at varying form factors and characteristics including dissimilar processor architectures, different operating systems and hardware resources as discussed in Section 2.3.2. This poses new requirements towards a uniform resource management for event-driven applications within fog computing, as the general assumption of homogeneous node resources must be discarded. In particular during resource allocation in the pipeline deployment phase, it is crucial to find eligible execution targets according to individual pipeline element requirements, user preference and specific capabilities of nodes. As an initial step, we categorize heterogeneity aspects in fog computing infrastructures into three general dimensions as shown in Figure 6.1:

- *Hardware*—The hardware dimension subsumes all hardware-level node resources. This includes system resources such as number of CPU cores and processor architectures (e.g., x86, ARM32, ARM64), amount of random access memory, and secondary storage types (e.g., solid-state drive, hard disk drive) and storage amount, in addition to novel resources such as GPU (e.g., NVIDIA Volta, or Maxwell).

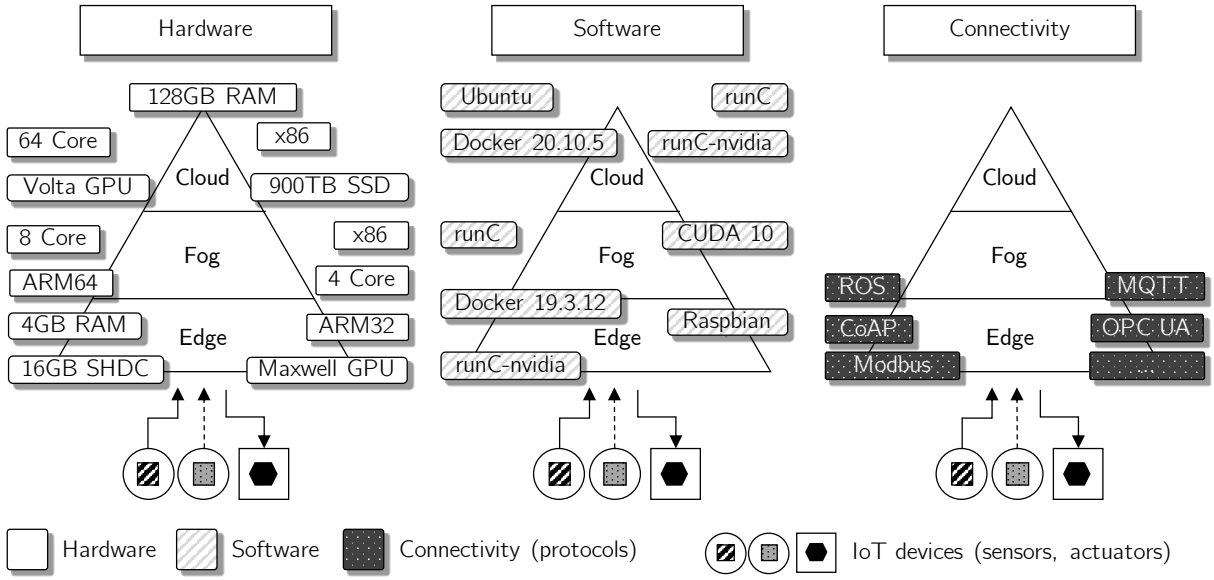


Figure 6.1 Heterogeneity dimensions and varying manifestations in fog computing

- *Software*—The software dimension covers aspects in conjunction with all software-level information. This includes information such as operating system type and version (e.g., Linux, macOS, Windows), container runtime type and version (e.g., general `runC`[1], or `runC-nvidia`[2] to support GPU-accelerated ML-EPAs), alongside several other information such as specific driver versions (e.g., CUDA 10) that play a vital role from a resource management perspective.
- *Connectivity*—Lastly, the connectivity dimension denotes all specific node capabilities with regard to IoT device accessibility to either collect data or trigger actions. This especially targets event producers in the in the directed application models, and both event producers and event consumers within SPR application model (see Section 3.2). In particular in the IIoT, a large number of IoT protocols exist. That include protocols build on open protocol standards, e.g., MQTT, CoAP, ROS, Modbus, OPC UA, as well as propriety protocols which are mostly vendor-specific. Despite rare cases where IoT devices actively push raw data to cloud-level message brokers, this is generally assumed to be impractical in the IIoT due to privacy, or bandwidth concerns. Hence, it is more common for nearby edge and fog nodes to be capable of connecting to IoT devices to collect data, if not edge nodes themselves are equipped with sensor modules such as a camera.

[1] A run-time container that handles kernel-level interaction of running containers in compliance with the OCCI specification (https://opencontainers.org/) and used by container technologies such as Docker.
[2] A forked and modified version of `runC` served by NVIDIA, providing a run-time container that is GPU-aware and used by container technologies such as Docker.

6.2 Pipeline Application Model

In this section, we introduce a formal model for processing pipelines that serves an abstract definition for modeling event-driven applications which we consider as an input for our application management middleware. To this extent, we briefly introduce related pipeline element types that assimilate the abstract definition of EPN components, and thus fall into any of the following three categories:

- *Adapter*—An adapter allows to connect to an external event source in a configurable (e.g., MQTT, ROS, OPC UA) way in order to provide an event stream [Zehnder et al. 2020]. Each event stream provides a *description graph* with information on the event schema itself (i.e., event properties) and the designated stream grounding (i.e., a transport format such as JSON, and transport protocol such as MQTT, or Kafka) that is used to publish events. Adapters are wrapped inside an EP-service. Through the concept of service invocation, EP-services instantiate adapter instances.
- *Processor*—A processor provides an arbitrary type of event processing logic that is applied on an event stream at run-time. Processors are encapsulated in an EPA-service and expose a *description graph* which contains information on supported stream groundings, static properties (i.e., user-defined configuration options), or its output event schema. This description is used to verify compatibility between any two interlinked element on a semantic level. Like adapters, EPA-services also instantiate processor instances via service invocation [Riemer 2016].
- *Sink*—A sink is similar to a processor and thus one or more sinks are wrapped within an EC-service, at the exception that sinks do not produce any output event streams, thus marking the end of a processing pipeline [Riemer 2016].

Pipeline Element Requirements. As previously mentioned, semantic compatibility between various pipeline elements based on stream requirements have been extensively researched [Riemer 2016]. However, these *data-level* requirements only represent a subset of requirements that are indispensable to facilitate geo-distributed deployments. Hence, we propose additional requirements in compliance with the stated heterogeneity dimensions from Section 6.1 which are summarized in Table 6.1. First, there are *hardware-level* requirements which state the resource amount a pipeline elements needs in order to operate smoothly. This includes the number of CPU cores, amount of memory or disk space, or GPU-support for efficient model inferencing in case of ML-EPAs. Second, there are *software-level* requirements. These requirements incorporate information such as operating system types, container runtime version or other libraries. Finally, there are *connectivity-level* requirements which cover network and IoT protocol related aspects. As such connections to IoT devices only occur for EPN constituents that are interfacing external producers or consumers. Consequently, connectivity-level requirements are only appropriate for adapters and sinks, excluding processors.

Requirement Type	Adapter (A)	Processor (P)	Sink (S)
Data	●, ▲	●, ▲	●, ▲
Hardware	▲	▲	▲
Software	▲	▲	▲
Connectivity	▲	—	▲

● Requirements types in [Riemer 2016]
▲ Requirements types in this thesis
— Processors do not pose connectivity requirements

Table 6.1 Comparison of pipeline element requirement types

Formal Application Model. The pipeline application model consists of a set of connected pipeline elements wrapped inside dedicated EP-services, EPA-services, and EC-services. More formally, we model a processing pipeline $\mathcal{P}$ denoting a multi-component, event-driven application as an EPN, where a pipeline is represented as a triple $\mathcal{P}{=}(\mathcal{V}, \mathcal{E}, \Pi)$. Here, $\mathcal{V}{=}A \cup P \cup S$ is a finite set of all *pipeline elements* denoting vertices, such that A is a set of adapters with $A \neq \emptyset$, P is a set of processors, S is a set of sinks with $S \neq \emptyset$. Moreover, $\mathcal{E}$ is a set of *directed event stream edges* that connect adjacent pipeline elements, such that $\mathcal{E}{=}(A \times P) \cup (A \times S) \cup (P \times P) \cup (P \times S)$ without cycles $\forall p \in P : (p_i, p_i) \neq \mathcal{E}$. Consequently, the minimal valid pipeline consists of one adapter connected to one sink, with no intermediary processor. Further, Π denotes a set of *deployment and operation policies* which are further detailed in Section 7.3.1. Lastly, we introduce the notion of *pipeline element adjacency* alongside the definition for *predecessor* and *successor* pipeline elements. The adjacency characteristic describes the logical connection among any two pipeline elements in the processing pipeline and allows to infer useful knowledge from a deployment and operation perspective.

> **Definition 7 (Pipeline Element Adjacency).** Within a pipeline $\mathcal{P}{=}(\mathcal{V}, \mathcal{E}, \Pi)$, we consider two pipeline elements $\rho_i, \rho_j \in \mathcal{V}, \rho_i \neq \rho_j$ *adjacent*, if there exists a directed event stream edge $(\rho_i, \rho_j) \in \mathcal{E}$ from ρ_i to ρ_j where $\rho_i \in \mathcal{V}\backslash S$ and $\rho_j \in \mathcal{V}\backslash A$. Then, the pipeline element ρ_i is called the *predecessor* and ρ_j is called the *successor*.

We model a pipeline element $\rho \in \mathcal{V}$ as a triple $\rho{=}(\mathcal{F}_\rho, \Delta_\rho, \mathcal{R}_\rho)$, where $\mathcal{F}_\rho$ is the *event processing logic*, Δ_ρ is a set of *static properties* that represent custom user inputs which allows citizen technologists to configure ρ when the pipeline is modeled. In addition, $\mathcal{R}_\rho{=}\mathcal{R}^d_\rho \cup \mathcal{R}^h_\rho \cup \mathcal{R}^s_\rho \cup \mathcal{R}^c_\rho$ is a set of *resource requirements* for data $\mathcal{R}^d_\rho$, hardware $\mathcal{R}^h_\rho$, software $\mathcal{R}^s_\rho$, and connectivity $\mathcal{R}^c_\rho$ requirements. Thereby, connectivity requirements can only be specified for adapters and sinks $\forall \rho \in \mathcal{V}\backslash P : |\mathcal{R}^c_\rho| {\geq} 0$ and are not permitted for processors $\forall \rho \in \mathcal{V}\backslash(A \cup S) : \mathcal{R}^c_\rho{=}\emptyset$. In general, while resource requirements are specified by the corresponding software engineer at pipeline element development time, static properties are bound to application-specific demands of the respective use case. Hence, citizen technologists who perform the actual pipeline modeling task also configure individual pipeline elements and thus specify relevant static properties at pipeline creation time.

Example. *Figure 6.2 shows the location monitoring pipeline $\mathcal{P}_{loc}=(\mathcal{V}_{loc}, \mathcal{E}_{loc}, \Pi_{loc})$ comprised of three pipeline elements $\mathcal{V}_{loc}=\{a_1, p_2, s_3\}$ and their event stream edges $\mathcal{E}_{loc}=\{(a_1, p_2), (p_2, s_3)\}$ as introduced in Section 3.1.2. In brief, the GPS location of the delivery robot is constantly monitored and a custom notification is sent to the fleet operators upon leaving the specified geofence denoting a valid operation area. At this point, pipeline policies are not provided until we introduce them in Section 7.3.1. The bottom part of Figure 6.2 cover related requirements $\mathcal{R}_\rho$ and static properties Δ_ρ for dedicated pipeline elements. While the processor and sink declare the same hardware requirements, there are differences in the other requirement types. For instance, the adapter a_1 defines a GPS connectivity requirement $\mathcal{R}_1^c=\{$MEDA:GPSMODULE$\}$ that requires connectivity to a GPS module indicated by a domain-specific vocabulary. Besides, the subsequent point-in-polygon processor states a data requirement $\mathcal{R}_2^d=\{$GEO:LAT,GEO:LONG$\}$ expecting the event stream to contain events whose payload include latitude and longitude event properties in the WGS84 reference format according to the W3C Basic Geo Vocabulary[3] prefixed with* GEO.

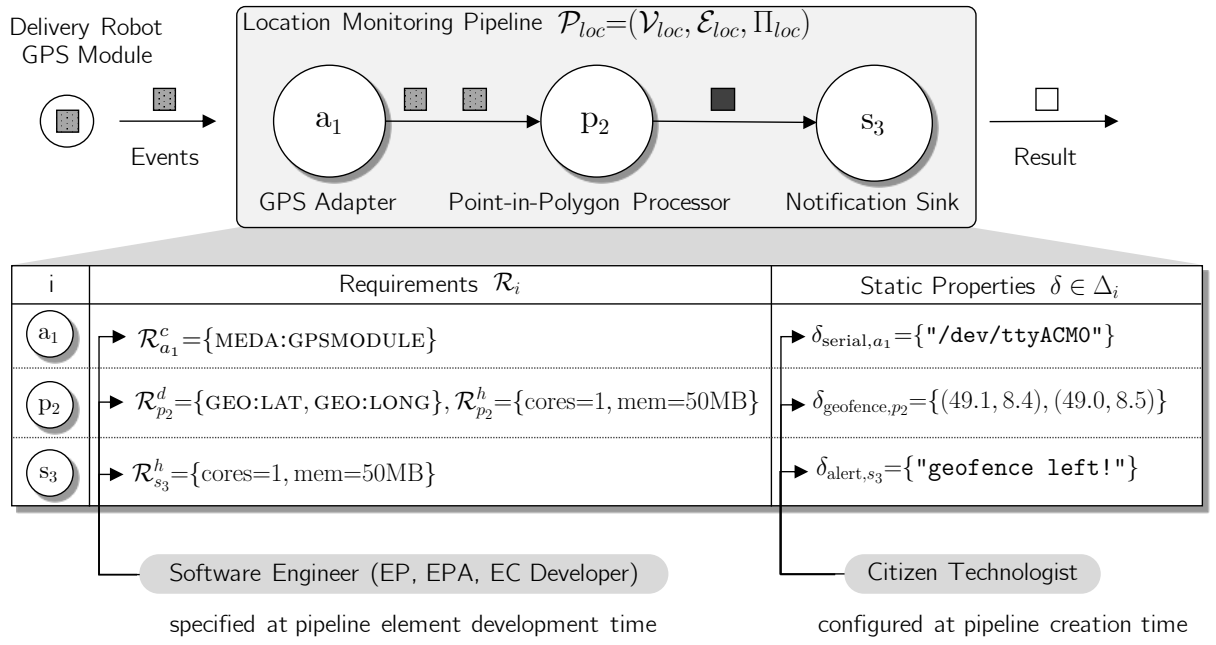


Figure 6.2 Running example: Pipeline element requirements and static properties

6.3 Fog Infrastructure Model

The fog computing infrastructure consists of set of nodes that possess varying resource capabilities with subject to the previously stated heterogeneity dimensions discussed in Section 6.1. Additional characteristics that differentiate individual nodes such as a node's logical location allow to address qualitative requirements, e.g., data sovereignty.

[3]https://www.w3.org/2003/01/geo/

Formal Infrastructure Model. We formally model a fog computing infrastructure as a graph $\mathcal{I}=(\mathcal{N},\mathcal{L})$. Thereby, $\mathcal{N}=\mathcal{N}_e \cup \mathcal{N}_f \cup \mathcal{N}_c$ is a finite set of nodes comprised of edge nodes $\mathcal{N}_e$, fog nodes $\mathcal{N}_f$, and cloud nodes $\mathcal{N}_c$. In addition, $\mathcal{L}$ is a set of network links connecting nodes. Due to the vast design space dictated by a myriad of applicational use cases, the graph structure for fog computing infrastructures greatly varies which leads to a manifold of different network topologies [Karagiannis and Schulte 2020]. For the sake of simplicity, we assume $\mathcal{I}$ to be a *complete graph*, i.e., $\mathcal{I}$ is an undirected graph in which every pair of distinct nodes is connected by a network link.

We consider a node $n \in \mathcal{N}$ to be a triple $n=(\mathcal{O}_n,\mathcal{C}_n,\mathcal{D}_n)$, where $\mathcal{O}_n=\mathcal{O}_n^h \cup \mathcal{O}_n^s \cup \mathcal{O}_n^c$ is a set of *resource offers* including hardware $\mathcal{O}_n^h$, software $\mathcal{O}_n^s$ and connectivity $\mathcal{O}_n^c$ resource offers. Thereby, connectivity resource offers allow to alleviate the connection to IoT devices and are only valid for edge and fog nodes, i.e., $|\mathcal{O}_n^c|\geq 0, \forall n \in \mathcal{N}\backslash\mathcal{N}_c$. We assume cloud nodes to not establish any direct connection to IoT devices due to limited bandwidth or privacy concerns, i.e., $\mathcal{O}_n^c=\emptyset, \forall n \in \mathcal{N}_c$. Further, $\mathcal{C}_n=\{c_{\text{NC}}, c_{\text{MB}}, c_{\mathcal{V}}\}$ denotes a set of *deployment containers*. There are two system management containers, namely a node controller container c_{NC} which is introduced later this chapter and a message broker container c_{MB} providing publish/subscribe messaging capability. In addition, there exists a pipeline element container $c_{\mathcal{V}}$ containing all pipeline elements for EP-, EPA-, and EC-services. $\mathcal{D}_n$ is a set of *node metadata* that include a unique node identifier, its IP address, the node type, as well as its logical affiliation, e.g., edge, fog, or cloud, besides the geographic location in the case of mobile nodes. Moreover, it contains domain-specific background knowledge that entails custom *node tags* $\mathcal{T}$ allowing to add arbitrary annotations. Adding meaningful annotations aid the node selection process for potentially larger node sets. Consequently, this allows to provide system-side support for citizen technologists prior to the pipeline deployment in order to select only eligible node targets.

We assume each node to run one instance of the node controller container. Further, we define that edge and fog nodes run one instance of the message broker container. In contrast, at the cloud layer, only a subset of all cloud nodes run a container for a distributed, shared message broker due to scalability and high-availability reasons.

6.4 Node Model

The central entity of interest for executing pipeline elements within the fog computing infrastructure is a *node*. Shown in Figure 6.3, we present a novel *node model* that builds the foundation for the so-called NodeDescription. This node description is exposed by each node controller across all layers of the hierarchical fog architecture and sits at the core of our holistic management approach. In essence, the node description allows to create awareness of node specificities. On the basis of explicitly modeling heterogeneity dimensions, this is vital to enable event-driven application management over geo-distributed fog infrastructures.

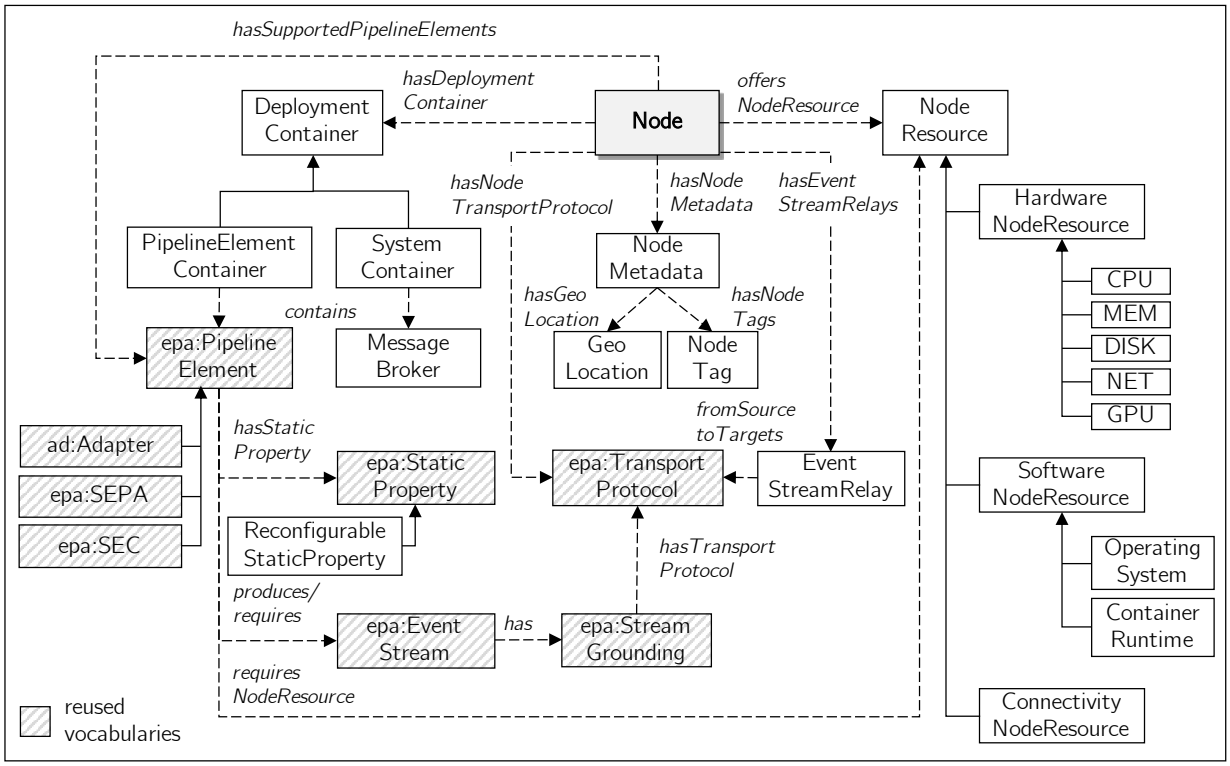


Figure 6.3 Node model: Overview

The node model reflects before stated formalisms besides reusing and integrating existing modeling efforts. This allows to create interoperability among systems by combining different ontologies for a new objective by means of extension, specialization or adaptation [Pinto and Martins 2000]. Accordingly, we reuse the following vocabularies:

- *Event Processing Application (EPA)*—The EPA vocabulary [Riemer 2016] provides a rich set of processing service descriptions for all types of *pipeline elements*. As our focus is on the management part of processing pipelines and not the pipeline modeling and authoring part, the EPA vocabulary is beneficial to our work. Yet, we enrich existing concepts to accommodate new properties in order to fulfill requirements constituting geo-distributed pipeline deployment, operation and adaptation. We use the prefix **epa** to identify the EPA vocabulary and refer to [Riemer 2016] for a more in-depth description of relevant concepts.
- *Adapter*—The adapter vocabulary [Zehnder et al. 2020] already extends parts of EPA in terms of a richer adapter model for EP-services. We use the prefix **ad** to identify the adapter vocabulary and refer to [Zehnder et al. 2020] for more details.
- *General vocabularies*—We use the prefix **rdf** for the RDF vocabulary, **rdfs** for the RDF Schema vocabulary, **qudt** for the QUDT vocabulary[4], **geo** for the Basic Geo vocabulary, **xsd** for the W3C XML Schema Definition Language.

[4]http://www.qudt.org/

In general, we can subdivide our node model in five elementary parts:

- NodeResource—Node resource offers, node resource requirements
- NodeMetadata—Additional node characteristics
- DeploymentContainer—Deployment definition for containerized services
- ReconfigurableStaticProperty—Concept to reconfigure EPAs at run-time
- EventStreamRelay—Concept for flexible event stream routing between nodes

Following, we give a detailed introduction of each part of the node model which is illustrated in Figure 6.3 and prefixed with **meda** to identify the developed vocabulary.

6.4.1 Node Resource

Node resources are essential to executing any kind of event-driven application regardless of a node's type and location in terms of edge, fog, or cloud node and thus considered as part of the NodeResource description. Since we are facing the challenge of managing processing pipelines in a geo-distributed environment spreading across edge, fog and cloud layers, these resources are considered highly heterogeneous. Apart from conventional hardware resources, there are also other resource types to take into account. Overall, we model three distinct concepts for resource offers to reflect a node's ability to host certain pipeline elements, namely: (1) HardwareNodeResource, (2) SoftwareNodeResource, and (3) ConnectivityNodeResource.

HardwareNodeResource subsume any type of resources that are essential for processing (compute), I/O (network) and storing data (storage). This typically includes resource types such as CPU, memory, disk, and network present in any modern compute platform that is inline with our notion of edge, fog, and cloud nodes presented in Section 2.3.3 and thus differ from IoT devices or microcontrollers. Consequently, our model specifies concepts for CPU, memory, disk, network and GPU as computational resources that are subclasses of the HardwareNodeResource concept as shown in Figure 6.3. In general, while there are multiple information that might be of interest in each subclass, we primarily focus on technical specifications, e.g., number of cores, total memory, or total disk space. Despite the benefits of containerized application deployments, a major drawback stems from the fact that container images[5] are platform-dependent, i.e., an image that is build for one processor architecture cannot be readily used and executed on another one. The fact that computational resources in fog computing are highly heterogeneous, also in terms of processor architectures, further exacerbates the situation. Therefore, the CPU concept defines the processor architecture which is an essential information in order to assess whether a node can safely run a dedicated containerized event-driven processing services.

[5] A container image is an immutable, static file that contains executable code libraries, dependencies, tools, and other files needed to run an application.

Example. *Listing 6.1 defines hardware resources for an NVIDIA Xavier AGX edge node with an ARM 64-bit CPU and GPU support.*

```
@prefix meda:   <https://managing-eda.example.de/vocabulary/v1/> .

<meda:xavier/cpu>
  a meda:CpuHardwareNodeResource ;
  meda:cores 8 ;
  meda:arch <meda:arm64> .

<meda:xavier/mem>
  a meda:MemHardwareNodeResource ;
  meda:memTotal 32 ;
  meda:unit qudt:GigaBYTE .

<meda:xavier/disk>
  a meda:DiskHardwareNodeResource ;
  meda:diskTotal 500 ;
  meda:unit qudt:GigaBYTE .

<meda:xavier/net>
  a meda:NetHardwareNodeResource ;
  meda:nic "eth0" ;
  meda:transferRate 1000 ;
  meda:unit qudt:MegaBIT-PER-SEC .

<meda:xavier/gpu>
  a meda:GpuHardwareNodeResource ;
  meda:cudaCores 512 ;
  meda:model "Volta" .
```

Listing 6.1 Hardware node resource definition: Example

The concept of SoftwareNodeResource has two subclasses, namely OperationSystem and ContainerRuntime. While the former holds information about the operating system of a node (e.g., "Ubuntu 18.04.5 LTS") operating system type (e.g., Linux), or kernel version (e.g., "5.10"), the latter describes specifications on the actual container runtime[6]. Although for simplicity, we consider DockerContainerRuntime or NvidiaContainerRuntime as specific incarnations of ContainerRuntime, our model is not exclusive to them. Especially, NvidiaContainerRuntime contains information about the present CUDA driver version that aids in preventing version incompatibilities and library conflicts when invoking ML-EPAs. As ML-EPAs contain pre-trained models, including neural networks, such models are oftentimes specifically dimensioned for the usage in constrained IoT environments and optimized for specific target GPU architectures to improve inferencing speeds, e.g., through quantization, pruning, or clustering to reduce model size and latency with no or minimal loss of accuracy.

[6]We assume that a container runtime compliant to the OCCI specifications is present on each node that serves as an abstraction for our approach to interact with low-level interfaces.

Example. *Listing 6.2 defines software resources for an NVIDIA Xavier AGX edge node with installed* `runc-nvidia` *container runtime.*

```
@prefix meda:   <https://managing-eda.example.de/vocabulary/v1/> .

<meda:xavier/operatingsystem>
  a meda:OperatingSystem ;
  meda:os "Ubuntu 18.04.5 LTS" ;
  meda:hasOsType <meda:linux> ;
  meda:kernelVersion "4.9.140-tegra" .

<meda:xavier/containerruntime>
  a meda:NvidiaContainerRuntime ;
  meda:apiVersion "1.41" ;
  meda:serverVersion "20.10.5" ;
  meda:cudaVersion: "11.0" ;
  meda:cudaDriverVersion: "450.36.06" .
```

Listing 6.2 Software node resource definition: Example

Lastly, the ConnectivityNodeResource concept allows to define specific knowledge in terms of accessibility to certain IoT devices such as sensors, actuators or machines, either *locally* (e.g., camera connected to an USB port on a Raspberry Pi, or GPS module attached to a serial port) or *remotely* (e.g., network-accessible OPC UA server). This allows to automatically retrieve relevant configuration properties necessary to alleviate the connection process to external event sources for adapters or sinks.

Example. *Listing 6.3 defines a connectivity resource to a locally accessible GPS module on the NVIDIA Xavier AGX edge node.*

```
@prefix meda:   <https://managing-eda.example.de/vocabulary/v1/> .

<meda:xavier/gpsmodule>
  a meda:ConnectivityNodeResource ;
  rdfs:label "GPS module" ;
  meda:hasDeviceType <meda:gpsmodule> ;
  meda:hasAccessType <meda:local> ;
  meda:connectionUrl "/dev/ttyACM0" .
```

Listing 6.3 Connectivity node resource definition: Example

To find eligible deployment target nodes, it is necessary for pipeline elements to express required node resources which we cover using the relation requiresNodeResource. This relation is introduced as an extension to the adapter [Zehnder et al. 2020] and EPA vocabulary [Riemer 2016]. Other relevant relations are briefly outlined as follows. For adapters (ad:Adapter), the protocol encompasses information on how to connect to the event source (ad:hasProtocol) while transformation rules define essential preprocessing steps (ad:hasRule). In terms of processors (epa:SEPA) and sinks (epa:SEC), there are event stream requirements on data-level (epa:requiresStream), information on supported

stream grounding (EPA:SUPPORTEDGROUNDING), output strategies (EPA:HASOUTPUTSTRATEGY, exclusive to processors) or static properties (EPA:HASSTATICPROPERTY). We clarify the notion of *eligibility* in Section 7.3.2 where we provide a detailed description of the respective validation procedure. Listings 6.4 and 6.5 illustrate connectivity requirements on the basis of an adapter and hardware requirements on the basis of a processor.

Example. *Listing 6.4 shows the definition for a GPS connectivity node resource requirement within the specific GPS adapter description from Figure 6.2.*

```
@prefix meda:   <https://managing-eda.example.de/vocabulary/v1/> .

<meda:gps>
  a ad:SpecificDataStreamAdapter ;
  rdfs:label "GPS adapter" ;
  ad:hasProtocol <ad:protocol/stream/gps> ;
  epa:hasFormat <epa:format/json> ;
  epa:hasDataStream <epa:gps/geostream1> ;
  ad:hasRule <ad:gps/transformationrule1> ;
  meda:requiresNodeResource <meda:gps/requirement1> .

<meda:gps/requirement1>
  a meda:ConnectivityNodeResource ;
  meda:hasDeviceType <meda:gpsmodule> .
```

Listing 6.4 Connectivity node resource requirement definition: Example

Example. *Listing 6.5 shows the definition for a hardware node resource requirement (1 core, 50 MB memory) within the point-in-polygon processor description from Figure 6.2.*

```
@prefix meda:   <https://managing-eda.example.de/vocabulary/v1/> .

<meda:pip>
  a epa:SEPA ;
  rdfs:label "Point-in-Polygon" ;
  rdfs:description "Processor performing point-in-polygon (pip) algorithm" ;
  epa:requiresStream <epa:geostream1> ;
  epa:supportedGrounding <epa:mqttgrounding> ;
  epa:hasOutputStrategy <epa:pip/outputstrategy> ;
  epa:hasStaticProperty <epa:pip/coordinate1>, <epa:pip/coordinate2> ;
  meda:requiresNodeResource <meda:pip/requirement1>, <meda:pip/requirement2> .

<meda:pip/requirement1>
  a meda:CpuHardwareNodeResource ;
  meda:cores 1 .

<meda:pip/requirement2>
  a meda:MemHardwareNodeResource ;
  meda:memory 50 ;
  meda:unit qudt:MegaBYTE .
```

Listing 6.5 Hardware node resource requirement definition: Example

6.4.2 Node Metadata

The concept of NodeMetadata complements the node resource description and provides additional metadata allowing to further differentiate or cluster nodes according to logical identifiers. Thereby, a node can be classified either as a *virtual* node (mostly the case for cloud nodes), or as a *physical* node (common for edge and fog nodes). Besides, nodes can be logically associated with a resource layer, i.e., edge, fog, or cloud. In order to establish communication to a node, the NodeMetadata concept contains related properties to access a node either by its IP address or via its fully qualified domain name. Moreover, the GeoLocation concept indicates the physical location of a node by reusing latitude and longitude properties from the W3C Basic Geo Vocabulary. However, referring to GPS coordinates to express a node's location is not practical in all cases. For instance, in the discussed Factory 4.0 use case, edge nodes are directly installed on the shop floor with fog nodes typically residing on the companies premises. This demands an alternative approach to reflect the node location while also allowing to specify relative location characteristics that are relevant from a deployment point of view. To this end, the NodeTag concept allows to add arbitrary information tags in the form of plain string literals that enclose additional domain-specific knowledge to express relative location information. Both concepts can jointly coexist such that mobile edge nodes can also have complementing node tags as shown in Listing 6.6.

Example. *Listing 6.6 depicts a node metadata resource for a mobile edge node (e.g., a delivery robot) that has a built-in NVIDIA Jetson Xavier AGX compute platform as a local edge node.*

```
@prefix meda: <https://managing-eda.example.de/vocabulary/v1/> .

<meda:xavier/nodemetadata>
  a meda:NodeMetadata ;
  meda:nodeControllerId "drobot-01.node-controller" ;
  meda:nodeControllerUrl "http://drobot01.example.de:7077" ;
  meda:nodeModel "Jetson-AGX" ;
  meda:hasNodeType <meda:physical> ;
  meda:hasAssociatedResourceLayer <meda:edge> ;
  meda:ipv4 "10.8.0.2" ;
  meda:fqdn "drobot01.example.de" ;
  meda:hasGeoLocation <meda:xavier/geolocation> ;
  meda:hasNodeTags <meda:xavier/nodetag> .

<meda:xavier/geolocation>
  a meda:GeoLocation ;
  geo:lat 49.012502 ;
  geo:long 8.426035 .

<meda:xavier/nodetag>
  a meda:NodeTag ;
  meda:tag "edge", "drobot", "ka-east" .
```

Listing 6.6 Node metadata definition: Example

6.4.3 Deployment Container

The container technology has many benefits and well-addresses the demands in resource-limited and heterogeneous fog infrastructures. Thus, the DeploymentContainer concept is an extensible and generic description of relevant properties for containerized services independent from the present container runtime. This includes the service identifier, the corresponding image tag and container name, as well as exposed ports. Besides, there are also properties that allow software engineers to define which processor architectures and operating system types are supported for the given image in order to verify the compatibility of an image on the underlying architecture. Additional configurations can be made by passing environment variables while labels allow to annotate dedicated containers, e.g., provide additional meta-information useful for filtering for specific instance types in the operation phase. In our case, we consider two types of DeploymentContainer, namely PipelineElementContainer and SystemContainer whereby each image wraps dedicated pipeline elements (i.e., ad:Adapter, epa:SEPA, epa:SEC) or a message broker technology respectively. In general, these deployment manifests are used by the node controller in the setup phase to automatically instantiate containers suited for the node. Afterwards, pipeline elements register themselves at the node controller. Thereby, only supported ones are accepted and linked with the hasSupportedPipelineElements relation (see Figure 6.3) given their resource requirements and the node's resource offers.

Example. *Listing 6.7 illustrates a pipeline element container description with an image suitable of running on "x86", "ARM32" and "ARM64" nodes and shows supported operating systems.*

```
@prefix meda: <https://managing-eda.example.de/vocabulary/v1/> .

<meda:deploymentcontainer/pe>
  a meda:PipelineElementContainer ;
  meda:serviceId "svc/org.example.pe" ;
  meda:image "exampleorg/pipeline-elements:0.1" ;
  meda:name "pipeline-element-container" ;
  meda:exposedPorts 8090 ;
  meda:hasSupportedArchs <meda:x86>, <meda:arm32>, <meda:arm64> ;
  meda:hasSupportedOs <meda:linux>, <meda:darwin>, <meda:windows> ;
  meda:hasContainerEnvVars <meda:containerenvvar/nodecontroller> ;
  meda:hasContainerlabels <meda:containerlabel/nodetype> .

<meda:containerenvvar/nodecontroller>
  a meda:ContainerEnvVar ;
  meda:envKey "NODE_CONTROLLER_URL" ;
  meda:envValue "http://node-controller:7077" .

<meda:containerlabel/nodetype>
  a meda:ContainerLabel ;
  meda:labelKey "org.example.container.type" ;
  meda:labelValue "pipeline-element" .
```

Listing 6.7 Deployment container definition: Example

6.4.4 Reconfigurable Static Property

The concept of EPA:STATICPROPERTY allows pipeline element developers to specify configurations which are declared during modeling-time, and in turn, are necessary to be configured by citizen technologists at design-time. Thus, static properties are user-defined and configured by citizen technologists during pipeline creation time [Riemer 2016]. Yet thereafter, these properties are considered immutable and cannot be modified to accommodate use case-specific changes to support pipeline evolution in terms of adaptation. A naive stop-the-world approach involves stopping the pipeline, updating relevant static properties of pipeline elements and redeploying it. However, not only does this oblige manual intervention by citizen technologists but also leads to an unwanted application unavailability which is infeasible in continuous monitoring scenarios in industrial settings. While the former is tolerable, the latter is not. To mitigate this, we introduce the concept of RECONFIGURABLESTATICPROPERTY which inherits general properties from EPA:STATICPROPERTY while adding an additional relation to denote whether a given static property is reconfigurable (EPA:ISRECONFIGURABLE), and thus can be altered through an adaptation mechanism at pipeline execution time which we detail in Section 8.3.1.

Example. *Listing 6.8 shows the definition of two reconfigurable static properties, here exemplified retrieve geofence coordinates as part of the point-in-polygon processor from Figure 6.2. The property* EPA:ISRECONFIGURABLE *is set to true for both coordinate static properties.*

```
@prefix meda:   <https://managing-eda.example.de/vocabulary/v1/> .

<epa:pip/coordinate1>
  a epa:FreeTextStaticProperty ;
  rdfs:label "Top-left coordinate" ;
  rdfs:description "Specify top-left coordinate (Example: 'lat,long')" ;
  epa:requiredDatatype xsd:string ;
  epa:isReconfigurable true .

<epa:pip/coordinate2>
  a epa:FreeTextStaticProperty ;
  rdfs:label "Bottom-right coordinate" ;
  rdfs:description "Specify bottom-right coordinate (Example: 'lat,long')" ;
  epa:requiredDatatype xsd:string ;
  epa:isReconfigurable true .
```

Listing 6.8 Reconfigurable static property definition: Example

6.4.5 Event Stream Relay

As a processing pipeline might span the cloud-edge continuum, with individual pipeline elements being executed on different deployment targets. This induces the necessity to employ an inter-node communication mechanism that allows to disseminate event

streams between adjacent pipeline elements executed on different nodes. Following the topic-based publish/subscribe pattern, pipeline elements produce (adapters, processors) or require (processors, sinks) a dedicated epa:EventStream which denotes a concept to declare various event stream properties and related stream specifications, chief among is the epa:StreamGrounding. The grounding allows pipeline elements to declare supported event transport protocols (epa:TransportProtocol). This is a generic concept for flexibly engaging various publish/subscribe message broker technologies, e.g., epa:MqttTransportProtocol or epa:KafkaTransportProtocol. We extend this by introducing the EventStreamRelay concept for *inter-node* communication that allows to map a source transport protocol to an arbitrary number of target transport protocols depending on the *cardinality of communication* while employing additional relay strategies which we detail in Section 7.3.3. The attentive reader will have noticed that we operate in a multi-broker and thus multi-transport protocol environment, as briefly stated in Section 6.3. We based this design decision on the dynamics in the fog where intermittent network failures are common. Having one message broker per node instance not only prevents excessive round trips in edge-only deployments, but also facilitates local event exchange for *intra-node* communication in cases where adjacent pipeline elements share the same deployment target. This type of publish/subscribe communication is addressed using event stream grounding matching strategies [Riemer 2016].

Example. *Listing 6.9 defines an event stream relay between the point-in-polygon processor (executed on an edge node) and the notification sink (executed on a cloud node) from Figure 6.2. The* epa:MqttTransportProtocol *denote a lightweight edge transport protocol used by the point-in-polygon processor while the notification sink leverages a shared cloud transport protocol* epa:KafkaTransportProtocol*. Note the topic equality.*

```
@prefix meda:   <https://managing-eda.example.de/vocabulary/v1/> .

<meda:eventrelay/edgenode>
 a meda:EventStreamRelay ;
 meda:nodeControllerUrl "http://drobot01.example.de:7077" ;
 meda:relayOption "buffer" ;
 meda:fromSource <epa:source/pointinpolygon> ;
 meda:toTargets <epa:target/notification> .

<epa:source/pointinpolygon>
 a epa:MqttTransportProtocol ;
 epa:brokerHostname "10.8.0.2" ;
 epa:port 1883 ;
 epa:topicName "org.example.point.in.polygon-1234" .

<epa:target/notification>
 a epa:KafkaTransportProtocol ;
 epa:brokerHostname "10.8.0.37" ;
 epa:port 9092 ;
 epa:topicName "org.example.point.in.polygon-1234" .
```

Listing 6.9 Event stream relay definition: Example

6.5 Architecture

The scale, heterogeneity, and dynamics of fog computing make the use and management of computing resources for event-driven applications quite complex, especially for non-technical citizen technologists. In the domain of cloud-native application orchestration, a common architectural pattern for designing distributed, container-based systems evolves around the *master-worker* paradigm with systems such as Borg [Verma et al. 2015], a predecessor of today's de facto container orchestrator Kubernetes[7]. In general, such system architectures have at least one coordinating master and multiple workers each of which are hosted on available compute nodes that offer their resources to be consumed by arbitrary application services provisioned as containerized microservices. Oftentimes, these core management modules are complemented by other infrastructure services, e.g., providing distributed storage, or overlay networking capabilities, which allow to further abstract technical infrastructure complexities in terms of placement, deployment, or life cycle management from the application development itself.

Within this work, we build on top of these principle ideas while assimilating the master-worker paradigm in our distributed system architecture for event-driven application management. This geo-distributed architecture outlines the overall structure for the holistic application management middleware. In contrast to general-purpose container orchestrators, we target the architectural design on the applicational and organizational needs (see Section 3.2) that arise in the context of geo-distributed pipeline administration in fog computing. Figure 6.4 gives an overview of the distributed system architecture for managing event-driven applications and depicts the initial node registration during the setup phase which we discuss in the following.

Master-Worker Paradigm. As mentioned, the general architectural design follows the well-known master-worker paradigm for distributed systems that results in different responsibility levels for each management entity. Hence, we consider two types of components, namely a central coordinator and manager as part of the *node management* acting as the master, as well as a node-local management service which we refer to as *node controller* representing a worker. Apart from related management tasks, the node controller holds an instance of the NodeDescription and exposes it via web-based standards which will be detailed later in this section. Despite a worker node's location and type (i.e., virtual or physical), we treat all nodes the same from a management standpoint and thus assume a single, logical resource pool of varying characteristics. Nevertheless, worker nodes are classified according to their associated resource layer within the fog architecture hierarchy as discussed in Section 6.4.2, which provides additional knowledge in the course of pipeline element deployment and adaptation to account for use case specific requirements. For instance, Figure 6.4 shows four nodes: one edge node $n_1 \in \mathcal{N}_e$, one fog node $n_2 \in \mathcal{N}_f$ and two additional cloud nodes $n_3, n_4 \in \mathcal{N}_c$. The provisioning

[7] https://kubernetes.io/

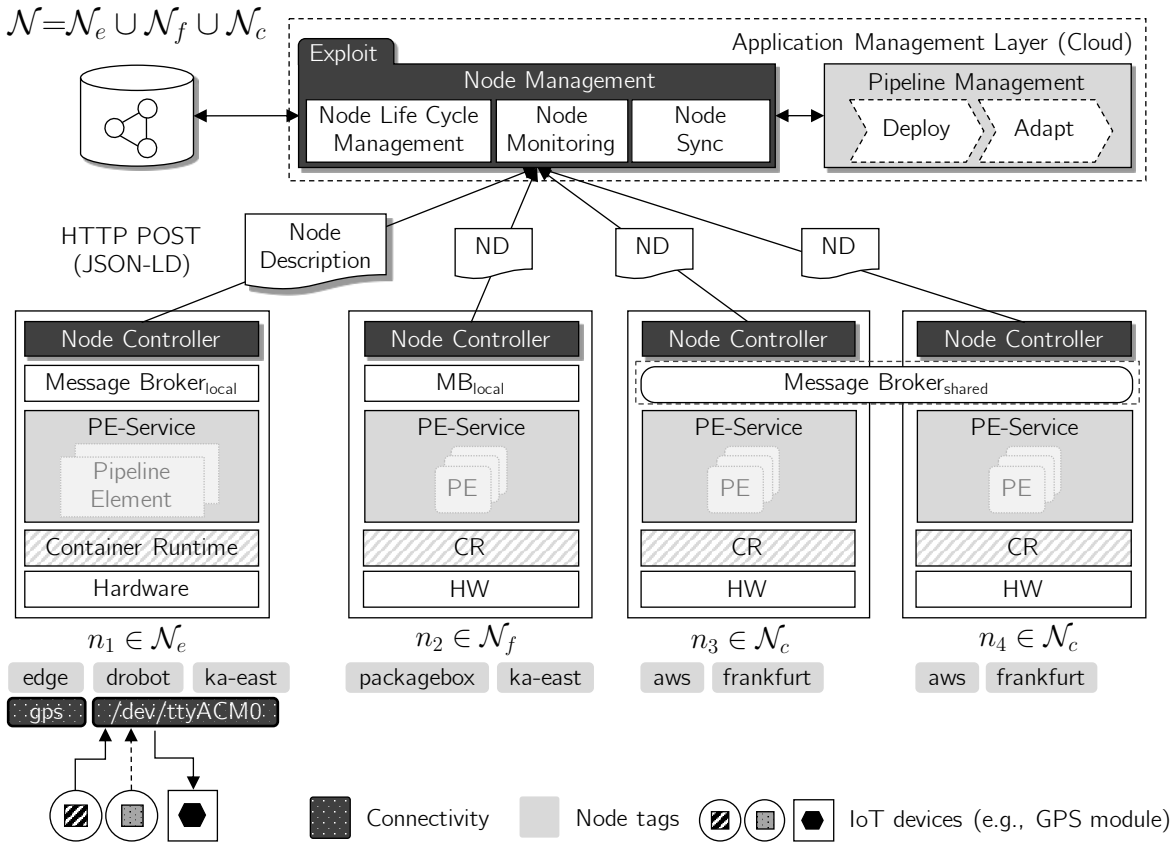


Figure 6.4 Geo-distributed architecture: Overview

and orchestration model within the architecture evolves around operating system-level containerization and thus assumes any container runtime compliant to the Open Cloud Computing Interface be present (see Section 6.1). For the sake of simplicity, Figure 6.4 only shows worker nodes being equipped with a container runtime, yet we also assume central application management components to be flexibly provisioned via containers as this also eases the overall system setup in the cloud. Consequently, apart from the management logic entailed in the node controller container, each node hosts a dedicated pipeline element container that contains a pipeline element service employing EP-, EPA, EC event processing logic, in addition to a message broker container. As stated in the formal infrastructure model, cloud nodes are exceptional as they use a scalable and shared message broker that typically runs multiple replica on cloud nodes as indicated for node n_3 and node n_4.

Node Management—Master. The node management is the central interface to the pipeline management and all worker node affairs. Thereby, both the node management and pipeline management are part of the central application management which runs in the cloud. We made this decision based on the general graphical flow-based development approach, where processing pipelines are centrally modeled and thus are

already present to be prepared for geo-distribution. The pipeline management receives processing pipeline definitions $\mathcal{P}=(\mathcal{V},\mathcal{E},\Pi)$ and prepares individual pipeline elements for deployment while issued adaptation requests in terms of reconfiguration, migration or offloading, are processed within adaptation modules. Both the deployment and adaptation aspects are inherent fog-related challenges and at the core of our conceptual considerations. This presents key differentiations to existing work on pipeline management [Riemer 2016]. Concepts for geo-distributed pipeline management with regard to deployment, operation and adaptation will be described in the following Chapters 7 and 8 in detail.

Besides, the node management deals with various core tasks on a central level, including the following: (1) node life cycle management from join, re-join, update and delete while providing necessary handlers to deal with create, read, update, and delete operation with the database, (2) cluster-wide monitoring to assess both liveness and current resource utilization once a node has registered and placed its resources at disposal and (3) node synchronization actions, including deactivating nodes for maintenance, adding or updating node tags and connectivity information when an edge node is equipped with an additional sensor, or provide new DeploymentContainer descriptions for newly developed pipeline element containers to ensure updates.

Node Controller—Worker. In contrast, the node controller resides on every node and is a local management service responsible for the following tasks: (1) generating the NodeDescription upon startup and registration at the central node management, (2) managing container deployments based on the DeploymentContainer concept agnostic to the underlying compute platform, (3) mediating deployment and adaptation requests between central pipeline management and the dedicated pipeline element service and thus accompany the whole pipeline element life cycle, (4) executing event stream relay requests according to centrally generated EventStreamRelay descriptions for flexible inter-node communication while incorporating resiliency mechanisms to deal with temporary network interrupts and (5) observing the resource utilization of a node and leveraging methods to self-reliantly decide when to offload certain pipeline elements.

Hereafter, we briefly outline the node controller registration where individual node controller instances register themselves and their NodeDescription at the central node management. This serves as a prerequisite for the subsequent pipeline deployment in Chapter 7 and pipeline adaptation in Chapter 8 where we give an in-depth description on the actual management part both on the central application management layer as well as on the node level for the node controller.

Node Controller Registration. Each node hosts a dedicated node controller service which itself is provisioned as a system container. The node controller generates the respective NodeDescription of its underlying node upon startup according to the node

model described in Section 6.4. Here, the NodeDescription holds generic deployment container descriptions for the pipeline element container as well as message broker container that may be auto-deployed after a successful node registration to ease the setup process. These generic deployment descriptions are independent of the underlying container runtime and contain relevant information on what types of processor architectures, i.e., x86, ARM32, ARM64, are supported by the designated container images, such that the node controller is capable of automatically selecting the correct image type suited for the underlying infrastructure. As previously stated, this is necessary as container images are platform-dependent and are thus purpose-built to target specific processor architectures, such that images build for x86 platforms are not capable to run on ARM platforms without any cross-compilation. Consequently, we assume that both pipeline element container images and their respective DeploymentContainer description are provided a priori by software engineers who develop dedicated event-driven processing services as mentioned in Section 3.2. For provisioning and interacting with running containers, the node controller implements interfaces to interact with the corresponding container runtime environment on the system and delegates low-level tasks such as container instantiation to it.

Moreover, system information regarding hardware and software resources that are part of the NodeDescription are automatically gathered. Additional metadata and connectivity information can either be passed prior to registration by using configuration parameters or ex post as part of the tool contributions to ensure extensibility. To allow a node to be considered for executing pipeline elements, the NodeDescription needs to be made available and registered at the central node management. After the RDF model for the NodeDescription is instantiated, the description is serialized in JSON-LD[8] and sent to the central node management by the node controller in order to register its resource description alongside other relevant metadata. Subsequently, the node description is persistently stored inside a designated database.

Example. *Let us consider a potential fog infrastructure for the smart urban logistics scenario as depicted in Figure 6.4. In this matter, the autonomous delivery robot is a mobile edge node that contains of a built-in NVIDIA Xavier AGX. Moreover, the delivery robot operates in the east urban district in the city of Karlsruhe. A fog node is installed and available at a nearby package box in the same urban district, complemented by two cloud nodes. Thus, each node is associated with certain clarifying node tags, namely for the edge node* n_1 *"edge", "drobot", "ka-east", for the fog node* n_2 *"packagebox", and "ka-east" while both cloud nodes* n_3, n_4 *are tagged with "aws" and "frankfurt". In addition, the edge node on the delivery robot has a special connectivity offer that allows it to directly connect to a GPS module indicated by the "gps" connectivity label which is accessible over a local serial port, here "/dev/ttyACM0".*

[8]JSON-LD is a lightweight Linked Data format and allows to encode RDF data in a more human-friendly and readable manner. In contrast to other representations such as Turtle or RDF/XML, JSON-LD is based on the well-known JSON format, is more concise and thus better suited for constrained environments such as within fog computing and the IIoT due to a reduced message overhead.

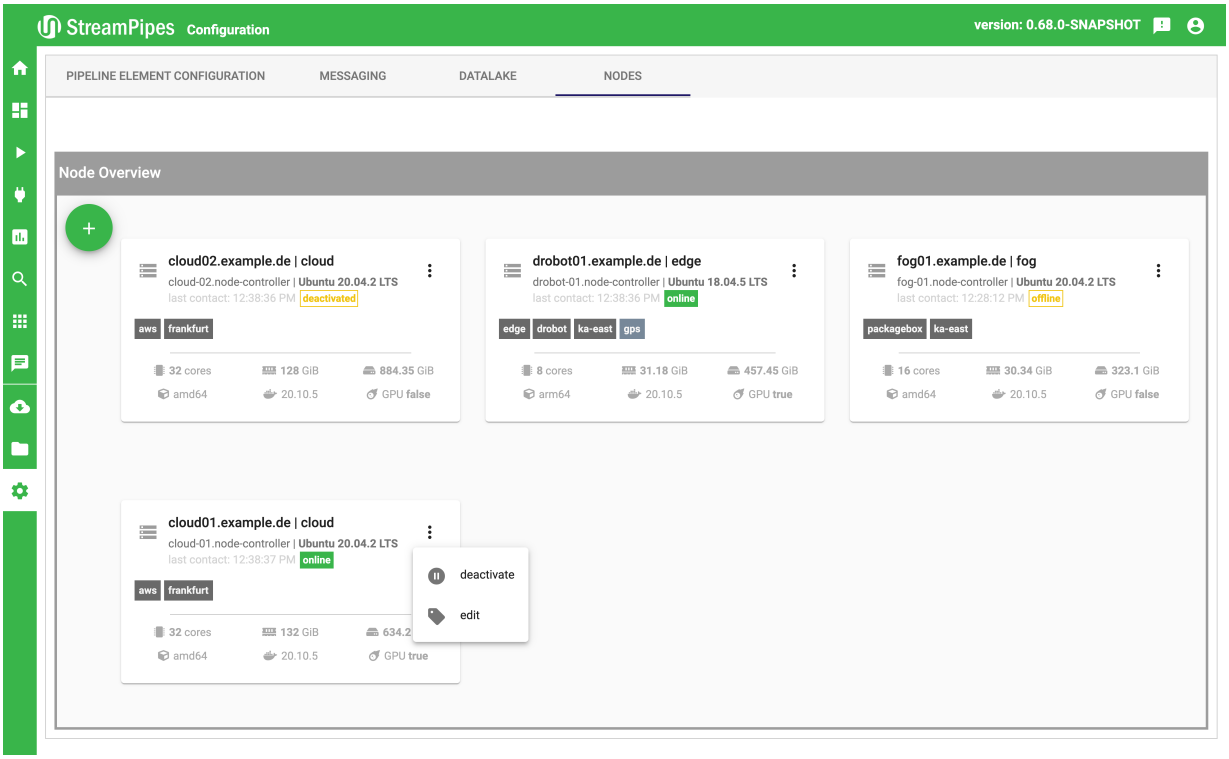


Figure 6.5 Node overview, monitor and management: Tool support

6.6 Tools

The concepts described in this chapter are transferred and integrated into the Apache StreamPipes project. It should be noted that we take a user-centric view when we talk about the tool support in question thereby targeting our citizen technologist role, yet with complete backend support. Figure 6.5 depicts the node overview of four registered nodes based on the previously shown architecture overview. Citizen technologists are presented with a node overview to quickly assess potential deployment targets, including node resources, node metadata (i.e., node tags, connectivity labels) and live node conditions, e.g., "online". The latter is the result of a node liveness check which is performed by the central node monitoring component at regular intervals. If the liveness check fails for several consecutive times, nodes are considered "offline" until they re-join. Individual node tags and connectivity information can be added or altered at run-time. Therefore, users can update the node description using a setting dialog where a node can also be deactivated, e.g. to perform maintenance tasks. In the latter case, nodes are marked as "deactivated". Moreover, nodes can be added at run-time to extend the fog infrastructure. Therefore, the web interface provides a designated dialog to configure relevant settings. Finally, a deployment command is provided to manually start the node controller container which then registers itself and auto-deploys a set of containerized services according to entailed DeploymentContainer descriptions.

6.7 Summary

In this chapter, we introduced relevant concepts for event-driven applications to exploit heterogeneous computational resources in fog infrastructures in view of Research Question 1. The first contribution is a generic and extensible node model that allows to capture node-specific resource characteristics that exceed conventional hardware specifications and incorporate both software and connectivity-related aspects alongside additional node metadata to address the needs emerging from three heterogeneity dimensions in fog computing. Moreover, our model reuses and extends existing vocabularies in order to alleviate both the deployment process and the provide a foundation for later adaptation schemes as a result of arising applicational needs to account for dynamics in the context of IIoT. The second contribution is a geo-distributed architecture for the management of event-driven application deployments over heterogeneous computational resources. Inspired by well-established designs for distributed systems, the architectural design follows the master-worker paradigm and proposes a two-level management approach with a central node management complemented by multiple worker nodes equipped with a dedicated node controller service. Consequently, worker nodes can be added at run-time allowing the system to grow with use case-specific demands. The proposed node model sits at the core of the architecture and allows to create awareness of relevant resource characteristics which be used for the declaration of node resource requirements on the application side. Both model and architecture provide the required foundation for subsequent investigations with regard to pipeline deployment discussed in Chapter 7 as well as pipeline adaptation discussed in Chapter 8 as part of the operation phase. Lastly, we presented the central node overview and management interface shown to citizen technologists which is based on the integration efforts of the node model into the knowledge base of Apache StreamPipes to provide necessary tool support.

7

Pipeline Deployment

In the last chapter, we laid fundamental concepts and introduced an architecture including main buildings blocks as a basis for a two-level application management approach for event-driven applications in fog infrastructures. In this chapter, we present concepts and methods with regard to Research Question 2 and elaborate how to *deploy* and *operate* event-driven applications in heterogeneous fog infrastructures. After an initial *walkthrough* in Section 7.1, we introduce pipeline element *life cycle* stages in Section 7.2. Afterwards, we elaborate how pipeline elements are *geo-distributed* along the cloud-edge continuum in Section 7.3 which involves both central application management and node-local management aspects. Next, we present *tool support* of our conceptual work in Section 7.4 before we summarize the chapter in Section 7.5.

7.1 Walkthrough

After the initial setup phase involving node controller registration, the subsequent phase deals with the application-side whose focus is on management-related objectives for processing pipelines and their comprised pipeline elements. This involves a clear view of the life cycle for processing pipelines and consequently for individual pipeline elements which are subject to our geo-distributed deployment approach. As a prerequisite for executing pipeline elements, pipeline element containers are to be deployed by the node controller according to their deployment container descriptions while ensuring compatibility to the underlying infrastructure. Once the pipeline element containers are deployed, the individual pipeline elements register themselves at the node controller which proxies a service registration request to the central application management. As we will discuss in the following in more detail, this step employs an initial eager requirements-resource validation in order to filter unsupported pipeline element requirements which cannot be fulfilled by the given node, e.g., because the required hardware resources exceed the available ones. As a result, the node description is updated to only contain supported pipeline elements. This information is further leveraged to compare node resource offers against node resources requirements issued by pipeline elements during the pipeline configuration process prior to the actual deployment and distribution. Resulting eligible nodes are exposed to citizen technologists at pipeline authoring completion and used for

manual node selection. Once citizen technologists submit a pipeline deployment request, the corresponding pipeline graph alongside deployment options and operation policies are sent to the pipeline management for geo-distribution. In brief, this involves the generation of event stream relays for adjacent pipeline elements on different deployment targets and distributing invocation requests for pipeline elements and event stream relays to dedicated node controllers of chosen nodes. The node controller in turn further redistributes pipeline element invocation requests to the colocated pipeline element service to instantiate individual pipeline elements while initializing necessary event stream relays for inter-node communication. In the remainder of this chapter, we clarify the pipeline element cycle and further detail the previously introduced architecture from Section 6.5 with regard to central pipeline management concepts and elaborate on the essential role of the node controller.

7.2 Life Cycle

In order for citizen technologists to use pipeline elements for modeling processing pipelines in the first place, respective containerized pipeline element services need to be provisioned on geo-distributed nodes within the fog infrastructure. In general, we differentiate between three different phases within the pipeline element life cycle:

- *Setup phase*—The setup phase is tied to the node controller registration discussed in Section 6.5. The node controller initiates the deployment of the pipeline element container which starts an eager validation and registration procedure where prefilter criteria are applied. Consequently, only pipeline elements that are supported by the given node are registered and made available.
- *Operation phase*—The operation phase spans from the (1) pipeline authoring, over (2) pipeline deployment, to (3) pipeline adaptation. Hereby, pipeline authoring concerns discussed in [Riemer 2016] are extended to allow citizen technologists to configure and specify preference-based deployment options and operation policies. These specifications are considered by the central application management layer when preparing the geo-distributed pipeline deployment and are further detailed in Section 7.3. After this step, pipeline elements are invoked and consequently running. Pipeline adaptation denotes run-time evolution aspects of individual pipeline elements, e.g., reconfiguration or relocation, which is the focus of Chapter 8.
- *Maintenance phase*—Lastly, started pipeline element containers are removed in the maintenance phase, e.g., to update the DeploymentContainer description followed by a container restart. This phase is out of scope in this work.

Figure 7.1 summarizes the pipeline element life cycle and presents significant stages that each pipeline element surpasses. Hereafter, our main concern is the *operation phase*. Yet, we clarify the pipeline element registration step during the setup phase in the following, as this is a prerequisite for subsequent deployment efforts.

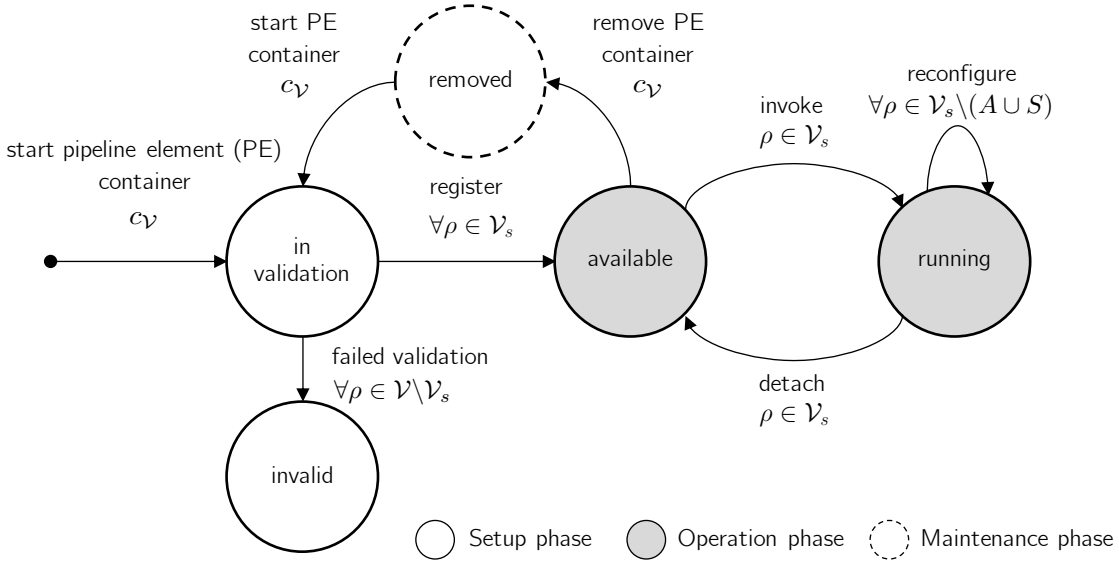


Figure 7.1 Pipeline element life cycle

Setup Phase—Eager Validation and Registration. As discussed in Section 6.5, each node controller holds a node description including generic definitions for deployment containers which allow to provision the underlying node with containerized services such as the pipeline element container. After the node controller starts the corresponding pipeline element container $c_\mathcal{V}$, the contained pipeline element service is initialized. The service tries to register its description including all wrapped pipeline elements $\mathcal{V}$ at the node controller which ultimately forwards service registration requests to the central node management to be stored. To forward a service description which only contains supported pipeline elements $\mathcal{V}_s$, each pipeline element is validated according to its specified resource requirements $\mathcal{R}_\rho$ and compared to the present resource offers $\mathcal{O}_n$ of the underlying node. Therefore, we introduce a set of helper functions.

$$\begin{aligned} \textsc{req}(\rho) &= \mathcal{R}_\rho & \rho &\in \mathcal{V} \\ \textsc{off}(n) &= \mathcal{O}_n & n &\in \mathcal{N} \end{aligned}$$

While the function $\textsc{req}(\rho)$ allows to extract all resource requirements from a pipeline element, the function $\textsc{off}(n)$ extracts all resource offers from a node. Therefore, given a pipeline element $\rho \in \mathcal{V}$ and a node $n \in \mathcal{N}$, the boolean function $\textsc{isSupported}(\mathcal{R}_\rho, \mathcal{O}_n)$ defines whether a pipeline element is supported by this node.

$$\begin{aligned} \rho \in \mathcal{V}, n \in \mathcal{N}, \textsc{req}(\rho) = \mathcal{R}_\rho, \textsc{off}(n) = \mathcal{O}_n : \Big(\textsc{isSupported}(\mathcal{R}_\rho, \mathcal{O}_n) \iff \\ \forall r \in \mathcal{R}_\rho^t \subseteq \mathcal{R}_\rho, \exists o \in \mathcal{O}_n^t \subseteq \mathcal{O}_n : (\textsc{isSatisfied}_t(r, o)) \Big) \end{aligned}$$

Consequently, pipeline elements are only considered supported if all their requirements are fulfilled by the node resource offers. Here, $\textsc{isSatisfied}_t$ represents individual boolean functions per resource type t, i.e., hardware, software, and connectivity type, that take

a resource requirement and a resource offer as inputs and validate them according to a specific validation scheme. For volatile hardware resources, e.g., memory, the eager validation procedure performs its validation based on the theoretical maximum value for a required resource type, e.g., total memory. Here, the main objective is to early identify incompatibilities prior to the pipeline authoring. Consider a pipeline element that requires GPU support such as a ML-EPA for inferencing or scoring—consequently, only nodes equipped with a GPU are eligible and thus capable of executing such a pipeline element. Although the validation concept is explicitly kept generic and considers all resource levels equally, in praxis connectivity and software-related resource offers change infrequently, such that mainly hardware resources are decisive. Algorithm 1 illustrates how the function isSupported is leveraged within the node controller to eagerly validate pipeline element candidates while only supported pipeline elements $\mathcal{V}_s$ are subsequently registered and made available on the central application management layer. At the same time, the node description is updated accordingly to hold references to supported pipeline elements to prevent a reevaluation in case of a pipeline element service restart, e.g., in the presence of a container or node failure. All other pipeline elements are considered invalid and are thus not supported.

Algorithm 1 Eager validation upon pipeline element service registration

Input: node description n, set of all pipeline elements $\mathcal{V}$
Output: set of supported pipeline elements $\mathcal{V}_s$

1: $\mathcal{V}_s \leftarrow \emptyset$
2: $\mathcal{O}_n \leftarrow \text{OFF}(n)$
3: **for each** $\rho \in \mathcal{V}$ **do**
4: $\quad \mathcal{R}_\rho \leftarrow \text{REQ}(\rho)$
5: $\quad$ **if** $\mathcal{R}_\rho \neq \emptyset$ **then**
6: $\quad\quad$ **if** isSupported$(\mathcal{R}_\rho, \mathcal{O}_n)$ **then** ▷ requirements validation
7: $\quad\quad\quad \mathcal{V}_s \leftarrow \mathcal{V}_s \cup \rho$
8: $n \leftarrow$ setSupportedPipelineElements$(\mathcal{V}_s)$ ▷ update node description
9: **return** $\mathcal{V}_s$

7.3 Geo-Distribution

As an outcome of the setup phase, the registration of node descriptions allows to gain transparency and cater the awareness for present node resources. Alongside the eager validation applied to shortlist pipeline elements to only node-supported ones, this builds the foundation of the operation phase. In this respect, administrating the geo-distribution of pipelines marks a critical point within the operation phase and requires to be dealt with at both the central application management level and the node level. In the following, we

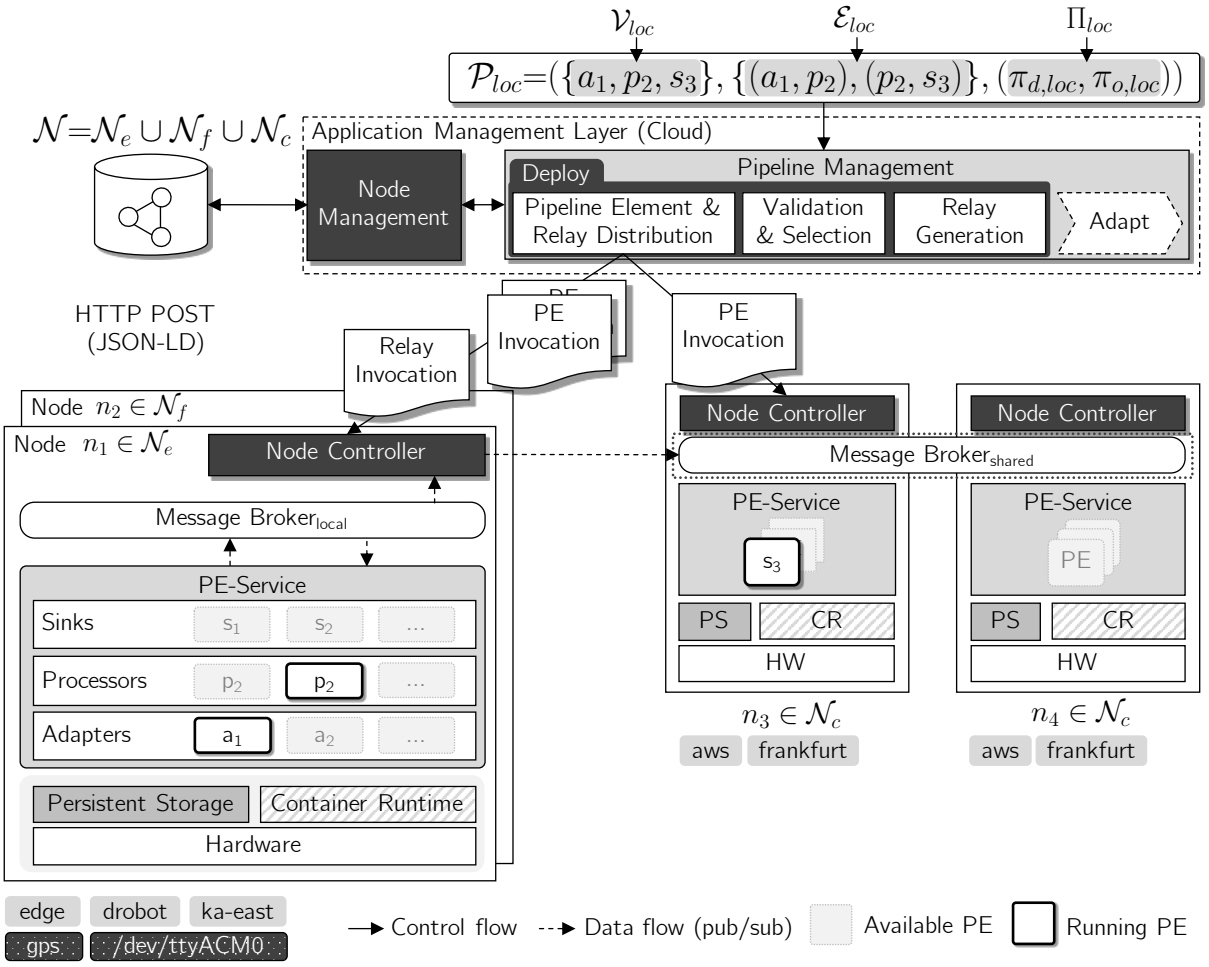


Figure 7.2 Geo-distributed pipeline deployment: Architecture overview

elaborate on relevant management concepts employed in the overall system design and provide an in-depth description of each constituent part along the deployment process. Therefore, Figure 7.2 further details our previously introduced architecture overview with regard to geo-distributed pipeline deployment.

7.3.1 Deployment Options and Operation Policies

In Section 6.2, we introduced our formal pipeline application model $\mathcal{P}=(\mathcal{V}, \mathcal{E}, \Pi)$. Aside from comprised pipeline elements $\mathcal{V}$ and their directed event stream edges $\mathcal{E}$, we also consider additional preference-based configurations Π. Preference-based configurations $\Pi=(\pi_d, \pi_o)$ are defined by citizen technologists at the end of the pipeline authoring process. So, apart from the actual analytical objective of a given pipeline, citizen technologists are provided with additional settings to flexibly configure *deployment options* π_d and *operation policies* π_o for processing pipelines according to use case-specific requirements.

This step is essential as it not only prepares pipelines for deployment, but also sets run-time instructions for the remaining operation phase. In view of the preparation, this includes the validation and selection of eligible nodes, methods for flexible event stream management and the final geo-distribution of the pipeline to dispersed nodes in the fog infrastructure which we detail in later.

In general, we differentiate between two types of *preference-based* configurations:

- *Deployment options*—Deployment options describe an arbitrary set of mapping strategies that allow to assign individual pipeline elements to eligible deployment target nodes.
- *Operation policies*—Operation policies denote run-time behavior of pipelines and their surrounding application management.

Thereby, *deployment options* fall in the general research area of application placement or scheduling. In the context of fog computing, this is oftentimes referred to as the fog application assignment problem which has been broadly studied also in terms of multi-component application placements [Wang et al. 2017; Bahreini and Grosu 2017]. As our main focus is on event-driven application management in fog environments, formulating a fog application assignment problem is out of scope for this work. In contrast, we assume an a priori assignment as a result of user-made assignment decisions. To this extent, citizen technologists are supported by the system and only presented valid and eligible nodes deployment targets per pipeline element. We refer to this as the *custom* deployment option. The custom deployment option further benefits from additional node metadata, in particular from node tags. Node tags allow to further filter nodes according to its semantics, in general representing any form of domain-specific meaning such the logical location of a node.

In contrast, *operation policies* target relevant preference-based configurations that allow citizen technologists to pass specific run-time instructions. These instructions are additionally added to the pipeline description and taken into consideration by the application management. In particular, this targets the node controller due to its responsibility for managing pipeline elements and associated tasks throughout the operation phase. In contrast to deployment options where node assignments are applied per pipeline element, operation policies general reflect *global pipeline policies* and are always applied on pipeline-level, i.e., they are equally effective for all pipeline elements. In this work, we focus on two types of operation policies with respect to (1) event stream relay management, and (2) preemption. Operation policies for *event stream relays* π_o^r provide global operation guidelines for the event stream relay mechanism employed in the node controller. Concretely, we are concerned with the reactive behavior of event dissemination in case of intermittent network outages. Thus, citizen technologists can choose between different resiliency options for event stream relays. The relay operation policy is generic, like all other policies and deployment options. Yet, in this work we exemplify two concrete options, namely *purge* or *buffer*. While the naive purging option simply drops events

between adjacent pipeline elements during a network outage, the buffer option implicates a specific buffering strategy which allows to hold and persist output events from the preceding pipeline elements. Further, operation policies for *preemption* π_o^p provide global operation directives regarding a policy-driven offloading approach employed in the node controller. This is part of the run-time adaptation of processing pipelines which we detail in Section 8.3.3. In brief, citizen technologists may enable preemption and set a priority class for a given pipeline. This information is obtained and used by the node controller to evict lower-prioritized pipeline elements in the event of undesired context changes during the execution, e.g., when a node is over-utilized. Similar to deployment options, operation policies are extensible and allow to be further expanded in dependence of additional run-time management tasks.

Example. *Again, let us consider a four node fog infrastructure for the smart urban logistics scenario and recall the location monitoring pipeline* $\mathcal{P}_{loc}$ *described in Figure 6.2. In contrast to our initial settings, the citizen technologist now specifies additional deployment options and operation policies. Figure 7.2 depicts the deployment request for* $\mathcal{P}_{loc}$ *with additional preference-based configurations* $\Pi_{loc}{=}(\pi_{d,loc}, \pi_{o,loc})$*. Therefore, after choosing the custom deployment options* $\pi_{d,loc}{=}(\langle \textit{selected}{=}custom\rangle, \langle \textit{nodeTags}{=}\mathcal{T}_{loc}\rangle)$ *including the node tags* $\mathcal{T}_{loc}{=}\{"aws", "edge"\}$*, the citizen technologists is presented with a validated list of eligible deployment target nodes. Moreover, the overall operation policies* $\pi_{o,loc}{=}(\pi^r_{o,loc}, \pi^p_{o,loc})$ *comprise configurations for the event stream relay management which is set to buffer* $\pi^r_{o,loc}{=}(\langle \textit{buffer}{=}true\rangle)$*. Additionally, preemption is enabled with a "high" priority class* $\pi^p_{o,loc}{=}(\langle \textit{enabled}{=}true\rangle, \langle \textit{prio}{=}high\rangle)$*.*

7.3.2 Validation and Selection

Previously, we assumed citizen technologists to be presented with a list of pre-filtered deployment target nodes which we refer to as *eligible* nodes. In this section, we clarify the notion of *eligibility* in the given context. After that, we describe how to find eligible nodes as potential deployment targets which are provided to citizen technologists in the node selection process.

To assist citizen technologists when assigning individual pipeline elements to nodes, two tasks are required by the system: First, it must ensure the basic compatibility of a given pipeline element with respect to present nodes. Second, it must assess and validate the actual available node resource offers to reflect run-time changes in resource usage. Moreover, user-selected node tags introduce additional criteria to shortlist node candidates. This needs to be taken into account when finding eligible deployment target nodes as they present meaningful domain-specific knowledge to citizen technologists. In essence, given an individual pipeline element the notion of *eligibility* refers to nodes that (1) are online and healthy at the time, (2) support to execute this pipeline element as a result of the eager validation (see Section 7.2), (3) match at least one of the selected node tags (if any) and (4) satisfy all resource requirements.

This demands an iterative validation approach that involves coarse to fine-grained node filtering to find eligible nodes for a given pipeline element. Algorithm 2 shows our validation approach which we explain in the following.

Algorithm 2 Coarse to fine-grained validation procedure for eligible nodes filtering

Input: a pipeline element $\rho \in \mathcal{V}$ in pipeline $\mathcal{P}$, set of selected node tags $\mathcal{T}$
Output: set of eligible nodes $\mathcal{N}_{el,\rho}$ for pipeline element ρ

```
1: N_el,ρ ← ∅
2: N_o ← getOnlineAndHealthyNodes                          ▷ node health monitor
3: if N_o ≠ ∅ then
4:     for each n ∈ N_o do
5:         V_s ← getSupportedPipelineElements(n)           ▷ eager validation
6:         if containsElement(V_s, ρ) then
7:             N_el,ρ ← N_el,ρ ∪ n                         ▷ pre-filter step num. 1
8:     if N_el,ρ ≠ ∅ then
9:         if T ≠ ∅ then
10:            N_el,m ← ∅
11:            for each n ∈ N_el,ρ do
12:                T_n ← getNodeTags(n)
13:                if anyTagMatch(T_n, T) then             ▷ node tag validation
14:                    N_el,m ← N_el,m ∪ n
15:            N_el,ρ ← N_el,ρ ∩ N_el,m                    ▷ filter step num. 2
16:        R_ρ ← req(ρ)
17:        N_el,s ← ∅
18:        for each n ∈ N_el,ρ do
19:            O'_n ← off(n)                               ▷ node resource monitor
20:            if isSupported(R_ρ, O'_n) then              ▷ requirements validation
21:                N_el,s ← N_el,s ∪ n
22:        N_el,ρ ← N_el,ρ ∩ N_el,s                        ▷ filter step num. 3
   return N_el,ρ
```

The general idea behind the approach is to iteratively reduce the set of nodes which ultimately leads to a subset of eligible nodes $\mathcal{N}_{el,\rho}$ for a given pipeline element ρ and selected node tags $\mathcal{T}$. Hereafter, when referring to a node we oftentimes imply the NodeDescription whose concepts were discussed in the previous chapter. First, all online and healthy nodes $\mathcal{N}_o$ are collected from the node health monitor—a central service within the node monitoring component and part of the node management that constantly performance health checks and liveness probes on registered node controllers as discussed in Section 6.5. Hence, if a registered node is considered "offline" or "unhealthy", either because it is temporarily unavailable, e.g., due network outages in the edge and fog layer, or due a software failure, it is excluded from this initial set of nodes. Next, each

available node contains a set of supported pipeline elements $\mathcal{V}_s$ as a result of the eager validation which is now leveraged in order to pre-filter for nodes that support the given pipeline element ρ (Lines 4 to 7). Thereafter, the subset of temporary eligible nodes can be validated with regard to potential user-selected node tags $\mathcal{T}$(Lines 9 to 15). Thereby, a node must have at least one of the selected node tags to be further considered as eligible. If citizen technologists do not select any node tags, this step is skipped. Lastly, the resource requirements $\mathcal{R}_\rho$ for the given pipeline element are extracted and compared against the current resource offers $\mathcal{O}'_n$ of each of the remaining temporary eligible nodes using the introduced isSupported($\mathcal{R}_\rho, \mathcal{O}'_n$) function (Lines 16 to 22). It is worth noting that the current hardware resource offers typically differ from the initial ones which are used in the setup phase. This is due to the fact that all pipeline elements on a node operate in a shared resource pool. To this end, available hardware resources are continuously collected by the node resource monitor—another central service within the node monitoring component that constantly gathers resources information via the node controllers. If all pipeline elements requirements are still satisfied, this node is considered eligible and appended to the set. Finally, the set of eligible nodes $\mathcal{N}_{el,\rho}$ for the given pipeline element is returned to be utilized by citizen technologists.

Citizen technologists complete the pipeline authoring process by selecting an eligible deployment target node per pipeline element.The resulting set $\mathcal{Z} = \{<\rho, n>, \ \ldots\}$ which holds a key-value data structure with pipeline element-node description pairs is then sent to the pipeline management. Here, individual pipeline element descriptions are first updated with the assigned node and operation policies which are then used together with directed event stream edges $\mathcal{E}$ and preference-based configurations Π to create a so-called *configured pipeline* $\mathcal{P}_c$. The configured pipeline serves as a basis for generating event stream relays which we discuss in the next section. Algorithm 3 shows the required steps to create a configured pipeline.

Algorithm 3 Complete processing pipeline configuration

Input: a set of pipeline element-node description pairs $\mathcal{Z}$, incomplete pipeline $\mathcal{P}_i$
Output: configured pipeline $\mathcal{P}_c$

1: $\mathcal{P}_c, \mathcal{V}_c \leftarrow \emptyset$
2: $\mathcal{V} \leftarrow$ getPipelineElements($\mathcal{P}_i$)
3: $\mathcal{E} \leftarrow$ getDirectedEventStreamEdges($\mathcal{P}_i$)
4: $\Pi \leftarrow$ getOptionsAndPolicies($\mathcal{P}_i$) ▷ preference-based configurations
5: **for each** $\rho \in \mathcal{V}$ **do**
6: $\quad n \leftarrow$ findSelectedNode($\rho, \mathcal{Z}$)
7: $\quad \rho \leftarrow$ updateDesc(n, extractOp(Π)) ▷ set node target and operation policies
8: $\quad \mathcal{V}_c \leftarrow \mathcal{V}_c \cup \rho$
9: $\mathcal{P}_c \leftarrow (\mathcal{V}_c, \mathcal{E}, \Pi)$ ▷ create configured pipeline
10: **return** $\mathcal{P}_c$

7.3.3 Event Stream Management

After selecting eligible deployment target nodes for pipeline elements, the configured pipeline including deployment options and operation policies is sent to the application management layer in order to prepare the geo-distribution. As a result of the node assignments, pipelines potentially span multiple layers along the cloud-edge continuum with adjacent pipeline elements being dislocated on different fog infrastructure nodes. This induces the need to provide a suitable mechanism that allows to forward output event streams of preceding pipeline elements deployed on one node to their succeeding pipeline elements deployed on another one. As a consequence, a crucial aspect within our holistic management approach for event-driven applications is the *event stream management*, in particular in terms of *event stream relays* based on the EventStreamRelay concept. In the following, we first introduce the notion of *location* by defining pipeline element co- and dislocation before elaborating on the cardinality of communication between pipeline elements. Lastly, we introduce an event dissemination strategy for event-driven applications in fog computing, present a definition for event stream relays and show how to generate unique event stream relays for adjacent, dislocated pipeline elements.

First, let us introduce a helper function when dealing with pipeline element locations. The function targetNode(ρ) allows to retrieve the user-selected deployment target node from the pipeline element description.

$$\text{targetNode}(\rho) = n \qquad\qquad \rho \in \mathcal{V}, n \in \mathcal{N}$$

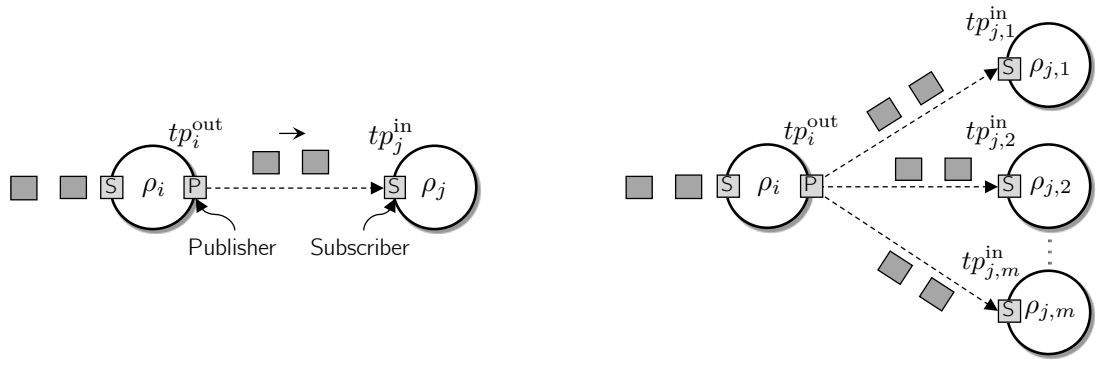
Thus, we can formally define *pipeline element colocation* as follows:

> **Definition 8 (Pipeline Element Colocation).** Within a configured pipeline $\mathcal{P}_c{=}(\mathcal{V}, \mathcal{E}, \Pi)$, we consider two pipeline elements $\rho_i, \rho_j \in \mathcal{V} : \rho_i \neq \rho_j$ *colocated*, if the target node of ρ_i is n_i, such that $\text{targetNode}(\rho_i) = n_i$ and the target node of ρ_j is n_j, such that $\text{targetNode}(\rho_j) = n_j$ and $n_i = n_j$ for $n_i, n_j \in \mathcal{N}$.

Similarly, we can formally define *pipeline element dislocation* which, in principle, is the negation of pipeline element colocation (and vice versa).

> **Definition 9 (Pipeline Element Dislocation).** Within a configured pipeline $\mathcal{P}_c{=}(\mathcal{V}, \mathcal{E}, \Pi)$, we consider two pipeline elements $\rho_i, \rho_j \in \mathcal{V} : \rho_i \neq \rho_j$ *dislocated*, if the target node of ρ_i is n_i, such that $\text{targetNode}(\rho_i) = n_i$ and the target node of ρ_j is n_j, such that $\text{targetNode}(\rho_j) = n_j$ and $n_i \neq n_j$ for $n_i, n_j \in \mathcal{N}$.

Cardinality of Communication. As the pipeline application model discussed in Section 6.2 assumes an EPN, potentially arbitrary pipeline topologies can occur. In contrast to simple, linear pipeline topologies, more sophisticated and complex structures may be



$\rho_i \in \mathcal{V} \backslash S, \rho_j \in \mathcal{V} \backslash A$

(a) single-connected

$\rho_i \in \mathcal{V} \backslash S, \rho_{j,k} \in \mathcal{V} \backslash A, k{=}\{1, 2, \ldots, m\}$

(b) multi-connected

Figure 7.3 Cardinality of communication between pipeline elements

modeled by citizen technologists including segments where the pipeline splits. In such a case, the pipeline element that fans-out the event stream has more than one successor which is, in general, what we refer to as the *cardinality of communication*. Hence, the actual cardinality of communication for a given pipeline element is determined by the number of its successors. Despite arbitrary pipeline topologies, we can generally classify the cardinality of communication according to the following two categories which are illustrated in Figure 7.3:

- *Single-connected*—A single-connected communication considers a $1{:}1$-mapping between exactly two adjacent pipeline elements, i.e., between a predecessor ρ_i and its only successor ρ_j. Events published by ρ_i over its output transport protocol tp_i^{out} are subscribed by ρ_j over its input transport protocol tp_j^{in}. The cardinality is 1.
- *Multi-connected*—A multi-connected communication is a $1{:}m$-mapping between one predecessor ρ_i and multiple successors $\rho_{j,k}$ for $k = \{1, 2, \ldots, m\}$. While the predecessor is always the same a multitude of successors exist. Still, events published by ρ_i over its output transport protocol tp_i^{out} are subscribed by each successor over its dedicated input transport protocol $tp_{j,k}^{\text{in}}$. Here, the cardinality is m. As can be seen, the single-connected communication is a specialization of the multi-connected one for $k = 1$.

In general, as the communication model among any two interconnected pipeline elements is based on the topic-based publish/subscribe pattern, the event stream relay extends this by providing a flexible publish/subscribe concept that allows to map event streams from arbitrary source transport protocols to arbitrary target transport protocols as pointed out in Section 6.4.5. In particular, this approach generalizes the event dissemination among various message broker technologies and reflects the induced needs within the IIoT in terms of lightweight edge and fog transport protocols such as MQTT and scalable cloud transport protocols such as Kafka.

Locality-Aware Event Dissemination Strategy. Following, we elaborate on the concepts behind our *locality-aware event dissemination strategy* (LAEDS). Thereby, LAEDS sits at the core of the event stream management approach for operating geo-distributed processing pipelines and builds the foundations for event stream relays. Thereby, event stream relays play a vital role in realizing complex flows of events in highly geo-distributed systems such as fog computing infrastructures. In this regard, by assigning deployment target nodes to pipeline elements, we introduced the notion of *location* to a pipeline element which is a decisive factor for LAEDS as it determines whether adjacent pipeline elements are co- or dislocated. As such, we provided a formal introduction to the fog infrastructure model in Section 6.3 and stated related assumptions about both *local* and *shared* message broker with publish/subscribe capability. Hereby, LAEDS tries to reduce excessive network round trips to remote message brokers while still employing the flexibility of the publish/subscribe mechanism. Therefore, the main objective is to keep event dissemination between adjacent and colocated pipeline elements local referred to as *intra-node* communication and only leave a node to forward events if any two adjacent pipeline elements are dislocated which is referred to as *inter-node* communication. The communication models employed using LAEDS can be described as following and are further exemplified in Figure 7.4.

- *Intra-node communication model*—Used between any two *adjacent* and *colocated* pipeline elements deployed on edge or fog nodes over a local message broker. Adjacent and colocated pipeline elements on cloud nodes use a shared message broker for event dissemination not necessarily residing on the same node.
- *Inter-node communication model*—Used between any two *adjacent* and *dislocated* pipeline elements and require an *event stream relay* to realize edge-edge, edge-fog, edge-cloud or fog-cloud communication (and vice versa). Similarly to the intra-node case, adjacent and dislocated pipeline elements on cloud nodes, i.e., cloud-cloud communication, use a shared message broker and do not require an event stream relay.

While the intra-node communication model is targeted by extending state-of-the-art transport protocol negotiation and matching mechanisms [Riemer 2016], the inter-node communication model typically requires the event stream relay concept. In this case, the output event stream of the preceding pipeline element is first sent to the local message broker. From there, it is intercepted and forwarded by the event stream relay manager component of the node controller which we discuss in Section 7.3.5. The reason for this additional step is to incorporate resiliency options for event stream relays in situations where network connections are interrupted, e.g., events can be buffered and re-sent after reestablishing the connection. A special case for both communication models are adjacent pipeline elements on cloud nodes. Here, shared cloud message brokers are typically highly available due to several broker replica instances deployed on multiple nodes which makes event stream relays obsolete.

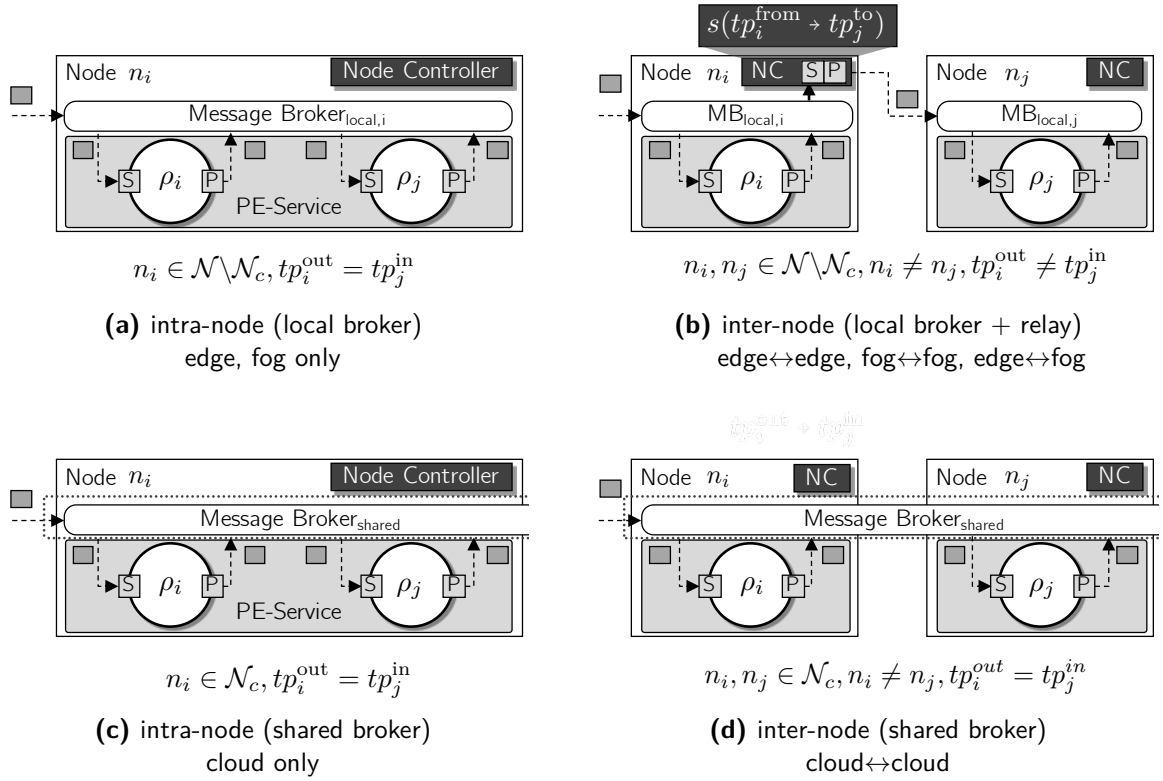


Figure 7.4 Locality-aware event dissemination strategy: Intra and inter-node communication models

Event Stream Relay—Definition and Generation. In the following, we focus on inter-node communication and elaborate on the notion of *event stream relays* that are a key enabler for geo-distributed operation of processing pipelines. Hereafter, we will occasionally use the shorter term *relay* as a synonym for event stream relay. As previously discussed, a relay serves as an intermittent proxy for inter-node event dissemination between adjacent, dislocated pipeline elements. While the transport protocol description for the message broker technology can differ, the identifying topic for the corresponding event stream and thus for publish/subscribe interaction stays the same. Event stream relays are exclusive to its respective processing pipeline and are thus not shared among different pipelines. Consequently, relays are generated according to the user-defined mapping of pipeline elements to deployment target nodes and are thus typically static during the operation phase. Yet, with one exception during pipeline element migration which requires relay modification and is subject of Section 8.3.2. Another crucial aspect results from the topic-based publish/subscribe communication mechanism employed within our event stream model which demands for *unique* target transport protocols. For instance, consider two colocated successors, so-called *neighboring successors*—in such a case, there must only be one relay instance present that forwards events from the preceding pipeline element to the target input stream transport protocols to prevent run-time event duplicates on the subscriber side. Thus, the cardinality of communication differ-

entiates the *logical level* and the *physical level*. The logical level denotes conceptualized processing pipelines as modeled by citizen technologists. The physical level denotes the actual event dissemination during pipeline operation dependent on the node assignment of pipeline elements. Therefore, not all logical predecessor-successor connections result in an instantiated relay. Further, relays are *transient* and only exist at pipeline run-time during the operation phase. In general, we can define event stream relays as follows:

> **Definition 10 (Event Stream Relay).** An event stream relay $s \in \mathcal{S}_r$ is a transient, inter-node communication mechanism for adjacent, dislocated pipeline elements. A relay is a quadruple $s{=}(n, \pi_o^r, tp_i^{\text{from}}, \Gamma_j^{\text{to}})$, where n is the originating node $n \in \mathcal{N}$, π_o^r is the relay operation policy, tp_i^{from} is the source transport protocol for the output-side of the predecessor $\rho_i \in \mathcal{V}\backslash S$, and Γ_j^{to} is a set of unique target transport protocols for the input-side of successors $\rho_{j,k} \in \mathcal{V}\backslash A, k = \{1, 2, \ldots, m\}$.

Given an event stream relay s, the cardinality of communication on a physical level is determined by the size of the set of *unique* target transport protocols Γ_j^{to}. Hence, the physical cardinality of communication is at most m, i.e., the number of successors for a minimum of $m{+}1$ nodes as a prerequisite for evenly distributed and dislocated successors without any neighboring effects:

$$\textsc{card}(s) = |\Gamma_j^{\text{to}}| = \begin{cases} \text{multi-connected}, & \text{for } |\Gamma_j^{\text{to}}| > 1 \text{ with } |\Gamma_j^{\text{to}}| \leq m \\ \text{single-connected}, & \text{for } |\Gamma_j^{\text{to}}| = 1 \end{cases}$$

Next, we will elaborate on how event stream relays are generated within the central pipeline management. Hence, given the configured processing pipeline $\mathcal{P}_c{=}(\mathcal{V}, \mathcal{E}, \Pi)$ with a user-selected deployment target node per pipeline element, we can break down the general task of constructing event stream relays in two phases that altogether reflect the previously stated criteria for event stream relays according to Definition 10, namely (1) *adjacency*, (2) *dislocation* and (3) *uniqueness*. First, relevant pipeline element pairs need to be identified where relays are indispensable (criteria 1 and 2). Second, a distinct set of target transport protocols with only unique elements, i.e., no duplicates, must be ensured (criteria 3). This is especially of relevance in the case of a multi-connected communication with several successors. For the first part, we can formulate a logical statement expressed within the boolean function $\textsc{reqRelay}(\rho_i, \rho_j)$ which defines whether an event stream relay between two input pipeline elements is generally required.

$$\rho_i \in \mathcal{V}\backslash S, \rho_j \in \mathcal{V}\backslash A,\ \rho_i \neq \rho_j : \Big(\textsc{reqRelay}(\rho_i, \rho_j) \iff \textsc{adjacent}(\rho_i, \rho_j) \ \wedge\ \textsc{dislocated}(\rho_i, \rho_j)\Big)$$

First, given a set of directed event stream edges $\mathcal{E}$ from our pipeline description, the boolean function $\textsc{adjacent}(\rho_i, \rho_j)$ defines whether two pipeline elements are adjacent according to Definition 7 (see Section 6.2).

$$\rho_i \in \mathcal{V}\backslash S, \rho_j \in \mathcal{V}\backslash A,\ \rho_i \neq \rho_j : \Big(\textsc{adjacent}(\rho_i, \rho_j) \iff \exists(\rho_i, \rho_j) \in \mathcal{E}\Big)$$

Second, the boolean function $\textsc{dislocated}(\rho_i, \rho_j)$ determines whether two pipeline elements are dislocated according to Definition 9. We can leverage the helper function $\textsc{targetNode}(\rho)$ in order to extract the assigned deployment target node for a given pipeline element.

$$\rho_i, \rho_j \in \mathcal{V}, \rho_i \neq \rho_j : \Big(\textsc{dislocated}(\rho_i, \rho_j) \iff \exists n_i, n_j \in \mathcal{N} : (\textsc{targetNode}(\rho_i) = n_i, \textsc{targetNode}(\rho_j) = n_j, n_i \neq n_j)\Big)$$

While both the adjacency and dislocation criteria postulate the general reason to instantiate an event stream relay at all, it is the uniqueness criteria that needs to be carefully considered when instantiating a relay to prevent run-time event duplicates as previously mentioned. This particularly applies for potential neighboring successors that are both *dislocated* from the same predecessor but in turn are *colocated*. Consequently, these neighboring successors not only share the same node but also share the same input transport protocol on their subscriber side, e.g., the same local node broker and the same topic scheme. In order to evaluate the *uniqueness* criteria for a relay we can examine the *existence* of an event stream relay $s \in \mathcal{S}_r$ for a given *source-target* transport protocol relation and use the negation to assess whether or not such as a relay can be considered as unique. Let us introduce two helper functions when dealing with an event stream relay.

$$\begin{aligned} \textsc{fromSourceTp}(s) &= tp_i^{\text{from}} && s \in \mathcal{S}_r \\ \textsc{toTargetsTp}(s) &= \Gamma_j^{\text{to}} && s \in \mathcal{S}_r \end{aligned}$$

While the function $\textsc{fromSourceTp}(s)$ allows to extract the source transport protocol of the relay, the function $\textsc{toTargetTp}(s)$ extracts the set of unique target transport protocols for all successors. Finally, the boolean function $\textsc{completeMatch}(tp_i^{\text{out}}, tp_j^{\text{in}})$ determines whether an even stream relay already exists given an output stream transport protocol of a predecessor and an input stream transport protocol of a successor. Hereby, Γ denotes a set of all transport protocols.

$$tp_i^{\text{out}}, tp_j^{\text{in}} \in \Gamma : \Big(\textsc{completeMatch}(tp_i^{\text{out}}, tp_j^{\text{in}}) \iff \exists s \in \mathcal{S}_r : (\textsc{fromSourceTp}(s) = tp_i^{\text{from}}, \textsc{toTargetsTp}(s) = \Gamma_j^{\text{to}}, \Gamma_j^{\text{to}} \subset \Gamma \; (tp_i^{\text{from}} = tp_i^{\text{out}}) \; \wedge \; \exists! tp_j^{\text{to}} \in \Gamma_j^{\text{to}} : (tp_j^{\text{to}} = tp_j^{\text{in}})))\Big)$$

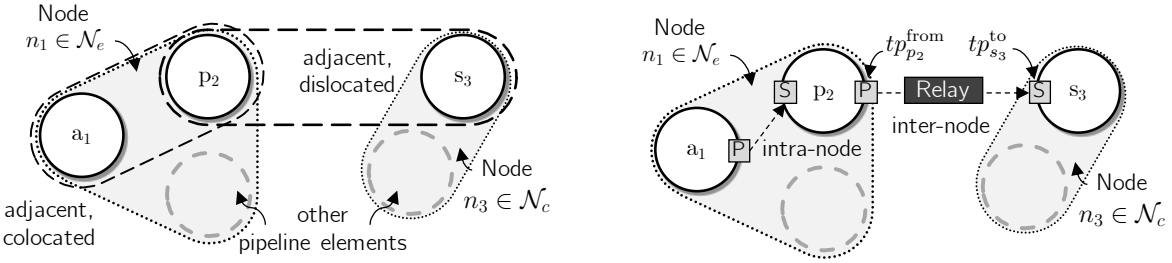
Given the inputs, the existence of an event stream relay and thus a complete match can be assured by verifying matching transport protocols for both the *source*-side and the *target*-side against existing relay descriptions. This requires a full property match of the provided transport protocol descriptions. For instance, two epa:MqttTransportProtocol instances which contain the same properties for the broker hostname "`localhost`", the port "`1883`" and the topic scheme "`org.example.point.in.polygon-1234`" are considered matching. If both condition are met, a designated relay was already added to the relay set for another neighboring successor. Hence, the logical event stream edge from the predecessor to the inspected successor is already covered by an existing relay.

Algorithm 4 Event stream relay generation

Input: a configured pipeline $\mathcal{P}_c{=}(\mathcal{V}, \mathcal{E}, \Pi)$
Output: set of event stream relays $\mathcal{S}_r$

```
1: S_r ← ∅
2: π_o^r ← getRelayOption(P_c)                          ▷ global relay operation policy
3: V ← getPipelineElements(P_c)
4: V\S ← findPredCandidates(V)                          ▷ set of potential predecessors
5: V\A ← findSuccCandidates(V)                          ▷ set of potential successors
6: E ← getDirectedEventStreamEdges(P_c)
7: for each ρ_i ∈ V\S do
8:     for each ρ_j ∈ V\A do
9:         pred ← ρ_i
10:        succ ← ρ_j
11:        if reqRelay(pred, succ) then                 ▷ verify adjacency & dislocation
12:            n_pred ← targetNode(pred)
13:            tp_pred^from ← outTp(pred)
14:            tp_succ^to ← inTp(succ)
15:            Γ_succ^to ← ∅ ∪ tp_succ^to               ▷ init target protocol set
16:            s_new ← new EventStreamRelay(n_pred, π_o^r, tp_pred^from, Γ_succ^to)
17:            if S_r ≠ ∅ then
18:                S_r ← S_r ∪ s_new
19:            else if ¬completeMatch(tp_pred^from, tp_succ^to) then   ▷ verify uniqueness
20:                s_old ← findRelay(tp_pred^from, S_r)
21:                if s_old ≠ null then
22:                    appendToTargetsTp(s_old, tp_succ^to)   ▷ update description
23:                else
24:                    S_r ← S_r ∪ s_new
return S_r
```

Algorithm 4 incorporates the stated relay criteria and functions to describe how an event stream relay is generated based on a configured pipeline $\mathcal{P}_c$. First, a set of event stream relays is initialized to store all generated ones for the given pipeline. Apart from the relay operation policy, a valid predecessor and successor candidate set of pipeline elements as well as all directed event stream edges are extracted from the pipeline description in a preliminary step (Lines 2 to 6). Next, we iterate over the candidate sets and perform a pairwise evaluation of preceding pipeline elements $pred$ and succeeding pipeline elements $succ$ to assess whether an event stream relay is required. This essentially validates the fundamental two relay criteria, namely adjacency and dislocation, by means of the introduced function (Line 11). Next, all required relay properties according to Definition 10 are extracted, including the originating node, the source transport protocol as well as the target transport protocol which is added to an initial target protocol set.



(a) a_1 and p_2: adjacent, colocated
p_2 and s_3: adjacent, dislocated

(b) LAEDS: intra-node for $a_1 \rightarrow p_2$
LAEDS: inter-node for $p_2 \rightarrow s_3$

Figure 7.5 Running example: Event stream management

This allows to create an initial, yet temporary, relay s_{new} for the current $pred{-}succ$-pair (Line 16) which can be added to the overall relay set if this is still empty. Otherwise, the uniqueness criteria must be evaluated to ensure that there is no such relay already present (Line 19). Still in a multi-connected case, additional target protocols must be added to the target set to provide the originating node controller with essential information where to publish events to. Therefore, it is necessary to evaluate if there is a relay, here referred to s_{old}, which has the same source transport protocol as the current temporary one (Line 20). If so, the target transport protocol is added as an additional entry and used to update the existing relay description. If this is not the case, the existing temporary relay is added as a new entry to the overall relay set. Eventually, after iterating over all $pred{-}succ$-pairs, the required relays of the given pipeline are generated and the set of event stream relays is returned.

These are then leveraged in combination with the configured pipeline to distribute individual pipeline elements alongside event stream relay invocation requests which we will describe next.

Example. *Figure 7.5 highlights essential points in view of event stream management aspects. Again, let us use the location monitoring pipeline $\mathcal{P}_{loc}$ and its three pipeline elements, namely the GPS adapter a_1, the point-in-polygon processor p_2 and the notification sink s_3. In order to examine how the event stream management based on LAEDS works, we rely on a configured pipeline $\mathcal{P}_{loc,c}$. The configured pipeline contains information on assigned node for pipeline elements apart from operation policies such as the relay operation policy specified by citizen technologists. This allows to determine what communication model (intra-node, inter-node) between adjacent pipeline elements is employed. Here, the GPS adapter a_1 and the point-in-polygon processor p_2 are assigned to the edge node n_1. At the same, the notification sink s_3 is assigned to the cloud node n_3. This provides essential location information to determine that both a_1 and p_2 are adjacent and colocated while p_2 and s_3 are adjacent but dislocated (Figure 7.5a). As defined by LAEDS, the communication model between a_1 and p_2 falls into the category of the intra-node communication using the local node broker of the edge node. However, as the point-in-polygon processor p_2 and the notification sink s_3 fulfill the two preliminary criteria for event stream relays, namely*

adjacency and dislocation, an event stream relay is required to realize inter-node communication from the edge node n_1 to the cloud node n_3. As the point-in-polygon processor p_2 has no further succeeding pipeline elements, the uniqueness criteria is fulfilled. Consequently, an event stream relay needs to be generated (Figure 7.5b). According to Definition 10, the generated event stream relay $s = (n_1, \langle \textit{buffer}{=}true \rangle, tp_{p_2}^{from}, \{tp_{s_3}^{to}\})$ comprises four essential elements which define it: (1) the origin node for the relay, here the edge node n_1, (2) the user-select relay operation policy π_o^r which can be taken from the configured pipeline $\mathcal{P}_{loc,c}$, here the citizen technologist enabled the buffer option, (3) the publisher output transport protocol of the predecessor, here the point-in-polygon processor with $\text{OUTTP}(p_2) = tp_{p_2}^{from}$ for its output publisher and (4) a set of subscriber input transport protocols of the successors, here only the notification sink $\text{INTP}(s_3) = tp_{s_3}^{to}$ which is subsequently added to the overall target transport protocol set. So, while there is topic equality on both the source and the target transport protocols, the actual represented broker technology might differ. In our case, the relay mediates between the lightweight EPA:MQTTTRANSPORTPROTOCOL *on the edge node and the scalable* EPA:KAFKATRANSPORTPROTOCOL *on the cloud node.*

7.3.4 Pipeline Element and Relay Distribution

From the central application management side, the generation of required event stream relays for a processing pipeline finalizes the required preparation steps prior to geo-distributing individual pipeline elements and required event stream relays to chosen deployment target nodes. Afterwards, all run-time management tasks regarding the execution and operation of individual pipeline elements are delegated to the node controller which we describe in Section 7.3.5.

In order to distribute pipeline elements and even stream relays to destined deployment target nodes, we leverage the concept of service invocation as explained in Section 6.2. Therefore, this allows to easily instantiate required pipeline elements and relays upon receiving individual service invocation requests for *event stream relay invocation* and *pipeline element invocation*. While the event stream relay invocation request encompasses an EVENTSTREAMRELAY description which is directly processed within the node controller itself, the pipeline element invocation request is proxied by the node controller to the colocated pipeline element service. In turn, the pipeline element service leverages the entailed invocation graph which contains all user-defined configurations and allows to instantiate the designated pipeline element type as briefly touched in Section 2.2.3. In general, invocation graphs are serialized as JSON-LD and sent to the respective node controllers via the web standard *Hypertext Transfer Protocol* (HTTP). Thereby, pipeline elements and event stream relays are iteratively rolled-out. At first, all required event stream relays are instantiated. Afterwards, all pipeline elements of the pipeline graph are instantiated according to a post-order traversal, i.e., downstream pipeline elements are started prior to upstream ones. Instantiated relays and the configured pipeline are persisted within the central database on cloud level.

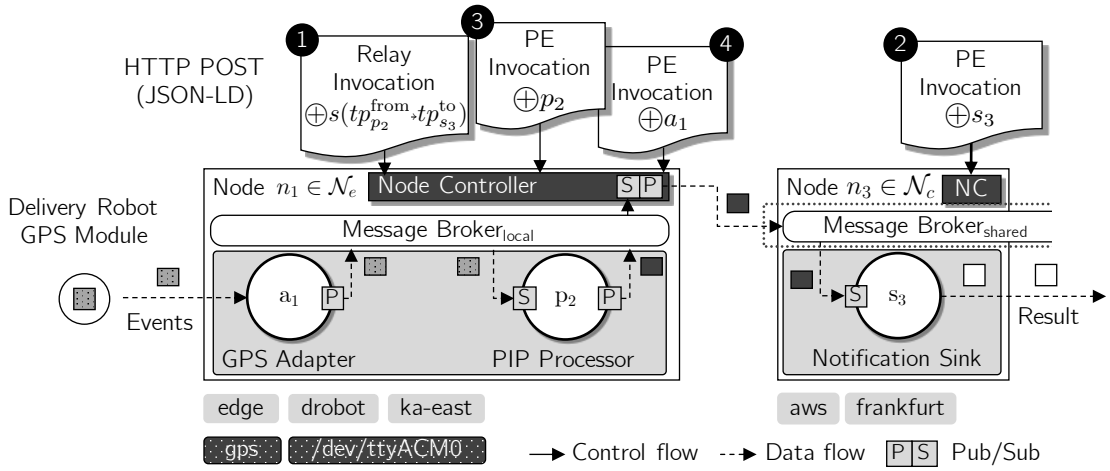


Figure 7.6 Running example: Pipeline element and event stream relay distribution

In view of the remainder of this work, we introduce a formalism for both invoking and detaching pipeline element and event stream relays. The notation opts to aid the reader to better understand respective management tasks during the operation phase. Consequently, we use the symbol "⊕" to indicate a service invocation request and "⊖" to indicate a service detach request as shown in the following overview:

Formalism for the notation

$\{\oplus, \ominus\} s(tp_i^{\text{from}}, \Gamma_j^{\text{to}})$	Relay {*invocation*, *detach*} request to {*start all*, *stop all*} relays from the output transport protocol of a predecessor tp_i^{from} to a unique set of input transport protocols of successors Γ_j^{to}.
$\{\oplus, \ominus\} s(tp_i^{\text{from}} \nrightarrow tp_j^{\text{to}})$	Relay {*invocation*, *detach*} request to {*start*, *stop*} an *individual* relay from the output transport protocol of a predecessor tp_i^{from} to the input transport protocol of one successor tp_j^{to}.
$\{\oplus, \ominus\} \rho_i$	Pipeline element {*invocation*, *detach*} request to {*start*, *stop*} a specific pipeline element ρ_i.

Example. *Figure 7.6 depicts the pipeline element and event stream relay distribution and instantiation of $\mathcal{P}_{loc}$ on two deployment target nodes. Numbers at the invocation requests indicate the deployment sequence. Nodes were selected by the citizen technologist after being validated while respecting custom deployment options $\pi_{d,loc}$=($\langle$selected=$custom\rangle$, $\langle$nodeTags=$\mathcal{T}_{loc}\rangle$) of the given node tags $\mathcal{T}_{loc}$={"aws", "$edge$"} and stated pipeline element requirements. First, the relay is instantiated $\oplus s(tp_{p_2}^{from} \nrightarrow tp_{s_3}^{to})$ for the inter-node communication between the dislocated point-in-polygon processor p_2 and the notification sink s_3 by the node controller on the edge node n_1. Subsequently, the notification sink is started $\oplus s_3$ on the cloud node n_3. Lastly, the point-in-polygon processor $\oplus p_2$ and the GPS adapter $\oplus a_1$ are started on the edge node n_1.*

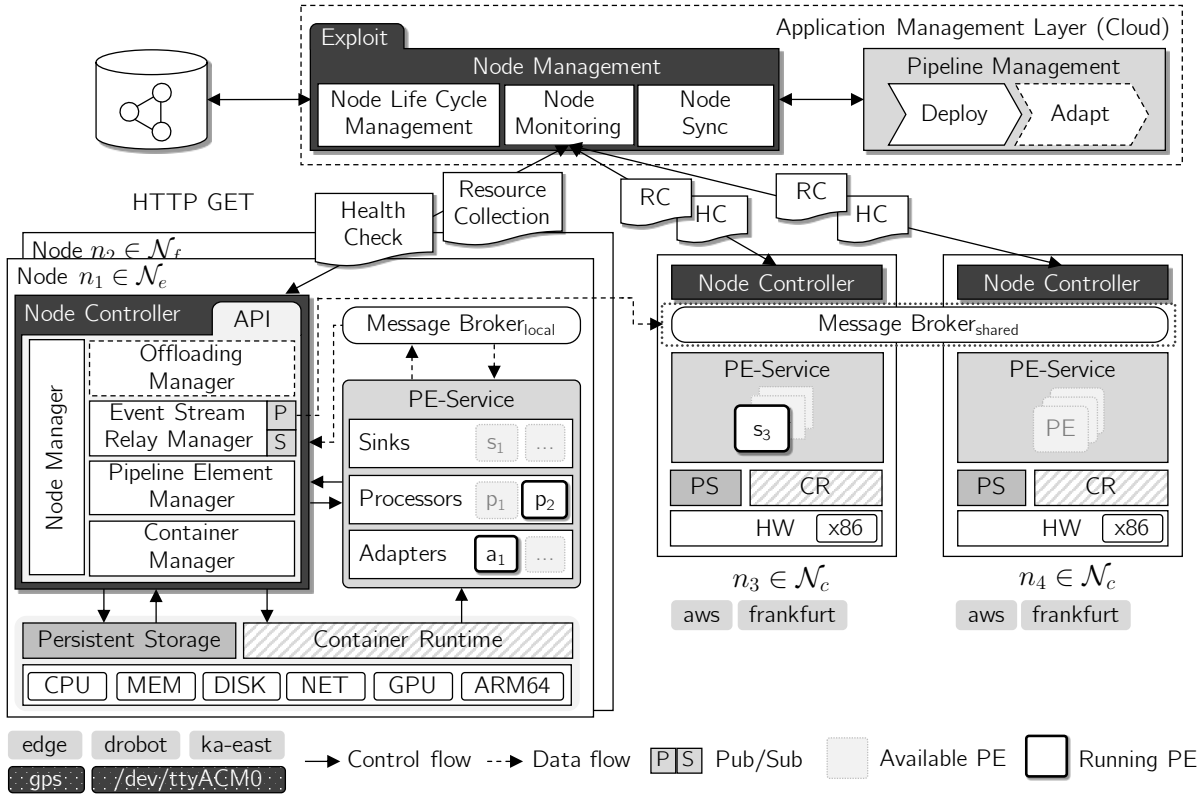


Figure 7.7 Node controller, pipeline element and event stream relay operation: Architecture overview

7.3.5 Node Controller

In this section, we focus on the node controller which receives pipeline element and event stream relay descriptions from the central pipeline management and assures local execution and run-time management during the operation phase. Yet, the node controller covers all related life cycle phases discussed in Section 7.2 in order to ensure a holistic event-driven application management on node level. Following, we provide an in-depth overview of the node controller architecture and elaborate on its interplay with the container runtime and the pipeline element and message broker service as illustrated in Figure 7.7.

In short, the node controller comprises six core components:

- *REST API*—Common API using web standard HTTP protocol
- *Node Manager*—Deals with all node-local management tasks
- *Container Manager*—Common interface to the container runtime
- *Pipeline Element Manager*—Manages pipeline elements throughout their life cycle
- *Event Stream Relay Manager*—Manages the execution of event stream relays
- *Offloading Manager*—Policy-driven offloading detailed in Section 8.3.3

In general, we distinguish between two types of communication, namely *data flow* and *control flow*. The former denotes the discussed dissemination of events throughout the pipeline from external IoT event sources over adapters and processors to sinks. The latter characterizes all system-side messages that in turn carry internal events essential for various management-related tasks in terms of pipeline deployment and operation. Therefore, the node controller exposes a set of management endpoints over a *REST API* for external communication and internal communication as shown in Figure 7.7. On the one hand, external communications describes all communication with the central application management layer, e.g., to perform regular health checks and collect resource metrics. On the other hand, internal communication refers to node-level communication to the pipeline element service.

The *node manager* component is responsible for all node-related management tasks that includes the instantiation of the NodeDescription for the underlying node resources on hardware, software and connectivity level alongside the registration at the central node management during the setup phase as presented in Section 6.5. In addition, the node controller handles the collection and monitoring of resource metrics during the operation phase and provides management interfaces to disable a node in preparation for eventual node maintenance. In particular, the node manager's ability to observe the current node resource consumption is leveraged within the previously described validation procedure to find eligible deployment target nodes prior to pipeline deployment. Relevant metrics are regularly retrieved by the central node monitoring component over the REST API to update the global view on the actual resource situation. Another aspect is related to the usage of container technology for the orchestration of services in general. While the feasibility of container technology within fog computing is indisputable, it introduces a non-negligible challenge in terms of file system littering. This results from partly downloaded container images and outdated volume mappings which reduces the effective disk storage. In particular on resource-constrained edge nodes with limited storage capacity, this becomes a serious issue. Hence, the node manager automatically removes unused image cache and volume mappings to free up disk space.

The *container manager* is the central interface to the present container runtime on a node and is responsible to handle all matters related to orchestrating and managing containerized services, i.e., starting, inspecting, stopping the pipeline element container or message broker container instances. Therefore, the node controller contains a set of DeploymentContainer descriptions which we discussed in Section 6.4.3. These descriptions are explicitly kept generic such that the container manager is able to construct concrete orchestration commands in dependence of the concrete container runtime present. Moreover, upon acknowledging the successful registration of the node controller, the initialization routine instructs the container manager to perform an auto-deployment of respective DeploymentContainer suitable for the given CPU architecture, e.g., x86, ARM64, which starts the pipeline element container and subsequently follows the individual life cycle stages which are reflected within the pipeline element manager.

Hence, the *pipeline element manager* is a central component within the node controller as its responsibility lies in overseeing and managing pipeline elements along their life cycle. This includes several operations, including (1) performing the eager validation upon pipeline element service registration to early identify supported pipeline elements as discussed in Section 7.2, (2) propagating service invocation and detach requests to the pipeline element service to start and stop corresponding pipeline elements, (3) conducting run-time adaptations such as pipeline element *reconfiguration*, *migration* or *offloading* which is part of Chapter 8. Lastly, when a pipeline element container is removed from a node, e.g., when performing an update to a new pipeline element service version, the pipeline element manager first clears the list of supported pipeline elements within the NodeDescription before issuing an update request to the central application management to deregister and remove these elements from the central storage. To perform these operations, the pipeline element manager entails several *interaction handlers* that each address a different stage within the pipeline element life cycle. For instance, once a certain pipeline element is invoked, a specific interaction handler deals with the received service invocation request and proxies it to the pipeline element service. Moreover, other interaction handlers exist, including ones for pipeline element reconfiguration detailed in Section 8.3.1 and offloading detailed in Section 8.3.3. In general, the objective of the pipeline element manager is to ensure the uptime of invoked pipeline elements at run-time in resilient manner. Therefore, once successfully invoked, a reference for the running pipeline element instance is stored in a persistent storage solution external to the container environment due to the ephemeral characteristic of containers as shown in Figure 7.7. Consequently, in case of a node failure or restart, the node controller first redeploys the containerized pipeline element and message broker service. It then checks for formerly stored instance references in the persistent storage and automatically restarts pre-invoked pipeline elements accordingly in order to recover from the failure and ensure the availability of this part of the processing pipeline.

The *event stream relay manager* is another key component of local management capabilities of the node controller. While we already discussed the fundamentals of event stream relays and their generation as part of the central application management in Section 7.3.3, it is the job of the event stream relay manager to instantiate dedicated relays according to the relay service invocation requests and their contained EventStreamRelay description. Therefore, it acts as both subscriber and publisher at the same time while mediating among arbitrary transport protocols for adjacent pipeline elements on different nodes and forwarding event streams from one message broker to another. The EventStreamRelay description not only includes relevant mapping information about the source and target transport protocol but also carries essential preference-based operation policies for relays as discussed in Section 7.3.1. These operation policies denote global run-time management aspects at pipeline-level that, in the case of relays, apply for all event stream relays of the given pipeline. The main objective behind this approach is to allow citizen technologists to configure the event stream relay manager for situations of unreliable network which in the present work includes, but is not limited to, a *purge* and a *buffer* option.

In fog computing, where a subset of nodes are exposed to the physical surroundings of the real world, including mobile node scenarios, a stable network connection cannot be guaranteed at any time and requires a resiliency mechanism employed in the event stream management. Therefore, the proposed relay options are generic preference-based configurations which define certain resilient behavior for the management of event-driven applications in order to meet use case-specific demands. While the purge option is a naive strategy that simply discards events in case of an offline situation and is valuable in situations where temporary unavailability of the the application can be tolerated. The buffer option implicates a strategy that stores events upon detecting connectivity issues to the downstream message broker of the succeeding pipeline element by providing a suitable and flexible storage approach. Upon reestablishing a stable connection, buffered events are re-sent while guaranteeing the order. For instance, such a flexible storage approach for buffering event streams can be achieved by leveraging ring buffer data structures over a fixed-length queue. Hence, the fixed length determines the maximum capacity of events that can be buffered in order to bridge intermittent network outages. Moreover, overflow situations can be managed by either dropping incoming events or overwriting the oldest event in the queue. Lastly, a reference for instantiated event stream relays is persistently stored external to the node controller container which is used to recover from node failures by reinstantiating previously active relays.

Lastly, the *offloading manager* is a component which prevents node resource overload by using a policy-driven approach to define specific pipeline element offloading policies according to one or more node resource violations. Upon detecting a violation, a pre-defined selection strategy is triggered to find a pipeline element candidate to be offloaded at run-time. This allows to move decision-making to the node controller itself which leads to a partial node autonomy where the node controller observes its own context and acts according to the defined strategy. Therefore, the offloading manager leverages the information on the preemption operation policy and the configured priority class which are defined by citizen technologists upon completing the pipeline authoring process. The offloading approach will be detailed in Section 8.3.3.

7.4 Tools

The concepts and methods presented and discussed within this chapter is integrated into the Apache StreamPipes project in order to allow citizen technologists to specify preference-based deployment options and operation policies to deploy and operate geo-distributed pipelines. Figure 7.8 exemplifies the configuration dialog presented to citizen technologists as part of the advanced deployment settings. Here, the known location monitoring pipeline with its three pipeline elements is prepared for deployment. Therefore, the configuration dialog splits into two parts, namely operation policies and deployment options. While *preemption* is enabled with the priority class set to "high" in addition to a

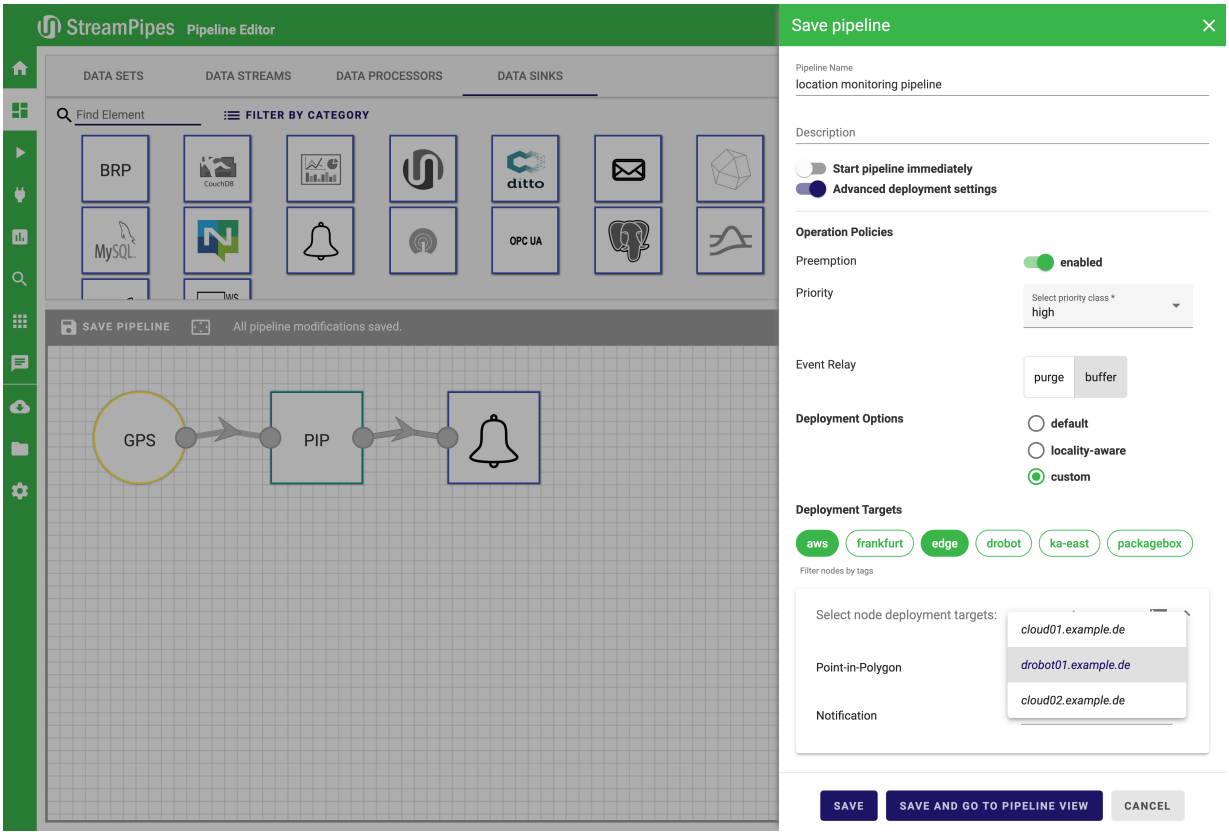


Figure 7.8 Geo-distributed pipeline management: Tool support

buffer relay option as pipeline global operation polices, the *custom* deployment option is chosen with user-selected node tags, here "aws" and "edge", to alleviate the node selection process. Eligible deployment target nodes are elected within the application management layer according to the discussed coarse to fine-grained node filtering procedure and presented to the user for final selection. Besides the custom deployment options, *default* and *locality-aware* ones exist that incorporate specific node mapping strategies and demonstrate the extensibility of the concept. The former can be used for simple data acquisition tasks for event streams originating from IoT devices, where the adapter is deployed on a user-selected edge node and all remaining pipeline elements are assigned to central cloud nodes. In contrast, the latter follows a basic bin packing approach and assigns as many pipeline elements as possible on the node that hosts the adapter until node resources are exhausted. Remaining, unassigned pipeline elements will be randomly assigned to other nodes. Still, the custom deployment options provides the most flexibility to citizen technologists and is well-suited for a broad spectrum of use cases including the ones presented in this thesis. Consequently, this represents another crucial building block to face the emerging needs towards democratizing the application management in fog computing.

7.5 Summary

In this chapter, we presented concepts and methods with regard to geo-distributed pipeline deployment and operation in view of Research Question 2. The first contribution denotes concepts for geo-distributed pipeline management including deployment and operation aspects which are entailed in the central application management within our proposed system architecture. As such, after having clarified critical stages along the pipeline element life cycle and discussed the eager validation procedure to early dismiss any incompatibilities, the main focus lied on the overall geo-distributed deployment and operation process. We presented preference-based deployment options that allow citizen technologists to assign pipeline elements to only eligible deployment target nodes. Therefore, we defined an extensible coarse to fine-grained validation procedure to find eligible deployment target nodes for pipeline elements while considering both user-selected node tags and actual resource availability. In addition, preference-based operation policies can be chosen to configure the run-time behavior of a processing pipeline and their surrounding application management. The second contribution is a generic approach for the dissemination of event streams referred to as event stream relays that forward event streams between logically adjacent but physically dislocated pipeline elements. The employed locality-aware event dissemination strategy for intra-node and inter-node communication was proposed which uses node assignment information of configured pipelines to infer location knowledge on adjacent pipeline elements and decide which communication model to use. In the case of a required inter-node communication, location information in addition to the cardinality of communication are used in the event stream relay generation procedure. The third contribution is a node controller which manages the local execution of pipeline elements and event stream relays along the pipeline element life cycle. Therefore, the node controller interacts with the application management layer and receives service invocation requests for pipeline elements and event stream relays which in turn are are either delegated to the colocated pipeline element service or processed internally. Finally, as previously presented, the concepts from this chapter are implemented and available within the Apache StreamPipes projects to give necessary tool support to citizen technologists to deploy and operate geo-distributed pipelines in fog infrastructures.

8

Pipeline Adaptation

In the previous chapter, we introduced conceptual work for geo-distributed pipeline deployment, operation and management. Within this chapter, we present concepts and methods related to Research Question 3 and discuss how to adapt processing pipelines at run-time which involves *reconfiguration* and *relocation* aspects of running pipeline elements. First, we give a brief *walkthrough* in Section 8.1 before proposing a general adaptation *methodology* in line with the definition of the event term in Section 8.2. Next, we present *run-time evolutions* for processing pipelines in Section 8.3. We show relevant realizations of the conceptual work as part of *tool support* in Section 8.4 before summarizing in Section 8.5.

8.1 Walkthrough

After geo-distributing all pipeline elements and event stream relays for a user-modeled pipeline, from a holistic application management standpoint, it is crucial to deal with changing conditions due to the dynamics in the real-world or newly arising application-specific requirements. Considering this properly allows to sustain turning raw data into insights and ultimately to meaningful knowledge on the user-side. Yet, this demands for a suitable pipeline adaptation methodology that allows in-flight modifications of running pipeline elements, i.e., run-time changes without the need for redeployment. On the one hand, such modifications include *reconfiguration* actions of pipeline element configurations, e.g., quality threshold parameters, or geofence coordinates, that need to be updated and refined accordingly. On the other hand, this also implies *relocation* actions where individual pipeline elements are migrated and thus moved away from their original deployment target nodes to another eligible one, both within a resource layer and across the hierarchical fog architecture. Consequently, as such adaptation forces occur, dedicated pipeline adaptations are to be performed which are categorized as *external* adaptations triggered by citizen technologists or *internal* adaptations triggered by the system itself. Hence, in order to realize the proposed adaptation methodology, the previously introduced system architecture is complemented by concepts and methods as part of essential adaptation components at the central application management layer and, analogously, on the node controller level.

8.2 Methodology

Unbounded, streaming event sources and their resulting event streams characterize the event processing model entailed within processing pipelines. Once deployed, processing pipelines and their comprised pipeline elements are continuous and long-running which implies static run-time behavior and restricted flexibility of the corresponding event-driven application to adapt to changes. However, as the real-world cannot be assumed to be static, this demands for suitable mechanisms in order to allow altering initial configurations and execution targets of individual pipeline elements if necessary. Though naive approaches including a stop-the-world pattern may be realized by stopping, updating, and redeploying existing processing pipelines using concepts described in Chapter 7, this inevitably increases pipeline downtimes and thus incurs service unavailability which is not acceptable for most real-world scenarios, in particular in the case of mission-critical deployments.

Consequently, this demands an approach which allows processing pipelines to evolve over time, commonly referred to as *pipeline evolution*. According to [Andrade et al. 2014], the notion of pipeline evolution describes the ability for the event-driven application to dynamically adapt to changing conditions or requirements. Thereby, there are two decisive dimensions which underpin the adaptation methodology, namely the *adaptation type*, and *decision-making origin* explained in the following.

- *Adaptation type*—The type of adaptation can be either a *reconfiguration* or a *relocation*. In the former case, static properties which configure the event processing logic entailed in individual pipeline elements are altered compared to their original configuration value without affecting the overall pipeline deployment topology. In contrast in the latter case, individual pipeline elements are moved from their original deployment target node to another eligible one. This profoundly affects the overall pipeline execution topology and requires necessary treatment.
- *Decision-making origin*—The decision-making origin represents either *external* and *internal* decision-making. The former includes user-initiated adaption actions based on *domain knowledge* of citizen technologists. The latter reflects autonomous, system-initiated adaptation actions within the application management components of our proposed architecture according to inherent *context knowledge*. In general, context-awareness is the ability of a system to independently detect and respond to changes in their situated environment [Schilit et al. 1994]. In our case, context-aware processing pipelines allow to be modified at run-time based on contextual changes. The term context may depend on time, location, resource utilization or any other property with relevance for the application.

Deduced from the notion of events from Section 2.1.2, where events are occurrences within a particular *system* or *domain*, the flow of desired pipeline adaptation actions from either decision-making origin can itself be represented as events. Concretely, both system-initiated and user-initiated actions express the intention of a specific adaptation

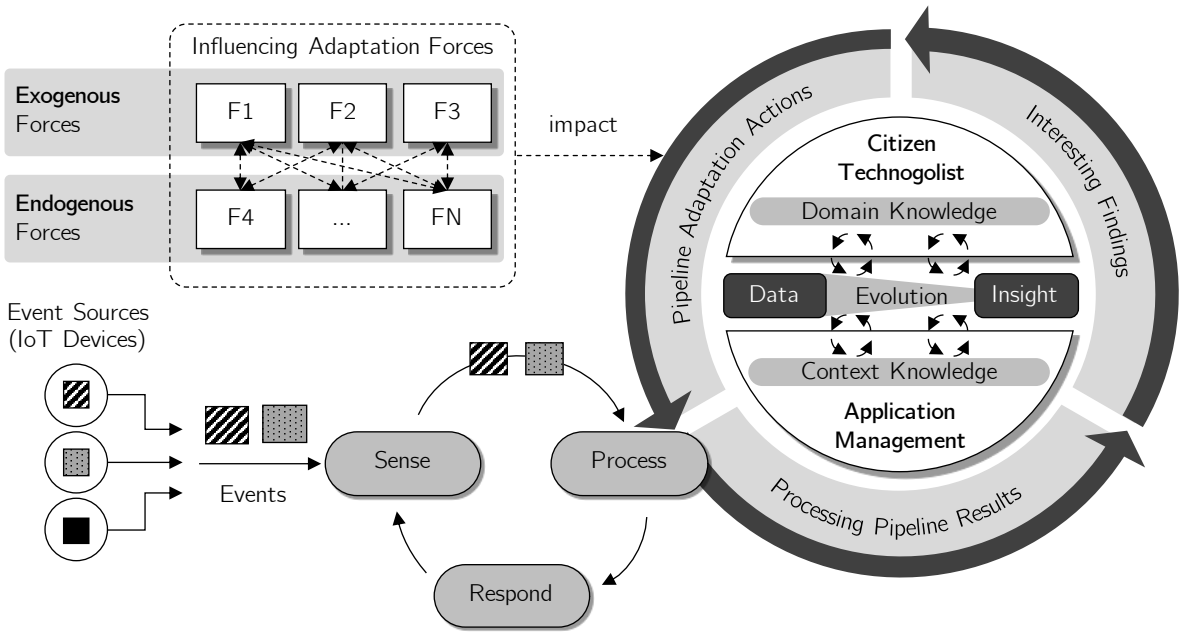


Figure 8.1 Pipeline adaptation methodology: Overview

and thus create events which can be observed and lead to a state change as postulated by [Mühl et al. 2006]. This either affects the encapsulated event processing logic in the case of a reconfiguration or the whole pipeline execution topology in the case of a relocation.

In the following, we propose an adaptation methodology on the basis of the event terminology and characteristics. The methodology attaches to the introduced sense-process-respond principles of EDAs (see Section 2.2.1) and is depicted in Figure 8.1. Over time, event-driven applications are continually impacted by influencing adaptation forces, namely *exogenous* and *endogenous* forces. For instance, exogenous adaptation forces result from newly arising business requirements or changing applicational needs. Moreover, endogenous adaptation forces are induced by mobile edge nodes leading to geo-context changes or by exceeding thresholds for computational resource limits leading to system-context changes. As a conceptual model, the methodology centers around the two previously described dimensions. Thereby, it offers certain degrees of freedom to citizen technologists to apply their domain expert knowledge while also incorporating the ability for system-side context knowledge employed within the application management. This not only allows to continuously refine analytical results. But, it also aids to build up the user's trust and confidence for the event processing part itself while arouse curiosity during the iterative process—from first results provided by the processing pipeline, over initial interesting and meaningful findings when citizen technologists apply their domain knowledge to interpret these results, to issued pipeline adaptation actions in order to generate new and refined results. Eventually, this allows to gain valuable insights while benefiting from the cognitive and perceptive skills of human-system interaction to facilitate continuous pipeline evolution.

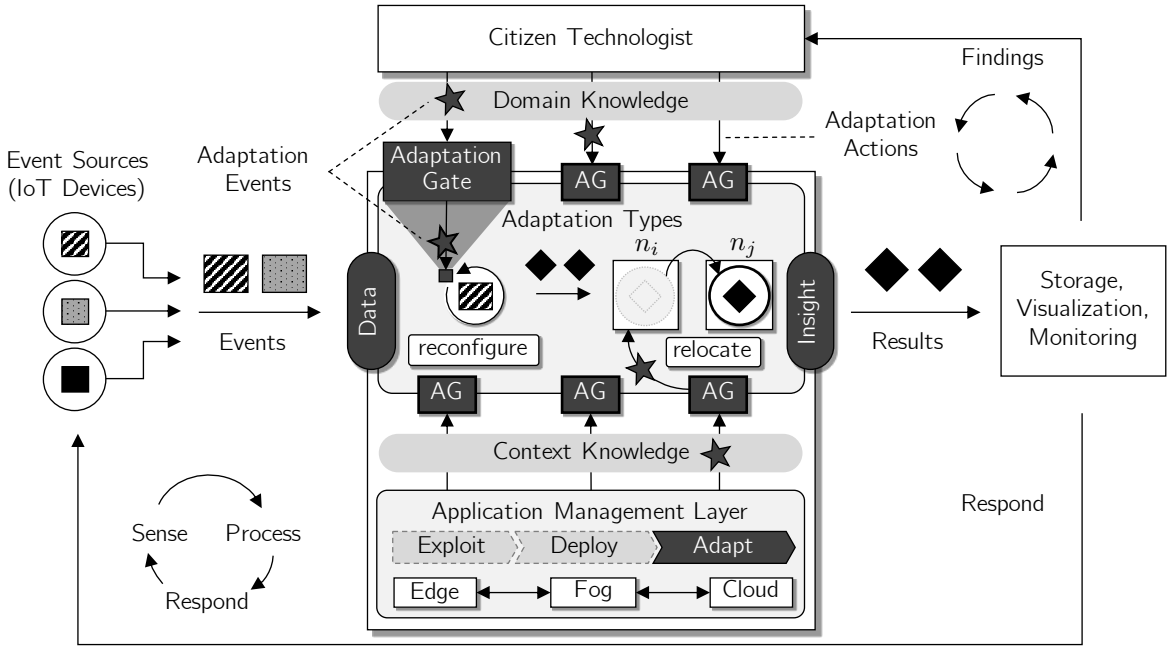


Figure 8.2 Pipeline adaptation methodology: Adaptation gate and adaptation event

Therefore, we introduce two key abstractions which facilitate the event processing pattern and sit at the core of the adaptation methodology, namely *adaptation gates* and *adaptation events* which are defined as follows:

> **Definition 11 (Adaptation Gate and Adaptation Event).** An *adaptation gate* is a special-purpose side input to pipeline elements tasked with receiving designated adaptation actions in the form of specific adaptation events. An *adaptation event* is a transient event created by the user or the system that contains instructions to dynamically change internal configurations of an associated pipeline element.

Figure 8.2 illustrates the adaptation methodology applied to our architecture on a high-level while leveraging both adaptation gates and adaptation events to inject adaptation actions from citizen technologists or the application management-side. Conceptually, adaptation gates do not differ from regular input ports consuming an event stream from external IoT event sources or preceding pipeline elements other than the event type. Hereby, in contrast to regular events, adaptation events do not produce any output events. Instead, adaptation events are used to either alter internal configurations, i.e., static properties, which are part of the event processing logic (reconfiguration) or prepare the migration of the corresponding pipeline element (relocation). In order to realize the concept of adaptation gates and adaptation events, we base our methodology on the topic-based publish/subscribe messaging pattern. Thereby, adaptation events are published on a unique system-managed topic scheme which binds to the adaptation gate (additional subscriber) of the associated pipeline element instance. Similar to regular subscribers, the adaptation gate is generated upon pipeline element instantiation.

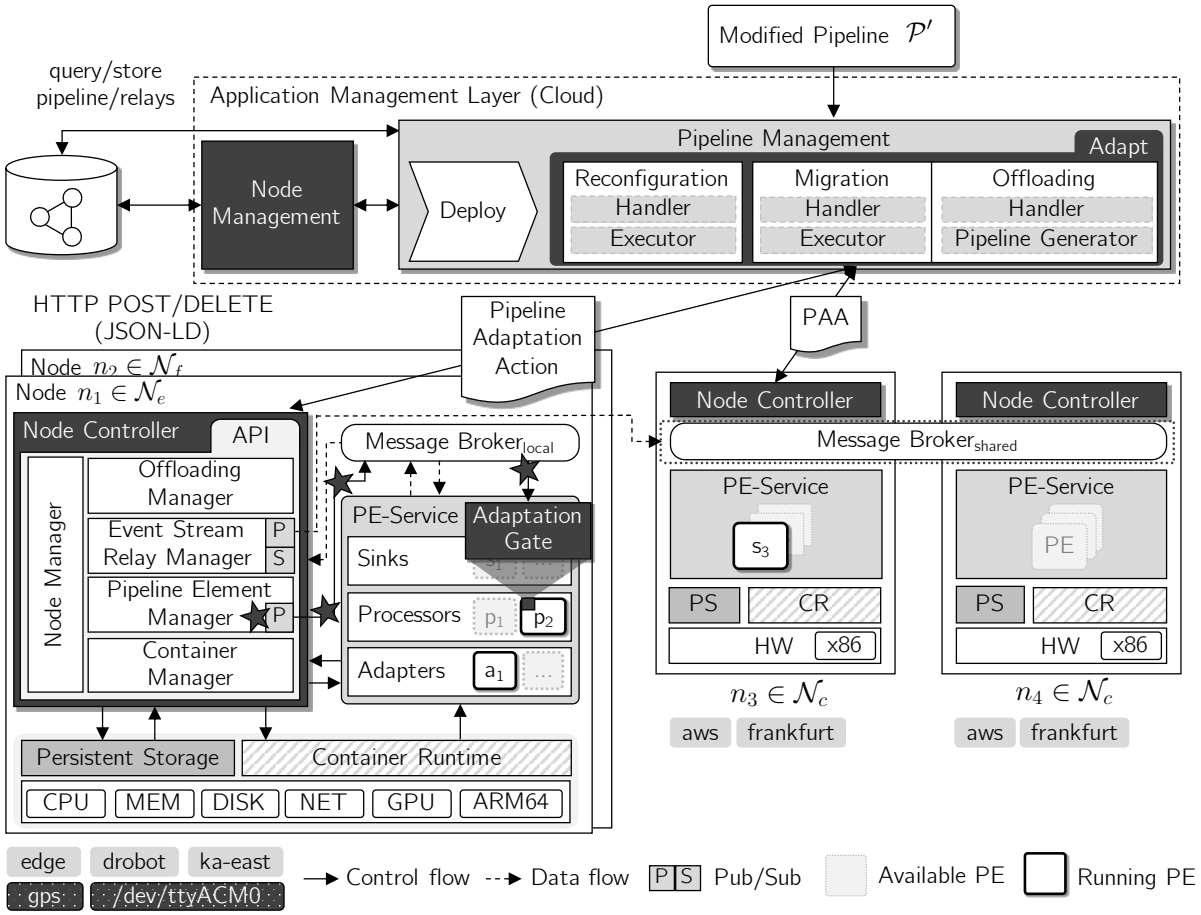


Figure 8.3 Pipeline evolution at run-time: Architecture overview

8.3 Run-Time Evolution

The proposed system architecture is complemented by essential components in order to realize the adaptation methodology at both the central application management layer and the node controller described in the following and illustrated in Figure 8.3.

As applicational needs and requirements change over time, citizen technologists may issue pipeline adaption actions in the form of reconfigurations or relocations. This allows to alter running pipelines from their original state $\mathcal{P}$ to a *modified* state $\mathcal{P}'$ (yet still configured) using the central graphical interface (not shown in Figure 8.3). As a result, the application management layer receives altered pipeline descriptions, further processes respective adaptation actions and redistributes them according to involved nodes. Despite these degrees of freedom for citizen technologists, the adaptation approach is kept generic and thus not limited to user-initiated actions only. In fact, adaptation capabilities are employed within the offloading manager of the node controller. Concretely, the offloading manager continuously observes node resource metrics and initiates offloading actions to evict pipeline elements in order to prevent over-utilized nodes while respecting specified

operation policies for preemption, e.g., the priority class. Finally, both user-initiated and system-initiated pipeline adaptation actions are continuous by nature and allow processing pipelines to evolve over time to adequately adapt to the constant changes.

Next, we detail the stated adaptation types with regard to *reconfiguration* and *relocation* to support pipeline evolution at run-time and give a detailed description on concepts and involved components. In the following, we focus on user-initiated reconfigurations of pipeline elements and deliberately exclude system-initiated ones on the basis of context knowledge, though generally supported by the concepts. Regarding relocation actions, we further differentiate between external, user-initiated relocations referred to as *migration* and internal, system-initiated relocations referred to as *offloading*.

8.3.1 Reconfiguration

Reconfiguration of deployed processing pipelines implies updating initially specified user configurations, i.e., static properties, of associated pipeline elements and thus alter employed parameters from their original value to a new target value as influencing adaptation forces arise. User-initiated reconfiguration actions are an integral part of the reconfiguration process as they provide citizen technologists with the ability to directly manipulate the event processing logic at run-time. Hence, this approach symbiotically combines system-side processing capabilities with human-side perceptive skills, cognitive reasoning according to expert domain knowledge. On the technical side, reconfigurations are based on the concept of ReconfigurableStaticProperty (see Section 6.4.4) and used to denote reconfigurable user configurations. In the operation phase, an additional reconfigure transition in the running state of the pipeline element life cycle (see Section 7.2) allows to mutate the original event processing logic by altering the respective static property. While the concepts behind pipeline element reconfiguration are generic, in practice, such run-time modifications are only of relevance for processors and sinks, omitting adapters. This is due to the fact that adapters only emit raw events originating from real-world IoT devices and thus do not mutate any state. Yet, we illustrate the execution of run-time reconfigurations on the basis of processors in the following.

First, this requires essential preparation steps for modified pipelines at the central application management layer in order to send out specific reconfiguration action requests to involved node controllers. Second, node controllers need to construct and publish adaptation events from received reconfiguration actions on adaptation gates bound to running processor instances to alter specific reconfigurable static properties as illustrated in Figure 8.3. Given a modified pipeline $\mathcal{P}'$, the reconfiguration handler employed within the central pipeline management is responsible for (1) computing reconfiguration actions, (2) triggering the reconfiguration executor to submit these reconfiguration actions to involved node controllers and (3) proxying the reconfiguration status message back to the graphical user interface to inform citizen technologists (not shown in Figure 8.3). Algorithm 5 states all relevant steps performed by the reconfiguration handler.

Algorithm 5 Compute and execute reconfiguration actions for modified reconfigurable static properties of processors within the reconfiguration handler

Input: modified pipeline $\mathcal{P}'$
Output: reconfiguration status message

1: $\mathcal{P}_{old}, \mathcal{V}'\backslash(A' \cup S'), \mathcal{V}_{old}\backslash(A_{old} \cup S_{old}), \mathcal{A}_{rec}, Status_{rec} \leftarrow \emptyset$ ▷ initialize
2: $id \leftarrow$ getPipelineId($\mathcal{P}'$)
3: $\mathcal{P}_{old} \leftarrow$ queryPipelineById(id) ▷ get stored pipeline
4: $\mathcal{V}'\backslash(A' \cup S') \leftarrow$ getProcessors($\mathcal{P}'$)
5: $\mathcal{V}_{old}\backslash(A_{old} \cup S_{old}) \leftarrow$ getProcessors($\mathcal{P}_{old}$)
6: **for each** $\rho' \in \mathcal{V}'\backslash(A' \cup S')$ **do**
7: **for each** $\rho_{old} \in \mathcal{V}_{old}\backslash(A_{old} \cup S_{old})$ **do**
8: **if** matchingProcessors(ρ', ρ_{old}) **then**
9: $\Delta_{\rho'} \leftarrow$ getStaticProperties(ρ')
10: $\Delta_{\rho}^{old} \leftarrow$ getStaticProperties(ρ_{old})
11: $\Delta_{\rho'}^{rec} \leftarrow \emptyset$
12: **for each** $\delta_{\rho'} \in \Delta_{\rho'}$ **do**
13: **for each** $\delta_{\rho}^{old} \in \Delta_{\rho}^{old}$ **do**
14: **if** isRec($\delta_{\rho'}$) **and** isRec(δ_{ρ}^{old}) **then** ▷ reconfigurable properties
15: **if** matchAndModified($\delta_{\rho'}, \delta_{\rho}^{old}$) **then**
16: $\Delta_{\rho'}^{rec} \leftarrow \Delta_{\rho'}^{rec} \cup \delta_{\rho'}$ ▷ modified property set
17: **if** $|\Delta_{\rho'}^{rec}| > 0$ **then**
18: $a_{rec} \leftarrow new$ ReconfigurationAction($\rho', \Delta_{\rho'}^{rec}$)
19: **if** $a_{rec} \notin \mathcal{A}_{rec}$ **then**
20: $\mathcal{A}_{rec} \leftarrow \mathcal{A}_{rec} \cup a_{rec}$
21: **for each** $a_{rec} \in \mathcal{A}_{rec}$ **do**
22: $s_{rec} \leftarrow new$ ReconfigurationExecutor(a_{rec}).$execute()$ ▷ invoke reconfiguration
23: $Status_{rec} \leftarrow Status_{rec} \cup s_{rec}$
24: **if** allSuccessful($Status_{rec}$) **then** ▷ verify reconfiguration success
25: overwriteAndStore($\mathcal{P}'$) ▷ store modified pipeline
26: **return** reconfiguration successful, stored $\mathcal{P}'$
27: **else**
28: rollback($\mathcal{P}_{old}$) ▷ rollback to old pipeline
29: **return** reconfiguration failed, rolled back $\mathcal{P}_{old}$

To compute reconfiguration actions, it is necessary to first retrieve the description of the currently running pipeline $\mathcal{P}_{old}$ from the database. As the modified pipeline and the old pipeline are syntactically equal, a pairwise comparison is performed between all reconfigurable static properties to find the ones that were changed in oder to store them in a modified property set $\Delta_{\rho'}^{rec}$ (Lines 6 to 16). This modified property set is used to instantiate new reconfiguration actions per modified processor. Each of the

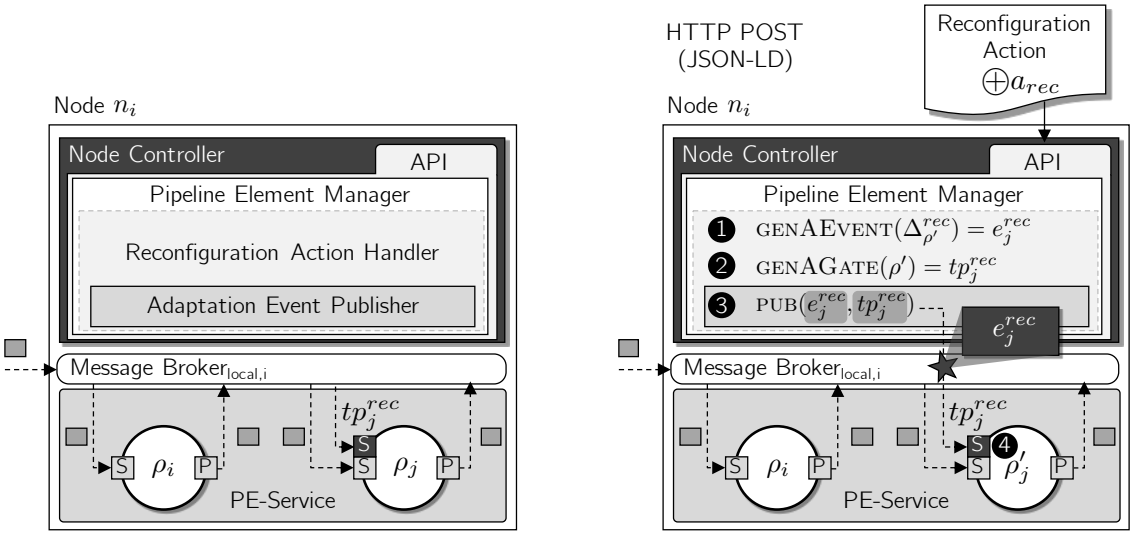
reconfiguration actions is then handed to the reconfiguration executor in order to submit the request to designated node controllers (Line 22). Eventually, the node controller returns a status message containing information on the success or failure of a processor reconfiguration which is collected in a set of status messages. Lastly, all status messages are verified for success which determines whether to overwrite the old pipeline in favor of the modified one, or to initiate a rollback to the original state of static property values. In addition, an overall reconfiguration status message is returned to inform citizen technologists about the current state (Lines 24 to 29).

On the node controller side, upon receiving the reconfiguration action, the pipeline element manager is instructed to execute the run-time reconfiguration of the processor instance defined in the request. In this matter, the reconfiguration action is first processed in order to extract the entailed information on the modified property set in addition to the associated processor instance that is affected by the reconfiguration. We introduce two helper functions to prepare the execution of the run-time reconfiguration within the pipeline element manager component of the node controller:

$$\begin{aligned} \textsc{genAEvent}(\Delta_{\rho'}^{rec}) &= e_j^{rec} & \Delta_{\rho'}^{rec} &\subseteq \Delta_\rho \\ \textsc{genAGate}(\rho') &= tp_j^{rec} & \rho' &\in \mathcal{V} \end{aligned}$$

To this end, the $\textsc{genAEvent}(\Delta_{\rho'}^{rec})$ function takes the modified property set and generates the required adaptation event which uses a key-value based data structure with a reference identifier for the processor instance as the key and the modified property as the value. Furthermore, the $\textsc{genAGate}(\rho')$ function computes the correct adaptation gate bound to the associated, modified processor instance. This represents an additional side-input in the form of a reconfiguration transport protocol using a unique system-managed topic scheme. At this point, it is worth noting that each processor which models a static property as reconfigurable, automatically subscribes to the reconfiguration transport protocol to receive adaptation events. This assimilates the mechanism of regular input-side subscribers using their input transport protocols to receive events from preceding pipeline elements.

Figure 8.4 illustrates two points in time in the running state of processors ρ_i and ρ_j. Time t_{run} shows the running state after being instantiated in the course of the geo-distributed pipeline deployment (Figure 8.4a). As shown, not all processors contain reconfigurable static properties. While processor ρ_j contains reconfigurable static properties and thus instantiates a side-input subscriber upon invocation, the static properties used in ρ_i are not reconfigurable and thus cannot be changed at runtime. At time t_{rec}, the node controller receives the reconfiguration action request (Figure 8.4b) and performs essential local preparation tasks prior to executing the run-time reconfiguration which involves the following four steps: In **step 1**, the corresponding adaptation event is generated by leveraging the $\textsc{genAEvent}(\Delta_{\rho'}^{rec})$ function. Next, in **step 2**, the adaptation gate is generated by using the $\textsc{genAGate}(\rho')$ function, i.e., reconfiguration transport protocol for the associated processor instance. After, in **step 3**, the adaptation event is published using

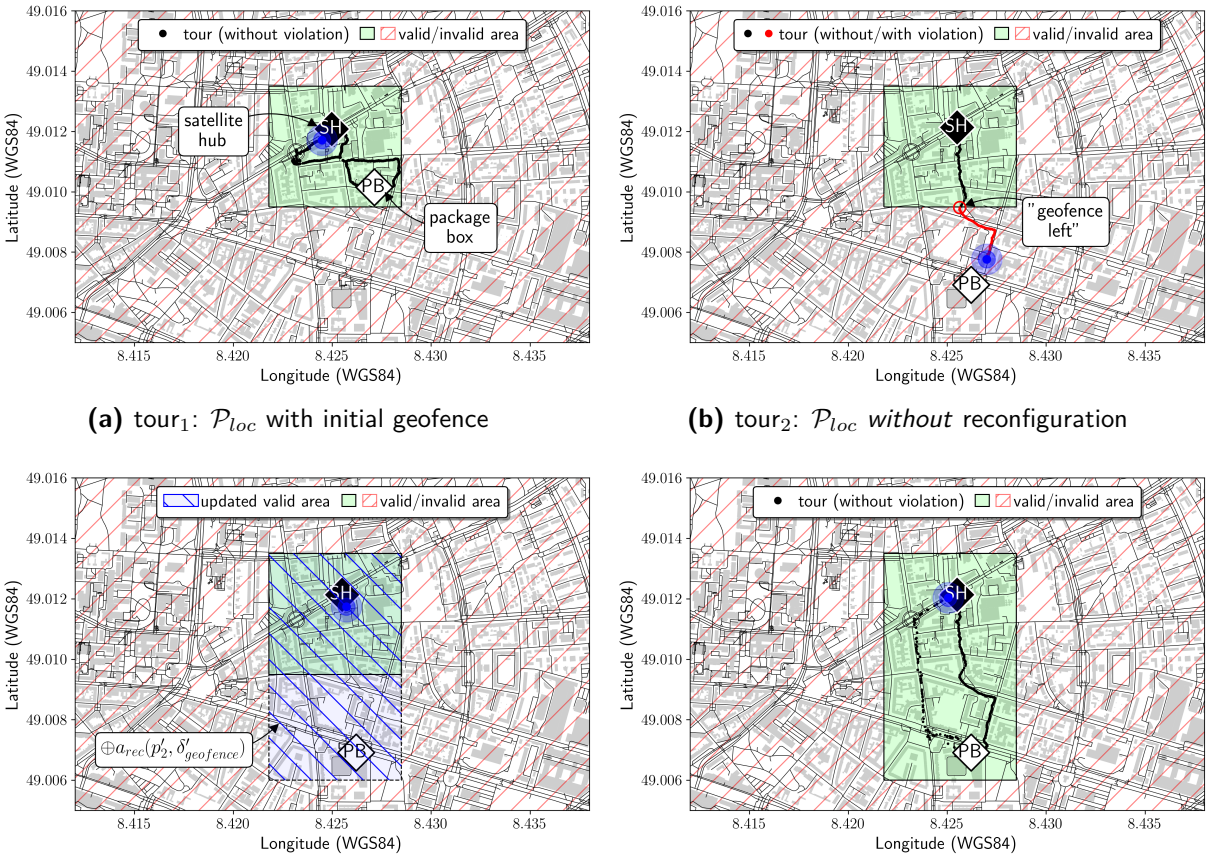


(a) t_{run}: running ρ_i and ρ_j **(b)** t_{rec}: reconfigure ρ_j to modified state ρ'_j

Figure 8.4 Processor run-time reconfiguration performed by node controller

the reconfiguration transport protocol. On the pipeline element service side and in the final **step 4**, the underlying processor runtime instantaneously receives the adaptation event over its reconfiguration side-input and finalizes the reconfiguration. Therefore, the modified property set is reconstructed from the adaptation event and used to update the static properties of the associated processor instance. After successfully completing the reconfiguration, the node controller informs the central application management layer about its success by returning a corresponding reconfiguration status message.

Example. *Figure 8.5 shows two pickup and delivery scenarios (tour$_1$ and tour$_2$) in the city of Karlsruhe. At the same time, the location monitoring pipeline $\mathcal{P}_{loc}$ is deployed on the delivery robot and running. This includes, among the others, the geofencing component referred to as the point-in-polygon processor p_2. During the delivery from the satellite hub (SH) to a package box (PB), the geolocation of the delivery robot is constantly monitored and compared against the initially provided geofence for tour$_1$ employed as a static property in p_2. Figure 8.5a shows the GPS trace for tour$_1$ where no violations are detected. After a larger tour$_2$ is scheduled, the delivery robot pursues the delivery, yet this time to another PB. As shown in Figure 8.5b shows the situation without any run-time reconfiguration. The geofencing criteria is violated as the statically defined valid operation area (geofence) cannot be updated. To overcome this limitation, the geofence can be modeled as reconfigurable in order to be modified during operation. At the same time when sending out the tour schedule for tour$_2$, the fleet operator updates the valid operation area by reconfiguring the geofence. Figure 8.5c illustrates the reconfiguration action request a_{rec} sent from the cloud to the delivery robot containing the updated coordinates $\delta'_{geofence}$ for the modified processor p'_2. Figure 8.5d shows the GPS trace for tour$_2$ with reconfiguration.*



(a) tour$_1$: $\mathcal{P}_{loc}$ with initial geofence

(b) tour$_2$: $\mathcal{P}_{loc}$ *without* reconfiguration

(c) update valid area (reconfigure geofence)

(d) tour$_2$: $\mathcal{P}_{loc}$ *with* reconfiguration

Figure 8.5 Running example: Run-time geofence reconfiguration. Map data © OpenStreetMap [OpenStreetMap contributors 2021].

8.3.2 Migration

Another essential run-time adaptation type characterizes the relocation of pipeline elements. In concrete view of migrating individual pipeline elements from their original deployment target node to another eligible one, the act of migration describes the result of an external, user-initiated migration action. Thereby, the migration action is based on the domain expertise of citizen technologists reacting to exogenous adaptation forces such as accounting for new business requirements that affect *where* along the cloud-edge continuum the event processing is performed. However, such a relocation profoundly modifies the existing pipeline deployment topology in terms of physical execution location of individual pipeline elements which potentially violates the stated criteria for event stream relays discussed in Section 7.3.3. Within this thesis, we focus on migrating stateless pipeline elements thereby explicitly leaving out state migration concepts which preserve processing semantics during migration [To et al. 2018; Brogi et al. 2018]. Similar to the reconfiguration, we illustrate the migration concepts on the basis of processors.

Following, we give an in-depth description on the run-time migration which broadly splits into an initial preparation performed within the migration handler and a step-wise execution of migration actions within the migration executor as depicted in Figure 8.3. Given the modified and configured pipeline $\mathcal{P}'$ containing a new processor node target, the migration handler is responsible for (1) computing the processor migration action, (2) triggering the migration executor to perform the step-wise migration and (3) proxying the migration status message back to the graphical user interface (not shown in Figure 8.3). Algorithm 6 summarizes the relevant preparation steps in the migration handler.

Algorithm 6 Compute and execute processor migration action within the migration handler

Input: modified and configured pipeline $\mathcal{P}'$
Output: migration status message

1: $\mathcal{V}'\backslash(A' \cup S'), \mathcal{V}_{old}\backslash(A_{old} \cup S_{old}) \leftarrow \emptyset$ ▷ initialize
2: $id \leftarrow$ getPipelineId($\mathcal{P}'$)
3: $\mathcal{P}_{old} \leftarrow$ queryPipelineById(id) ▷ get stored pipeline
4: $\mathcal{S}_r^{\text{old}} \leftarrow$ queryRelaysById(id) ▷ get stored relays
5: $\mathcal{S}_r^{\text{new}} \leftarrow$ generateRelays($\mathcal{P}'$) ▷ generate new relays
6: $\mathcal{V}'\backslash(A' \cup S') \leftarrow$ getProcessors($\mathcal{P}'$)
7: $\mathcal{V}_{old}\backslash(A_{old} \cup S_{old}) \leftarrow$ getProcessors($\mathcal{P}_{old}$)
8: **for each** $\rho_{new} \in \mathcal{V}'\backslash(A' \cup S')$ **do**
9: **for each** $\rho_{old} \in \mathcal{V}_{old}\backslash(A_{old} \cup S_{old})$ **do**
10: **if** matching(ρ_{new}, ρ_{old}) **and** dislocated(ρ_{new}, ρ_{old}) **then**
11: $s_{pred}^{\text{old}} \leftarrow$ evalPredRelay($\rho_{old}, \mathcal{S}_r^{\text{old}}, \mathcal{S}_r^{\text{new}}$) ▷ old predecessor relay
12: $s_{succ}^{\text{old}} \leftarrow$ evalSuccRelay($\rho_{old}, \mathcal{S}_r^{\text{old}}, \mathcal{S}_r^{\text{new}}$) ▷ old successor relay
13: $s_{pred}^{\text{new}} \leftarrow$ evalPredRelay($\rho_{new}, \mathcal{S}_r^{\text{new}}, \mathcal{S}_r^{\text{old}}$) ▷ new predecessor relay
14: $s_{succ}^{\text{new}} \leftarrow$ evalSuccRelay($\rho_{new}, \mathcal{S}_r^{\text{new}}, \mathcal{S}_r^{\text{old}}$) ▷ new successor relay
15: $a_{mig} \leftarrow new$ MigrationAction($\rho_{new}, \rho_{old}, s_{pred}^{\text{new}}, s_{pred}^{\text{old}}, s_{succ}^{\text{new}}, s_{succ}^{\text{old}}$)
16: $s_{mig} \leftarrow new$ MigrationExecutor(a_{mig}).$execute()$ ▷ start migration scheme
17: **if** successful(s_{mig}) **then** ▷ verify migration success
18: overwriteAndStore($\mathcal{P}', \mathcal{S}_r'$) ▷ store pipeline and relays
19: **return** migration successful, stored $\mathcal{P}'$ and $\mathcal{S}_r^{\text{new}}$
20: **else**
21: rollback($\mathcal{P}_{old}, \mathcal{S}_r^{\text{old}}$) ▷ initialize rollback
22: **return** migration failed, rolled back $\mathcal{P}_{old}$ and $\mathcal{S}_r^{\text{old}}$

At this point before elaborating on the preparation steps, it is noteworthy that whenever referred to "old", we mean the state of an entity (e.g., processor, relay) *before* the migration. In contrast, whenever referred to "new", we mean the state of an entity *after* the migration. In order to construct a migration action, the running pipeline and relay descriptions are first queried from the central database where they were persisted after finishing the

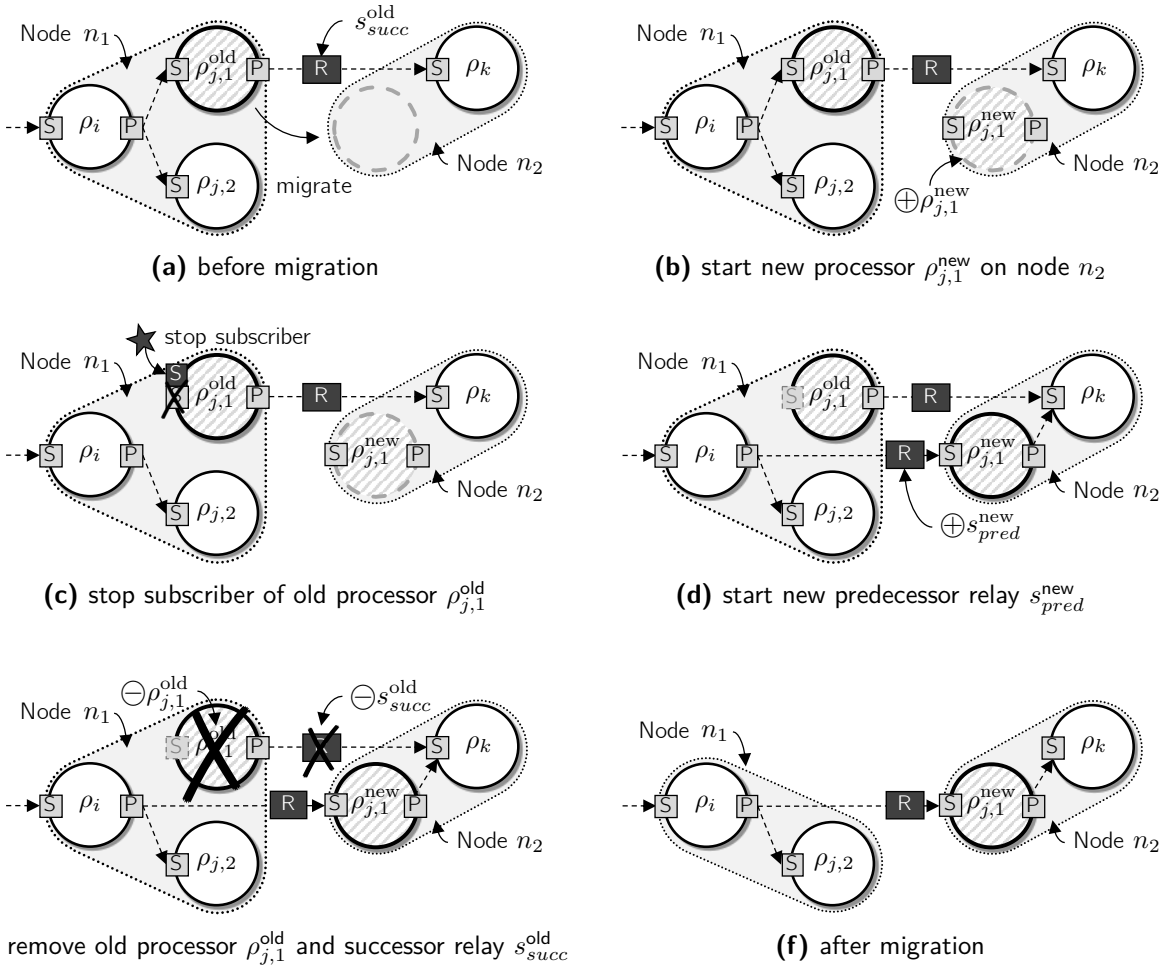
geo-distributed deployment. Next, a set of new event stream relays for the modified and configured pipeline $\mathcal{P}'$ are generated using Algorithm 4 from Section 7.3.3. Both the new pipeline description $\mathcal{P}'$ and the old one $\mathcal{P}_{old}$, as well as the new set of relay descriptions $\mathcal{S}_r'$ and the old one $\mathcal{S}_r^{\text{old}}$, build the foundation for comparing and assessing origin and target execution states prior and after the processor migration. Consequently, the list of processors from $\mathcal{P}'$ and $\mathcal{P}_{old}$ are traversed in order to first find the processor affected by the migration (Line 10).

Apart from identifying the processor affected by the migration, event stream relays need to be reevaluated to guarantee proper inter-node event dissemination provoked by the change of execution location. In particular in cases where preceding pipeline elements are multi-connected it is crucial to assure relay correctness. For instance, consider an adjacent and colocated predecessor-successor pair ($pred{-}succ$) prior to the migration, end up dislocated due to the $succ$ being migrated to another node. In principle, this fulfills the preliminary relay criteria and induces the need for instantiating a corresponding relay instance from the $pred$ to the new $succ$ node. Yet, only if it also complies with the uniqueness criteria. In cases where the $pred$ is multi-connected and the migrated $succ$ ends up neighboring to another present successor after the migration, no new relay is allowed to prevent run-time event duplicates. Similarly, an adjacent and dislocated $pred{-}succ$ pair prior to the migration, end up colocated afterwards. In such a case the previous relay instance might be removed, yet only if no other previously neighboring successor exists on the origin node which still requires this relay. Therefore, both the old predecessor and successor relays are evaluated in order to decide whether to keep or remove an already running relay instance while the new predecessor and successor relays are evaluated in order to assess if and where new relay instances need to be added (Lines 11 to 14). Next, a new migration action is created and delegated to the migration executor component responsible for performing the migration. The migration executor returns a status message containing information on the success or failure of the processor migration. This status message is verified and used to decide whether to overwrite and store new pipeline and relay descriptions or to initiate a rollback to the original execution state. Lastly, an overall migration status message is returned to inform the citizen technologist about the current state (Lines 17 to 22).

Upon receiving the migration action, the migration executor performs several consecutive steps aimed at sending out relay and processor invocation and detach requests in a coordinated fashion.

Therefore, we propose the following *step-wise migration scheme*:

1. Start new pipeline element ρ_{new} and new successor relays s_{succ}^{new} (if required)
2. Stop subscriber of old pipeline element ρ_{old} and await in-flight events to be processed
3. Remove old predecessor relay s_{pred}^{old} to old pipeline element (if any and permitted)
4. Start new predecessor relay s_{pred}^{new} to new pipeline element (if required)
5. Remove old pipeline element ρ_{old} and any old successor relays s_{succ}^{old} (if any)



(a) before migration

(b) start new processor $\rho^{\text{new}}_{j,1}$ on node n_2

(c) stop subscriber of old processor $\rho^{\text{old}}_{j,1}$

(d) start new predecessor relay s^{new}_{pred}

(e) remove old processor $\rho^{\text{old}}_{j,1}$ and successor relay s^{old}_{succ}

(f) after migration

Figure 8.6 Migration scheme in action: Excerpt of a running pipeline $\mathcal{P}{=}(\mathcal{V}, \mathcal{E}, \Pi)$ with two processors $\rho_i, \rho^{\text{old}}_{j,1} \in \mathcal{V}\backslash(A \cup S)$ and two sinks $\rho_{j,2}, \rho_k \in \mathcal{V}\backslash(A \cup P)$, directed event stream edges $\mathcal{E}$ (-➝) and running relay for inter-node communication between $\rho^{\text{old}}_{j,1}$ and its successor ρ_k. Processor $\rho^{\text{old}}_{j,1}$ is supposed to be migrated from edge node $n_1 \in \mathcal{N}_e$ to cloud node $n_2 \in \mathcal{N}_c$ shown in (a). Step-wise migration scheme for the processor is depicted in (b)-(e). Final pipeline execution topology after migration is shown in (f).

Figure 8.6 illustrates the migration scheme in action making use of the notation for pipeline element and relay invocation "⊕" and detach "⊖" requests introduced in Section 7.3.4. As indicated in the migration scheme, some of the steps may be skipped. This greatly depends on the origin and target pipeline execution topology and the involved node types as both determine whether or not relays need to be instantiated or removed.

In the following, the migration steps for processor $\rho^{\text{old}}_{j,1}$ are exemplified which is supposed to be migrated from edge node n_1 to cloud node n_2. In the given case, the excerpt of pipeline $\mathcal{P}$ consists of two processors ρ_i and $\rho^{\text{old}}_{j,1}$ and two sinks $\rho_{j,2}$ and ρ_k that are deployed over both nodes. Moreover, directed event stream edges are highlighted with

intra-node communication on the edge node apart from inter-node communication from edge to cloud node with one running event stream relay between the processor subject of the migration and its only successor (Figure 8.6a). In **step 1**, a corresponding pipeline element invocation request is sent to the node controller of the migration target node, here the cloud node, to start a new processor instance for the same processor type (Figure 8.6b). As the new processor ends up colocated to its only successor, new successor relays are not required. Next in **step 2**, similarly to the reconfiguration of processors, the adaptation gate and adaptation event concepts are used in the course of the migration for injecting migration instructions. As such, the node controller publishes the adaptation event on the adaptation gate of the old processor to stop the subscriber from retrieving published events from its predecessor while awaiting remaining in-flight events to be fully processed (Figure 8.6c). This step marks a crucial point of the migration scheme as it effectively interrupts the flow of events through the pipeline. Consequently, all successors, here the sink on the cloud node, do temporarily not receive new events which we refer to as the *downtime* timespan. The **step 3** is skipped as no old predecessor relay exists due to the initial colocation of the predecessor and the old processor. In **step 4** the downtime terminates with the instantiation of the new predecessor relay to the cloud node which reestablishes the event stream (Figure 8.6d). Finally, in **step 5**, all old instances that are no longer required are terminated, here the old processor as well as the old successor relay on the origin edge node (Figure 8.6e). This allows the transition into the final pipeline execution topology after the migration is completed (Figure 8.6f). The migration executor informs the migration handler about the overall migration status. If any of the described steps fail, the handler initiates a rollback to old pipeline execution topology.

After clarifying the adaptation types that result from user-initiated interactions based on external decisions, we focus on system-initiated adaptation measures in the following section. Therefore, we present an approach for pipeline element offloading as a result of internal decision-making based on system context knowledge.

8.3.3 Offloading

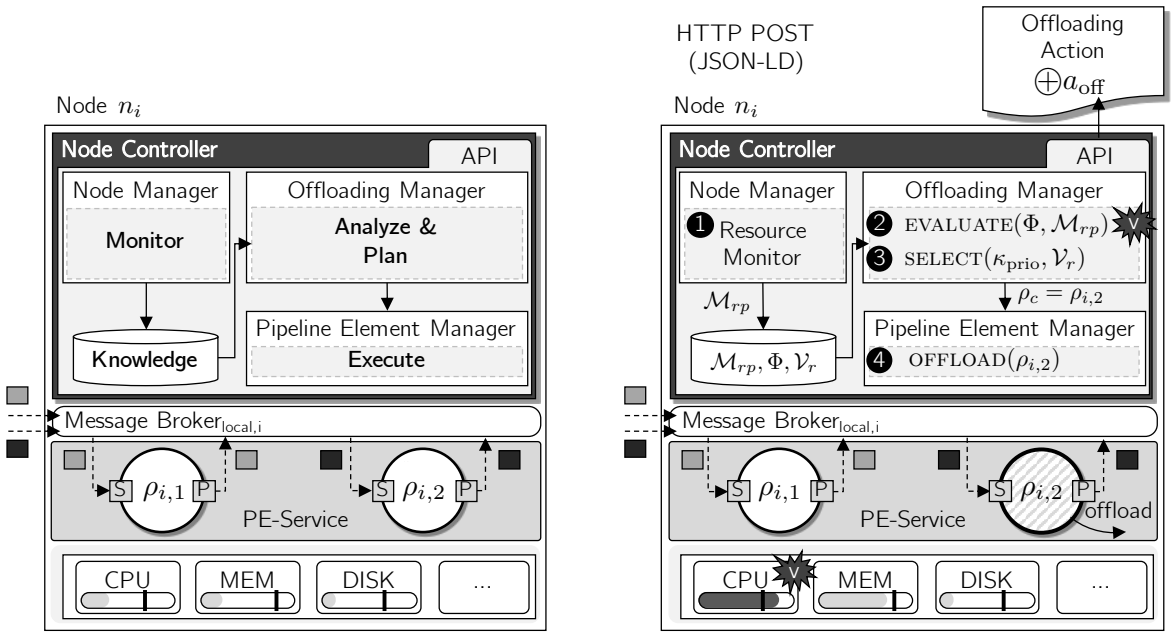
The dynamics and mobility aspects in fog environments in addition to shared computational resources constantly influence the pipeline execution topology at run-time and require to question initial user-specified deployment decisions. As these types of adaptation forces can hardly be evaluated by human, it is essential to provide certain autonomic mechanisms in response to impactful contextual changes in the node's operational and system context. Therefore, the context must be adequately monitored and assessed in order to first create situation awareness [Mozzaquatro et al. 2017] followed by conducting system-initiated adaptation actions. For instance, as computational resources of a node are typically shared among multiple pipeline elements of potentially multiple pipelines, it is inevitable to constantly observe the current utilization to assure healthy node operation and act accordingly upon detecting any deviations. A manifestation of such a

system-initiated adaptation action describes the *offloading* of a pipeline element from its origin node to a new target node, whereby the offloading decision itself is made by the node controller. In this context, *Monitor-Analyze-Plan-Execute* (MAPE) over a shared *knowledge* base, referred to as MAPE-K, is a well-recognized architectural pattern in autonomic computing for realizing self-adaptive systems [Kephart and Chess 2003; Brun et al. 2009]. In this thesis, we apply the MAPE-K approach as follows: (1) *monitor* and collect data over the system context and its environment, (2) *analyze* the gathered data and evaluate if an adaptation needs to be performed, (3) *plan* the adaptation actions to achieve a given goal, (4) *execute* the adaptation action, all while using a common and (5) shared *knowledge* base. The shared knowledge base maintains data, adaptation goals and additional state and is used during the MAPE phases. Hereafter, we focus on node resource utilization as this denotes a major impact factor driving the need for offloading, particularly on resource-constrained edge nodes. In addition, we focus on processors, as with the adaptation types discussed earlier. Yet, prior to elaborating on the act of offloading from a technical perspective, we need to first clarify *how* contextual changes can be detected and responded to in the first place which requires a generic concept.

Offloading Manifest. In order to realize offloading, a policy-driven approach is used to assess the utilization level of a node based on so-called *offloading manifests*. An offloading manifest comprises (1) a *resource property* to be observed, (2) an *evaluation policy* declaring a violation condition for the resource property and (3) a *selection strategy* to find a suitable offloading candidate. In addition, internal state and relevant context information, e.g., collected resource properties, the number of consecutive violations, or the defined preemption operation policy (see Section 7.3.1), are shared among the individual stages. Offloading manifests are globally registered on all participating nodes. From an architectural point of view, the node controller, concretely the offloading manager, periodically evaluates registered offloading manifests. The outcome of this evaluation determines whether or not an offloading action is required and, if so, an offloading candidate is selected. Only after having identified an offloading candidate, the pipeline element manager requests the central application management to proceed with the offloading. We follow the centrally coordinated approach to completing the offloading as this part relies on centrally hosted information on available, eligible nodes and existing pipeline descriptions used construct a new, modified pipeline description.

In summary, the proposed *offloading scheme* spans the following stages on the basis of the registered offloading manifests, whereby stages (1) to (4) are performed on local node level and stage (5) is coordinated from the central application management level:

1. Collect required metrics for resource properties
2. Detect and evaluate any violations using defined evaluation policies
3. Select offloading candidate using defined selection strategy
4. Send out offloading action to central offloading component
5. Find new eligible node, generate new configured pipeline, execute offloading



(a) MAPE-K **(b)** offloading action upon CPU threshold violation

Figure 8.7 Offloading scheme in action: Node n_i shows involved node controller components, two running processors $\rho_{i,1} \in \mathcal{V}_r$ part of pipeline $\mathcal{P}_1$ (preemption enabled, prio "high") and $\rho_{i,2} \in \mathcal{V}_r$ part of pipeline $\mathcal{P}_2$ (preemption enabled, prio "low"). MAPE-K constituents are depicted in (a). Offloading steps for $\rho_{i,2}$ are shown in (b) after violating the threshold-based CPU evaluation policy $\lambda_{cpu}^{\text{thold}}$ from the offloading manifest Φ, using resource property metrics $\mathcal{M}_{rp}$, running processor instance set $\mathcal{V}_r$ and a priority selection strategy κ_{prio}.

Next, we give an in-depth description of the offloading scheme and elaborate on each of the individual steps while commenting on the role of all involved architecture components from the node level to the central application management level. Figure 8.7 illustrates the offloading scheme in action showing all relevant stages on the node level. Consequently, the node controller employs the MAPE-K pattern which is distributed over several components, namely the node manager, the offloading manager, the pipeline element manager and an in-memory database (Figure 8.7a). Starting with **step 1**, the resource monitor constantly observes the node resource usage and collects run-time metrics for all resource offers at regular intervals. Latest resource offers are stored alongside computed resource property metrics $\mathcal{M}_{rp}$ (e.g., CPU utilization) in the in-memory database which also contains registered user-specified offloading manifests Φ and running processor descriptions $\mathcal{V}_r$. The knowledge base provides the foundation for all the following steps. In **step 2**, the offloading manager performs the evaluation of all offloading manifests at regular intervals. Algorithm 7 summarizes the relevant steps. The evaluation policies are assessed in order to detect any violations in a timely manner. Thereby, the relevant resource property metric is extracted from the set of collected metrics $\mathcal{M}_{rp}$, e.g., CPU utilization, and added to a metrics history set which is leveraged when formulating

Algorithm 7 Evaluate registered offloading manifests

Input: A set of offloading manifests Φ, a set of collected resource property metrics $\mathcal{M}_{rp}$

```
1: for each om ∈ Φ do
2:     λ_rp ← getEvaluationPolicy(om)              ▷ get evaluation policy
3:     m_rp ← extractMetricType(om, M_rp)          ▷ get metric m_rp ∈ (0, 1)
4:     λ_rp.addMetric(m_rp)                        ▷ add to metrics history set M_h
5:     if λ_rp.isViolated() then                   ▷ check evaluation policy
6:         κ ← getSelectionStrategy(om)            ▷ get selection strategy
7:         V_r ← ∅
8:         V_r ← getRunningProcessors()            ▷ get running processors
9:         ρ_c ← select(κ, V_r)                    ▷ trigger candidate selection
10:        offload(ρ_c)                            ▷ trigger offloading
```

certain evaluation conditions regarding the definition of evaluation policies. As a result, such a time-series for a dedicated metric can be used to detect possible trends apart from mitigating bursty resource spikes. Upon detecting a policy violation, the offloading scheme proceeds by first extracting the specified selection strategy followed by retrieving all currently running processor instances on the underlying node (Lines 6 to 8). These information are then used to trigger the candidate selection in **step 3** and subsequently the offloading in **step 4**. If no violation for any registered offloading manifest occurs, the overall procedure is terminated until the next evaluation interval starts.

Though the concepts of offloading manifests as well as its evaluation policies and selection strategies are generic to widen the design space, within this thesis, we refer to concrete realizations aligned with Figure 8.7b. Therefore, Algorithm 8 excerpts a *threshold-based* evaluation policy, where resource property metrics m_{rp} are compared to statically defined thresholds T_{rp} that result in a violation if the threshold is exceeded. Further, a maximum number of permissible violations N_ϑ is used to prevent costly offloadings due to single outliers.

Algorithm 8 Threshold-based evaluation policy $\lambda_{rp}^{\mathsf{thold}}$

Input: Resource property threshold $T_{rp} \in (0, 1)$, max. permissible violations $N_\vartheta \in \mathbb{N}_{>0}$
Output: $true$ if policy is violated, else $false$

```
1: ϑ ← 0                                          ▷ initialize
2: M_h ← getMetrics()                             ▷ get metrics history set
3: for each m_rp ∈ M_h do
4:     if m_rp > T_rp then                        ▷ exceeding threshold condition
5:         ϑ ← ϑ + 1
6: return ϑ > N_ϑ                                 ▷ exceeding violations condition
```

Example. *Figure 8.7b uses a threshold-based CPU evaluation policy with a configured CPU utilization threshold T_{cpu} and a maximum number of permissible N_ϑ. Therefore, collected CPU utilization metrics m_{cpu} are compared to the given threshold while increasing the number of detected violations ϑ for $m_{cpu} > T_{cpu}$. When $N_\vartheta + 1$ violations are detected within the CPU metrics history set, the overall policy is violated.*

After having detected that an offloading is required, a suitable offloading candidate needs to be found which is done using a specific selection strategy. An example for such a selection strategy is shown in Algorithm 9 which illustrates a *priority-based* approach. The overall objective behind the priority selection strategy is to find the lowest-prioritized processor leveraging the operation policy for preemption as described in Section 7.3.1. If preemption enabled by citizen technologists, information on the configured priority class are retrieved and leveraged to find lowest-prioritized processors for all currently running processor instances on the present node. Concretely, upon receiving a set of all currently running processors $\mathcal{V}_r$ as a preliminary step, a dedicated offloading candidate set $\mathcal{V}_c$ is populated by inspecting the preemption operation policy of the running processors. As some pipelines may have not been configured to use preemption, corresponding processors are assumed low-prioritized, such that their priority class is updated accordingly. Eventually, the lowest-prioritized processor from the offloading candidate set is returned.

Algorithm 9 Priority selection strategy κ_{prio}

Input: A set of all currently running processors $\mathcal{V}_r$
Output: Offloading candidate ρ_c

1: $\mathcal{V}_c \leftarrow \emptyset$ ▷ initialize
2: **for each** $\rho \in \mathcal{V}_r$ **do**
3: $\pi_o \leftarrow$ getOperationPolicy(ρ) ▷ get operation policy
4: **if** $\neg$preempt(π_o) **and** prio(π_o) $= null$ **then** ▷ check preemption setting
5: $\rho \leftarrow$ setPrioClass(low) ▷ treat as low-prioritized
6: $\mathcal{V}_c \leftarrow \mathcal{V}_c \cup \rho$ ▷ append to candidate set
7: **return** selectLowestPrioritized($\mathcal{V}_c$)

Example. *Figure 8.7b uses a priority selection strategy κ_{prio} to select a lowest-prioritized processor offloading candidate from the set of running processors $\mathcal{V}_r = \{\rho_{i,1}, \rho_{i,2}\}$. The two processors are part of two pipelines $\mathcal{P}_1$ (preemption enabled, prio "high") and $\mathcal{P}_2$ (preemption enabled, prio "low"). After populating the offloading candidate set $\mathcal{V}_c$, with $\mathcal{V}_c = \mathcal{V}_r$, the lowest-prioritized processor $\rho_{i,2}$ from pipeline $\mathcal{P}_2$ is selected and used for offloading.*

The selected offloading candidate is passed to the pipeline element manager which constructs a corresponding offloading action a_{off} and sends it to the central offloading component within the application management layer to finalize the offloading scheme as shown in Figure 8.7b.

Next, we describe the final **step 5** to complete the offloading at the central level as illustrated in Figure 8.3. This involves three sub-tasks, namely (1) finding a new eligible node, (2) generating a new configured pipeline and (3) executing the offloading. The idea is to reduce the offloading problem into a migration problem allowing to reuse existing components and algorithms. Thus, after receiving the offloading action request, the offloading handler performs all necessary steps summarized in Algorithm 10.

Algorithm 10 Completing offloading within central offloading handler

Input: Offloading action a_{off}
Output: Offloading status message

1: $\mathcal{P}'_c, \mathcal{P}_{old}, \mathcal{N}_{el,\rho} \leftarrow \emptyset$ ▷ initialize
2: $\rho_c \leftarrow$ extractProcessor(a_{off})
3: $\mathcal{P}_{old} \leftarrow$ findAssocPipeline(ρ_c) ▷ find associated pipeline
4: $\mathcal{T} \leftarrow$ getNodeTags$(\mathcal{P}_{old})$ ▷ get node tags
5: $\mathcal{N}_{el,\rho} \leftarrow new$ NodeValidator$(\rho_c, \mathcal{T}).validate()$ ▷ call Algorithm 2
6: $n' \leftarrow$ selectNodeCandidate$(\mathcal{N}_{el,\rho})$ ▷ random node selection
7: $\mathcal{Z}' \leftarrow$ updateMapping$((\rho_c, n'), \mathcal{P}_{old})$ ▷ update mappings
8: $\mathcal{P}'_c \leftarrow new$ PipelineGenerator$(\mathcal{Z}', \mathcal{P}_{old}).generate()$ ▷ call Algorithm 3
9: $s_{mig} \leftarrow new$ MigrationHandler$(\mathcal{P}'_c).handle()$ ▷ call Algorithm 6
10: **if** successful(s_{mig}) **then**
11: **return** offloading successful
12: **else**
13: **return** offloading failed, blacklist ρ_c and try another processor

First, the offloading candidate is used to find its associated pipeline which is used to extract declared node tags (if any). Consequently, the validation procedure employed in Algorithm 2 (see Section 7.3.2) can be leveraged to find a set of eligible node targets. It is worthy to note, that the current node is not considered eligible anymore as it lacks sufficient resource offers. From the set of eligible nodes, a new node candidate must be found according to a dedicated node selection strategy. Similarly to selection strategies for offloading candidates, the concept behind node candidate selection strategies is generic and allows for arbitrary procedures to be realized. In this thesis, we apply a random node selection from the set of eligible nodes. Accordingly, the declared pipeline element-node mapping by citizen technologists is now updated with the new target node. This finalizes essential preparations to complete the pipeline configuration by calling the pipeline generator component which employs Algorithm 3 (see Section 7.3.2). Lastly, the modified and configured pipeline is passed to the migration handler to perform the relocation task for the offloading candidate according to the step-wise migration scheme. After the migration operation is successfully performed, the offloading is considered complete and is acknowledged to the issuing node controller instance by sending an offloading status message. In case of a failure, the node controller is informed to blacklist the offloading candidate and select another one which restarts the process.

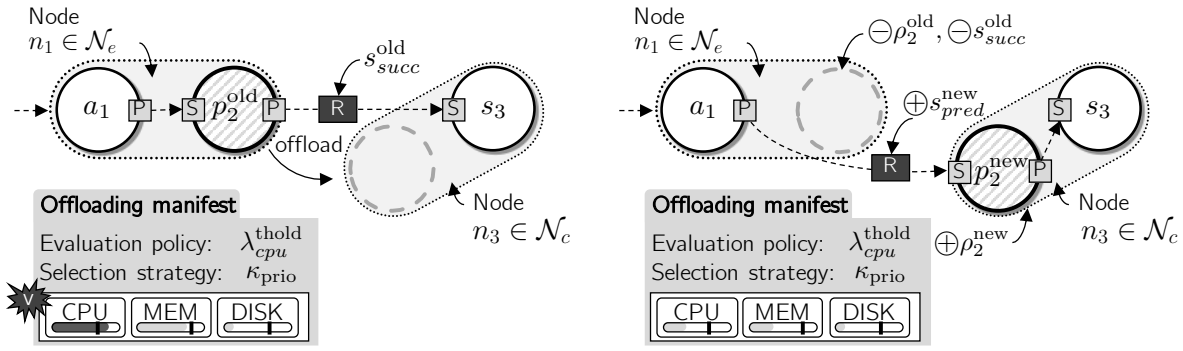


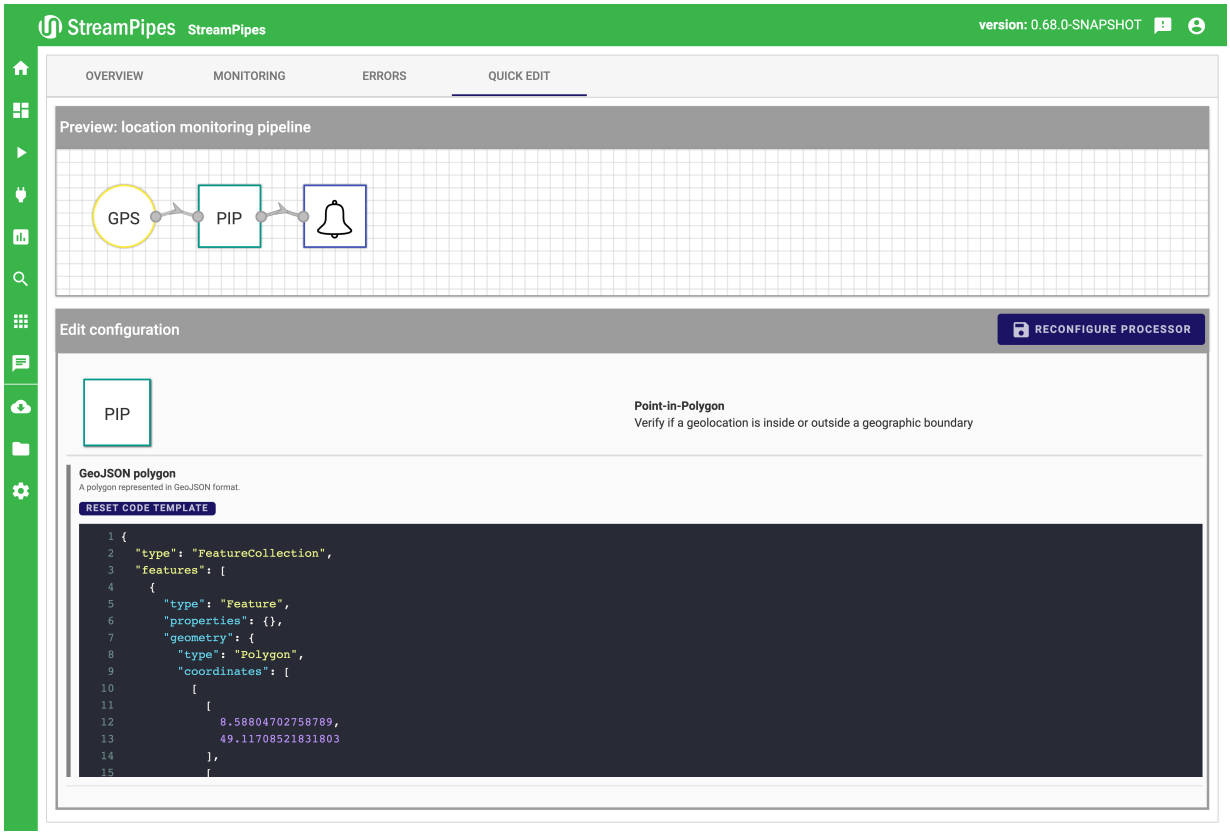
Figure 8.8 Running example: Offloading manifest and run-time offloading of point-in-polygon processor

Example. *Figure 8.8 shows the location monitoring pipeline example deployed on edge node n_1 and cloud node n_3. The offloading manifest entails a threshold-based CPU evaluation policy λ_{cpu}^{thold} (threshold $T_{cpu} = 80\%$, max. violations $N_\vartheta = 3$), and a priority selection strategy κ_{prio}. After the policy is violated (for 4 violations), the point-in-polygon processor is selected, and, hence offloaded to the cloud node n_3 while respecting the new pipeline execution topology (Figure 8.8a). After the offloading is completed, excessive CPU utilization on n_1 is reduced (Figure 8.8b).*

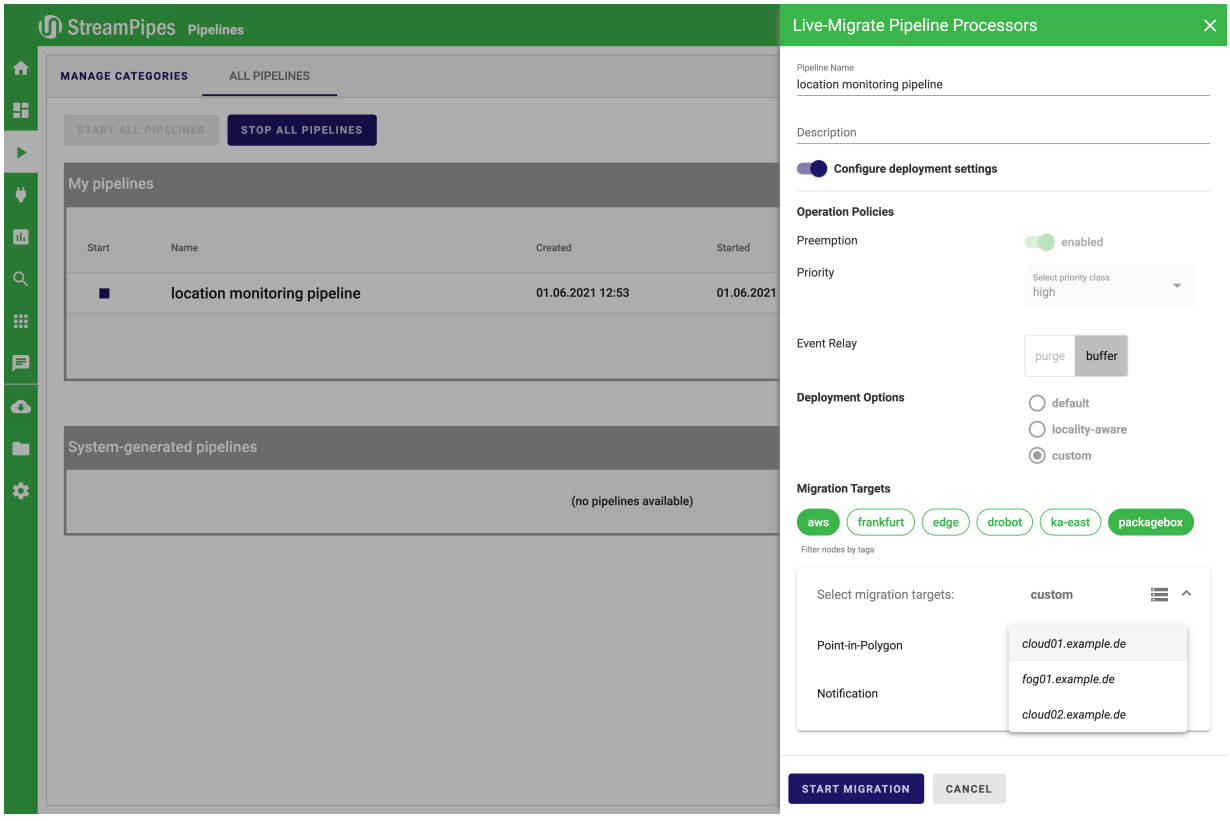
8.4 Tools

The concepts of this chapter are integrated into the Apache StreamPipes project to allow geo-distributed event-driven applications to support run-time evolutions for all discussed adaptation types. Therefore, Figure 8.9 gives an overview of the graphical support for citizen technologists, yet focusing on user-initiated reconfiguration and migration actions as the offloading is managed system-internally. As requirements change over time, citizen technologists such as the fleet operators for delivery robots can easily initiate run-time adaptation actions from the graphical user interface for both reconfigurations and migrations while being abstracted from any technical details in the application management layer. Thereby, fleet operators can update the valid operation area employed as a reconfigurable geofence property in the point-in-polygon processor. Concretely, they can provide an updated geofence version in the form of a GeoJSON[1], a widely adopted format for encoding geographic data structures and known to fleet operators (Figure 8.9a). For live migration, fleet operators can change deployment target nodes for processors on-the-fly to account for flexibly decide where certain event processing must be performed (Figure 8.9b). Overall, the comprehensive tool support for all adaptation types enables to operate mission-critical event-driven applications to account for business-driven changes and the dynamics in fog environments.

[1]https://geojson.org/



(a) Processor reconfiguration at run-time



(b) Processor migration at run-time

Figure 8.9 Pipeline evolution at run-time: Tool support

8.5 Summary

In this chapter, we presented our conceptual work with regard to run-time adaptations of processing pipelines in view of Research Question 3. The main contribution is an adaptation methodology for event-driven applications which attaches to the sense-process-respond principles of event-driven architectures. We first elaborated on the necessity for supporting pipeline evolutions aspects in general.Therefore, we approached the conceptual considerations by stating two decisive adaptation dimensions, namely adaptation types and decision-making origin, which both underpin our proposed adaptation methodology as described in Section 8.2. We further introduced adaptation gates and adaptation events abstractions on the basis of the event processing pattern for applying the adaptation methodology to processing pipelines which allow to pass configuration and control directives to pipeline elements at run-time. In view of the adaptation types, we described three run-time evolutions for event-driven applications to account for both exogenous and endogenous adaptation forces in Section 8.3. First, run-time reconfigurations on reconfigurable static properties of pipeline elements allow to alter initial design-time decisions regarding the event processing logic which facilitate citizen technologists to dynamically change the run-time behavior of event-driven applications. Second, we introduced a step-wise migration scheme which is capable of adapting initial pipeline execution topologies to account for new business requirements in order to dynamically displace certain pipeline elements from one node execution location to another one anywhere along the cloud-edge continuum. Third, we presented an offloading scheme for decision-making on node level. Influenced by the MAPE-K pattern, this allows node controllers to observe the operational and system context of the underlying node based on resource utilization metrics in order to deduce run-time offloading actions on dedicated pipeline elements in a self-aware manner. In this respect, we proposed the generic concept of offloading manifests. Offloading manifests comprise evaluation policies for defining violating conditions on resource property metrics and selection strategies to subsequently scope specific offloading candidates. Subsequently, offloading candidates are offloaded in coordination with the central application management by reducing the offloading into a migration problem. Finally, we presented necessary tool support for citizen technologists to perform run-time adaptations on geo-distributed pipelines in fog infrastructures to facilitate constant pipeline evolution. In this respect, all concepts from this chapter were integrated into the Apache StreamPipes project.

Part IV

Finale

9 Evaluation

In this chapter, we evaluate our proposed approach for a holistic application management of event-driven applications in heterogeneous fog computing infrastructures in view of all research questions. Evaluations are performed based on implemented artifacts for the discussed models, concepts and methods in Chapters 6 and 7 as well as Chapter 8. In Section 9.1, we first define an overall *evaluation framework* which systematically structures our conducted evaluations before describing developed software artifacts of the reference implementations. Section 9.2 presents two *case studies* covering practical experiences of applying our concepts in real-world deployments to evaluate the applicability in typical IIoT domains. Section 9.3 assesses the fulfillment of requirements by means of a *conceptual investigation* with regard to postulated research questions and identified needs. Lastly, we perform *performance tests* covering operation and adaptation aspects in Section 9.4.

9.1 Evaluation Framework

Within this thesis, we gradually introduced concepts to tackle our stated research questions. Thereby, our research aims at achieving a holistic application management approach for geo-distributed event-driven applications in fog environments that targets non-technical audience such as citizen technologists and allows to enhance data and analytics democratization from IT expert to knowledge workers. From foundational modeling efforts for enhanced resource exploitation and a geo-distributed architectural design in Chapter 6, over pipeline deployment and operation aspects in Chapter 7, to pipeline adaptations facilitating the application evolution in Chapter 8. Consequently, the pursued goal of the evaluation is to assess whether our contributions are able to solve the identified problems and needs in newly emerging application domains such as the IIoT. As our underlying research methodology is based on the design science paradigm for information systems, evaluating the derived artifacts is an essential task. To this extent, a systematic and comprehensive evaluation framework is required. The evaluation framework considers various dimensions and viewpoints, uses specific *evaluation metrics* and respective *evaluation methods* in order to investigate the designated *evaluation artifact*. Figure 9.1 gives an overview of our evaluation framework.

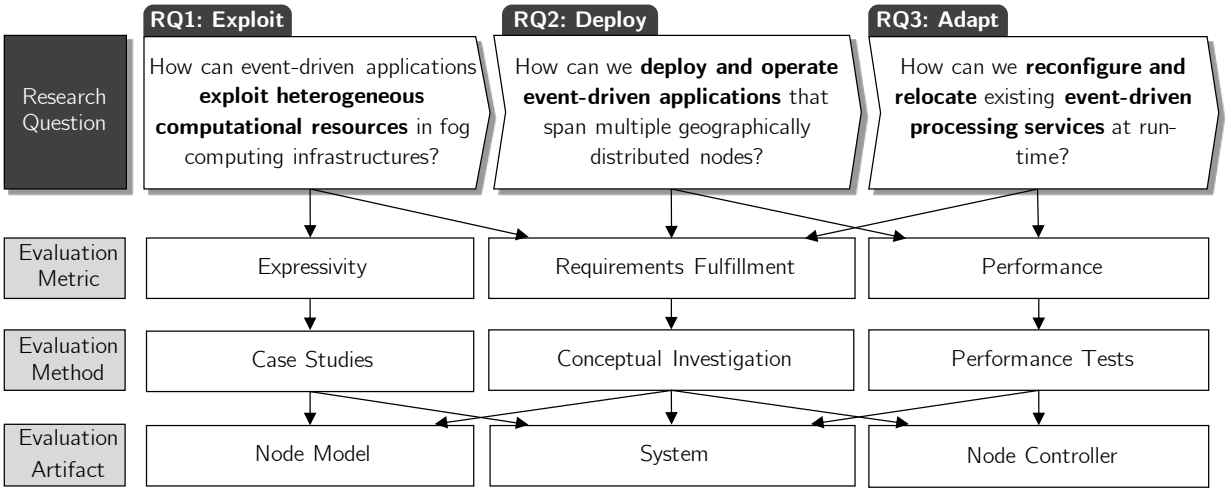


Figure 9.1 Evaluation framework: Overview

First, we use *case studies* to reflect emerging applicational and organizational needs from Chapter 3. We demonstrate the practical applicability and expressivity of our concepts behind the node model and methods employed within the developed system by presenting experiences gathered in real-world deployments. Second, elicited requirements from Chapter 5 are analyzed to evaluate the fulfillment level of all derived artifacts in terms of a *conceptual investigation*. As we specifically target holistic application management tasks regarding the deployment, operation and adaptation of geo-distributed event-driven applications in fog computing infrastructures, it is crucial to conduct performance evaluations of our developed system. Therefore, extensive *performance tests* over various scenarios are conducted for the developed system and the node controller leveraging a real-world fog computing testbed. This allows to obtain insights on the run-time performance with regard to operation and adaptation aspects and provide the foundation to draw conclusions on potential limitations of our concepts and methods.

Software Artifacts. While our introduced models, concepts and methods for managing geo-distributed event-driven applications in heterogeneous fog computing infrastructures are language and tool-independent, we provide a reference implementation of all presented architectural components as an extension for the graphical flow-based open-source project Apache StreamPipes. At its core, StreamPipes natively builds on top of the EPA and adapter vocabulary which we reuse and extend as part of our node model as stated in Section 6.4. All components are developed in Java, packaged as Maven[1] modules and distributed as multi-architecture Docker images in order to facilitate platform-agnostic deployment and operation of respective containerized node controller, pipeline element, message broker and other relevant services regardless of the present CPU architecture.

[1]https://maven.apache.org/

In brief, all our introduced concepts are either developed as standalone core modules or seamlessly integrated as extensions to employ geo-distributed event-driven application management capabilities in StreamPipes which we summarize in the following:

- *Models*—The introduced node model and related concepts from Section 6.4 are added and integrated to the existing knowledge base of StreamPipes.
- *Node management*—The proposed node management functionality from Section 6.5 is implemented and added to the core of StreamPipes.
- *Pipeline management*—The presented components for geo-distribution and event stream relay management from Section 7.3 and run-time evolution from Section 8.3 are implemented to complement existing core management functionality.
- *Node controller*—All node controller components elaborated in Section 7.3.5 are implemented and bundled as a standalone module to be deployed as a containerized service on each participating node in the fog computing infrastructure.

The aforementioned components built the foundation of the holistic geo-distributed application management in StreamPipes. Yet, additional adaptations are necessary with regard to our targeted citizen technologist and related peripheral organizational roles involved in the pipeline element development process.

- *Web application*—The web frontend is extended to encompass a node overview and management interface shown in Section 6.6, advanced pipeline deployment and operation options illustrated in Section 7.4, and reconfiguration and migration interfaces depicted in Section 8.4.
- *Run-time wrapper*—A new Java run-time wrapper for processors is developed which entails the proposed adaptation methodology from Section 8.2.
- *Software development kit* (SDK)—The SDK is extended to enable developers to declare node resource requirements and reconfigurable static properties.

9.2 Case Studies

In order to evaluate our developed approach for managing geo-distributed processing pipelines from a practical viewpoint, we applied our concepts in various real-world deployments within the IIoT domain by leveraging the developed tool extensions within Apache StreamPipes. Following, we elaborate on the use cases and present experiences drawn from the real-world setup with regard to the expressivity of the proposed models and general applicability of the concepts and methods employed in the architectural design of the system building upon these. First, we present a product quality inspection use case leveraging a collaborative robot in an automated assembly process within the manufacturing domain in Section 9.2.1. Second, we illustrate a remote condition monitoring use case for autonomous delivery robot platforms in Section 9.2.2.

9.2.1 Case Study 1: Cobot-based Product Quality Inspection

The first use case was performed in cooperation with an industrial manufacturing company. Hereby, a collaborative robot (so-called *cobot*) was used for autonomous product assembly. To this extent, the cobot was equipped with a smart gripper module with the ability to measure gripping forces which in turn allows to evaluate the permissible force within the spring mechanism of a dedicated part. Sensor measurements of the gripper module (*gripper event stream*) as well as the cobot's internal state machine (*state event stream*) are accessed via the *Robot Operating System* (ROS) for further analyses. While the former denotes gripping events with essential information on the measured force at a specific time instant, the latter represents discrete state change events to obtain knowledge on respective cobot execution steps, e.g., start and end signals for the quality check. Figure 9.2 illustrates the force events, the active test periods, the specified quality threshold, and the computed average force during the assembly in a set up, configurable testbed. The testbed allows for repeatably conducted quality checks on 4 test parts whereby 3 parts are "ok" and 1 part is "not ok" due to a defective spring mechanism.

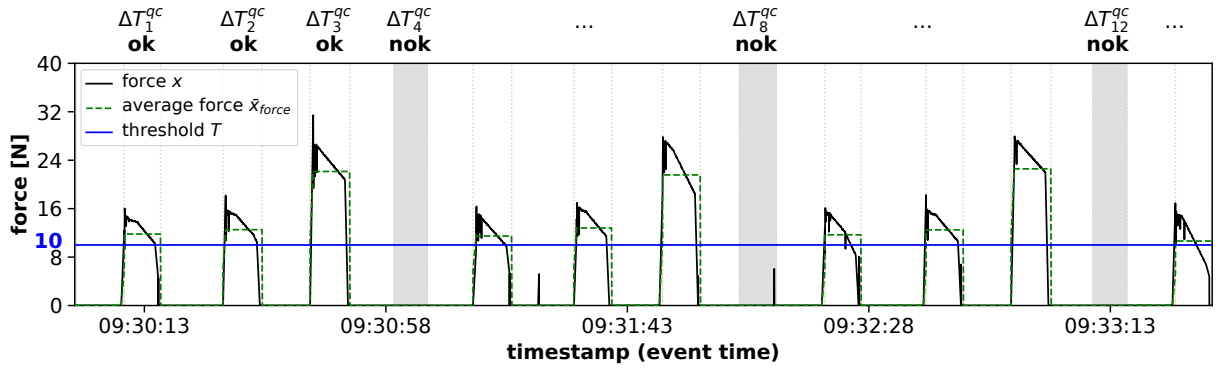
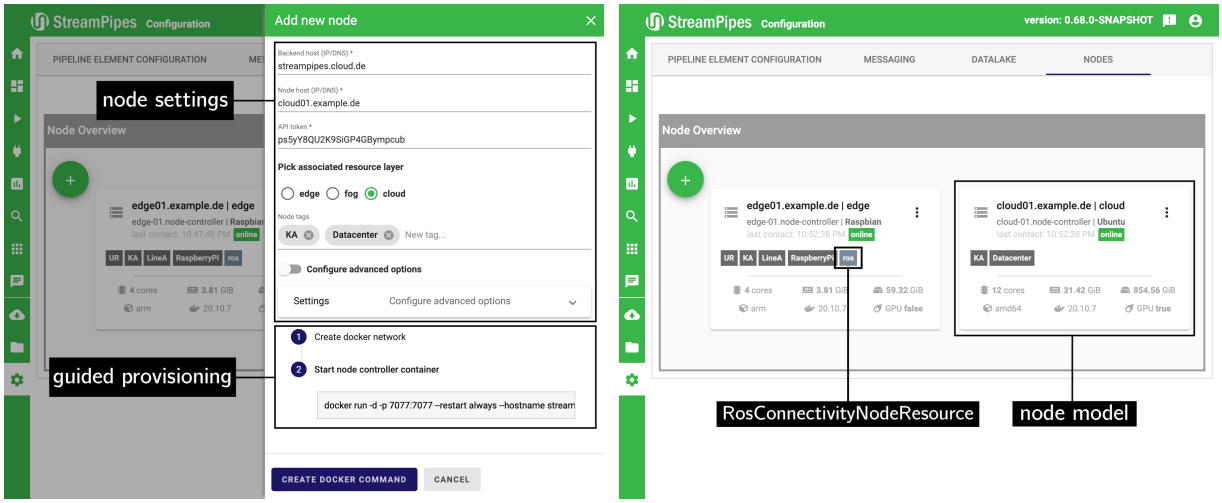


Figure 9.2 Excerpt of performed cobot-based quality checks. Time series shows force events over active and inactive test periods. Defined criterion for quality assessment follows a rule-based approach, i.e., the average force within the test period ΔT^{qc} must exceed a threshold T set by the quality engineer, s.t. $\bar{x}_{\text{force}} > T$, for $T{=}10\ N$ (parts $1, 2, 3, 4$). Highlighted time intervals indicate "not ok" results for defective part 4.

Thus, the goals are as follows:

- *Applicational needs*—First, the spring mechanism must be quickly assessed during pick-and-place operation using a rule-based approach. Therefore, both the gripper and the state event stream are leveraged while results are immediately sent back to the cobot. Second, relevant key performance indicators must be computed for monitoring purposes and to identify adjusting levers for process optimization.
- *Organizational needs*—Allow citizen technologists from the company's quality department to flexibly deploy, operate and adapt analytical product quality inspection pipelines in a geo-distributed fashion to meet business and application requirements towards low latency, data sovereignty and data locality.

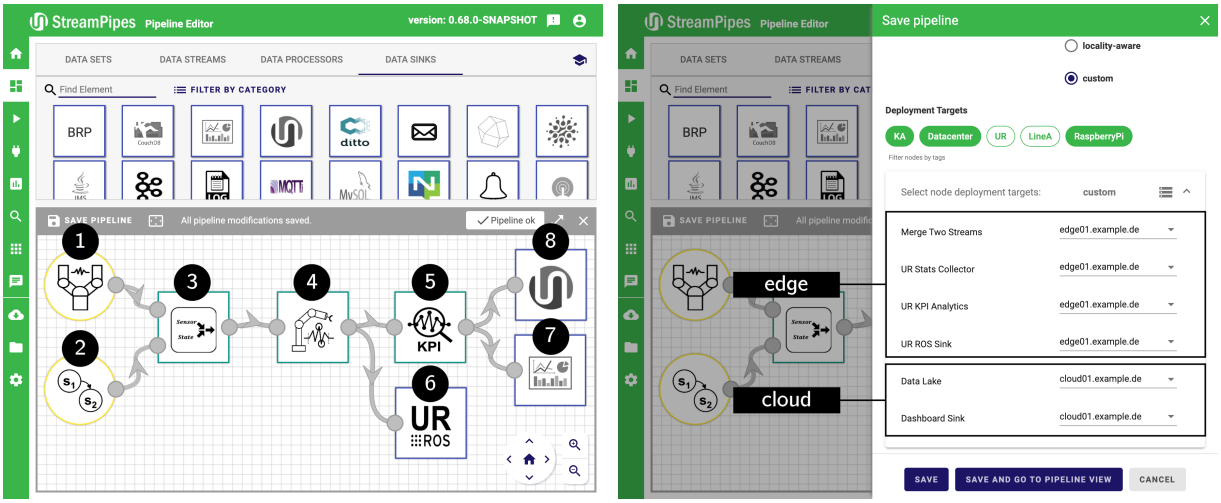


(a) add new node **(b)** node overview with newly registered node model

Figure 9.3 Setup phase: Adding new cloud node with node tags to extend the fog computing infrastructure managed by StreamPipes (a). Node overview after registration shows two nodes with one edge node (Raspberry Pi 4) and one cloud node (VM) including excerpts of the node model such as node resource properties and node metadata (b).

Setup Phase. One of the first steps was to provision the infrastructure containing available compute nodes with our node controller in order to exploit their exposed computational resources for executing potential pipeline elements. In order to provide sufficient user guidance through the provisioning step of new nodes, the node overview in the GUI enacts a step-wise procedure to add new nodes. This allows to easily configure relevant installation settings of the node controller which are translated in a ready-to-use deployment instruction to start the respective container on the desired destination node. In view of the task, we provisioned an ARM32-based Raspberry Pi 4 as an edge node in close proximity to the cobot and a x86-based virtual machine denoting a cloud node within the company's own datacenter. While we configured the Raspberry Pi including relevant node tags[2], i.e., "UR", "KA", "LineA", "RaspberryPi", and a RosConnectivityNodeResource to demonstrate relevant steps, the partner was able to perform necessary configurations and integrate the cloud node as shown in Figure 9.3a. The node controller constructed and registered the NodeDescription and self-reliantly managed the pipeline element and node broker container deployments in a platform-agnostic fashion. Therefore, it leveraged encompassed knowledge in the node description about the associated resource layer, i.e., edge, fog, or cloud, in addition to the generic DeploymentContainer descriptions for containerized services to decide which services needs to be started. Figure 9.3b gives an overview of available computational resources of the fog infrastructure and presents an excerpt of the underlying node model in terms of NodeResource information for all resource types and additional NodeMetadata.

[2]Due to confidentiality agreements node tags were altered for this thesis.



(a) pipeline authoring **(b)** preference-based configuration

Figure 9.4 Operation phase: Product quality inspection and KPI analytics pipeline $\mathcal{P}_{\text{Q,KPI}}$ during pipeline authoring (a) and when specifying preference-based configurations prior to geo-distributed pipeline deployment (b).

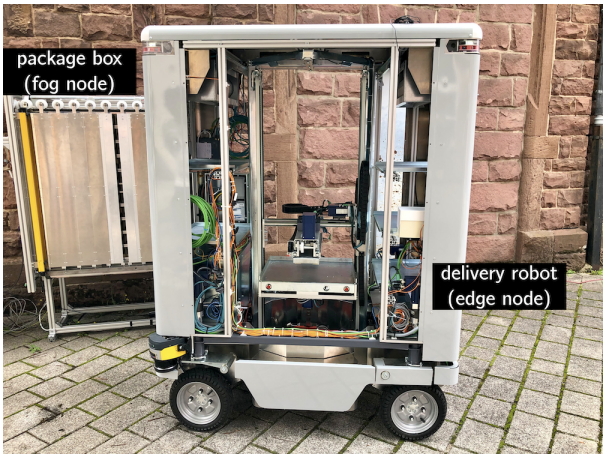
Operation Phase. In order to realize these goals, the problem description was decomposed into the respective cobot-based product quality inspection and KPI analytics pipeline, referred to as $\mathcal{P}_{\text{Q,KPI}}$, and shown in Figure 9.4a. This pipeline implies a set of eight pipeline elements which we developed and were used by the quality engineer during pipeline authoring: ❶ a *ROS* adapter for the *gripper event stream*, ❷ a *ROS* adapter for the *state event stream*, ❸ a processor for *merging* two event streams which provides an enriched force event stream encompassing knowledge on the current active cobot execution state, ❹ a *statistics and quality check* processor which buffers the enriched force time series over the quality check time window. Once the time window ends, descriptive statistics are computed, chief among the average force $\bar{x}_{\text{force}}$, to perform the quality check based on a reconfigurable static threshold property, ❺ a *KPI* processor for computing relevant quality and process metrics, e.g., scrap rate, first pass yield, average quality check duration, ❻ a *ROS* sink to send validation results to the cobot, ❼ a live *dashboard* sink to visualize KPI in real-time and ❽ a *datalake* sink to store analytics results for traceability purposes.

Prior to the geo-distributed deployment, we instructed the partner to carry out the preference-based configuration to prepare the deployment and operation options for $\mathcal{P}_{\text{Q,KPI}}$. Figure 9.4b illustrates the chosen "custom" deployment option and related pipeline element node mappings for processors and sinks while only eligible deployment target nodes were exposed to the quality engineer. Therefore, employed pipeline element resource requirements were validated and compared against the current resource offers. In view of this use case, hardware resource requirements were satisfiable by both edge and cloud node. In contrast, connectivity requirements for both ROS adapters and the

ROS sink were only fulfilled by the edge node due to the RosConnectivityNodeResource which was retrieved from the respective node model. This provided further support to the quality engineer when selecting deployment target nodes. Regarding the deployment, it is worthy to note that both ROS adapters were already deployed on the edge node in a previous connection step within the connect module of StreamPipes (not shown in the figure). Nevertheless, the logical representation as a registered event streams could be used by the quality engineer during the pipeline authoring process. Due to the manual node assignment, it was possible to explicitly adhere to related business and application requirements to prevent excessive edge-cloud round trips. Hence, apart from the adapters themselves, all processors including the ROS sink were deployed on the edge node and thus rely on intra-node event dissemination based on the lightweight epa:MqttTransportProtocol. Consequently, only pre-computed and aggregated production and quality KPIs were forwarded to the epa:KafkaTransportProtocol of the cloud message broker by means of the EventStreamRelay concept. From here, the dashboard and datalake sink on the cloud node were able to subscribe the provided results for central monitoring and long-term storage purposes. Besides, the reconfigurable threshold parameter based on the ReconfigurableStaticProperty concept provided the ability to update defined tolerable quality specifications at run-time. This was inevitable to incorporate adaptive behavior on the event processing side to supplement flexibility demands of cobot-based parts assembly.

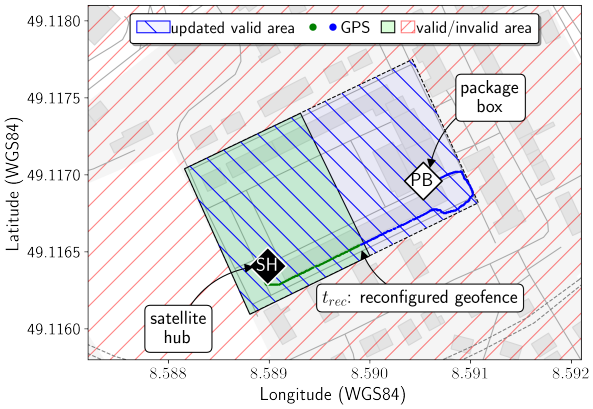
9.2.2 Case Study 2: Autonomous Delivery Robot Platform

The second use case was conducted in a consortium of academic and industry partners as part of a collaborative research project from where we drew the motivating smart urban logistics scenario presented in Section 3.1.2. In brief, the overall objective was to investigate issues related to the last-mile delivery problem by using autonomous delivery robot platforms for package delivery. Thereby, satellite hubs outside of city centers can be seen as handover points where delivery service provider handover respective packages to these delivery robot platforms. In turn, delivery robots autonomously execute delivery tours in urban city areas to deliver packages to available package boxes denoting end customer pick-up points. Figure 9.5 shows the developed prototypes for both the autonomous delivery robot platform (mobile edge node) and the package box (regional fog node). The delivery robot prototype was equipped with various sensor modules generating different sensor measurements (*system event stream*) which were provided over an integrated ROS interface. System events include information on the current battery state, acceleration, speed, heading, or the current GPS location at a specific time instant. Further, a dedicated test area situated on the company premises of one of our partners was set up which provided a real-world test environment for multiple experiments. This allowed to obtain real-world experience of the proposed models, concepts and methods along the whole event-driven application life cycle under the influence of mobility aspects in fog computing.

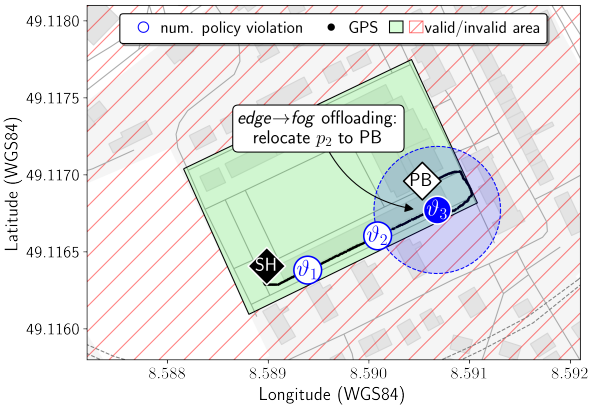


(a) delivery robot platform (front), package box (back)

(b) delivery robot in action



(c) Reconfigure geofence

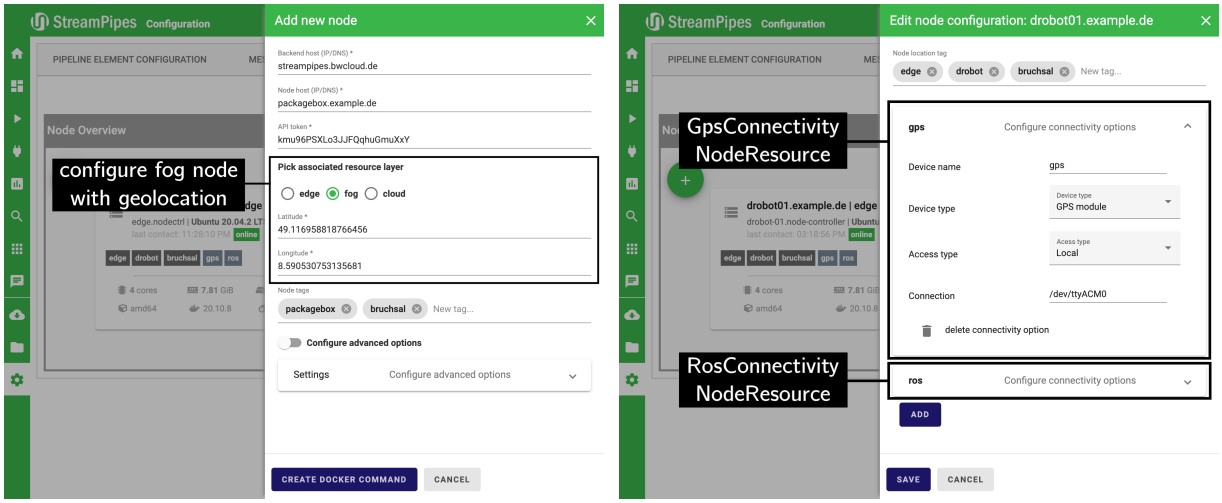


(d) Offload point-in-polygon processor p_2

Figure 9.5 Delivery robot and package box prototype (a), in action during package pickup and delivery scenario (b), test area with geofence reconfiguration (c) and point-in-polygon processor offloading (d). Map data © OpenStreetMap [OpenStreetMap contributors 2021].

Thus, the goals are as follows:

- *Applicational needs*—First, employ location monitoring of the delivery robot based on adaptive geofences and notify fleet operators in the event of any abnormal behavior. Second, temporarily buffer events to succeeding pipeline elements either at the fog or cloud layer during intermittent network outages and synchronize upon reestablishing a stable connection. Third, automatically offload processors to nearby packet boxes (if possible) or to the cloud when overly compute-intensive edge processing puts too much strain on the compute node in the delivery robot.
- *Organizational needs*—Enable citizen technologists such as fleet operators to flexibly integrate new delivery robots (edge nodes) and package boxes (fog nodes) and allow them to manage geo-distributed remote condition and location monitoring pipelines from a central control center.

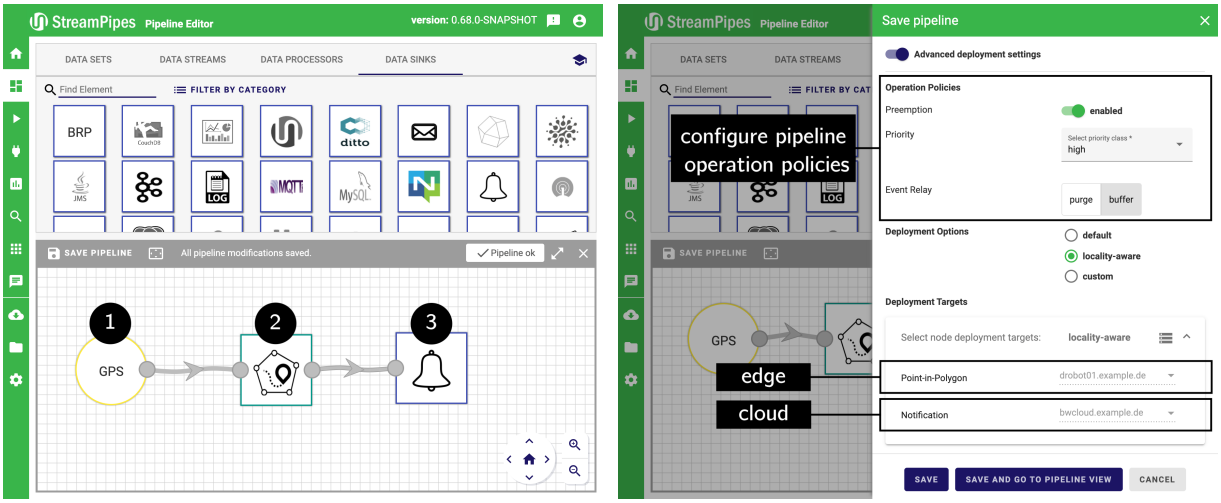


(a) add fog node with geolocation **(b)** GPS and ROS connectivity node resources

Figure 9.6 Setup phase: Adding new node fog node with geolocation information to extend the fog computing infrastructure managed by StreamPipes (a). Edge node configuration for the delivery robot showing connectivity node resources for a local GPS sensor to retrieve live location updates and local ROS interface (b).

Setup Phase. As part of the research project, a real-world fog computing testbed was set up which comprises two x86-based industrial edge PC employed within the delivery robot and the package box prototypes apart from a virtual machine cloud instance hosted in the bwCloud[3] infrastructure. Each node was first provisioned with the developed node controller using the previously described guided provisioning process while providing appropriate node tags. However, unlike the previous case study with stationary edge nodes, the delivery robot is inherently mobile. Therefore, our proposed node model allows to incorporate the concept of GeoLocation denoting the node's physical location in terms of latitude and longitude properties to complement domain-specific node tags. Figure 9.6 shows the configuration dialog when adding a new fog node for the package box, including geolocation properties and excerpts properties of the GpsConnectivityNodeResource and RosConnectivityNodeResource concept for the edge node. Thereby, provided information on the GPS connectivity node resource allowed to automatically retrieve live location updates for the delivery robot during the operation phase in order to continuously update related geolocation properties in its node description. In addition, the delivery robot was equipped with a LTE modem to communicate with the cloud node. Further, this allowed the nearby package box to establish a local wireless connection to directly communicate with the delivery robot. This provided the ability to exploit computational resources in close proximity and in an ad-hoc fashion when needed.

[3]bwCloud is an OpenStack powered flexible Infrastructure-as-a-Service offering optimized for researchers, lecturers and students in Baden-Wurttemberg (https://www.bw-cloud.org/).



(a) pipeline authoring **(b)** preference-based configuration

Figure 9.7 Operation phase: Location monitoring pipeline $\mathcal{P}_{loc}$ during pipeline authoring (a) and when specifying preference-based configurations prior to geo-distributed pipeline deployment (b).

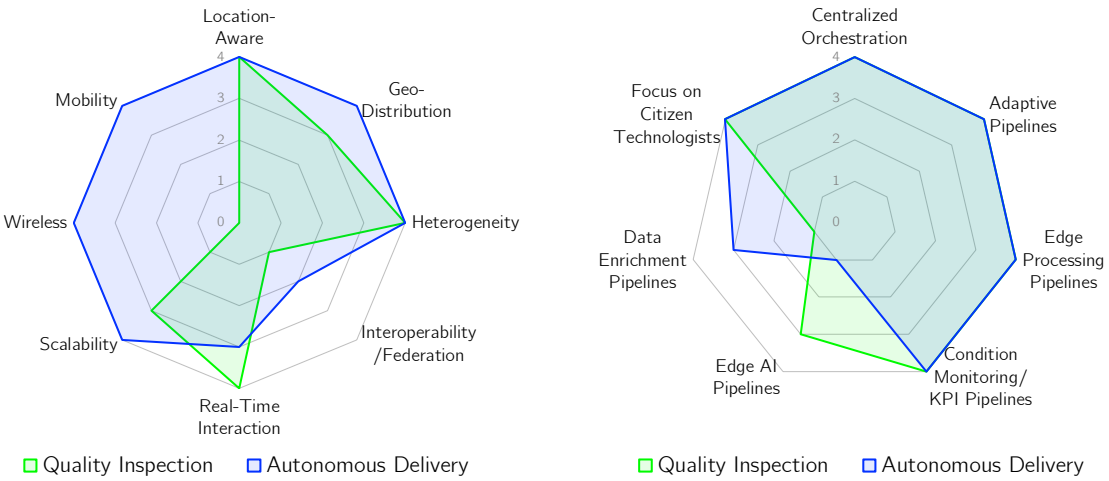
Operation Phase. To evaluate the practical applicability of our approach for both applicational and organizational needs, we leveraged our running example for the location monitoring pipeline $\mathcal{P}_{loc}$ which we introduced in Section 3.1.2 and further used in Chapters 6 to 8 to exemplify our proposed models, concepts and methods. In brief, Figure 9.7a shows the three pipeline elements used by the fleet operator during pipeline authoring: ❶ a *ROS* adapter for the system event stream encompassing information on the current geolocation, ❷ a *point-in-polygon* processor to constantly monitor the current location and compare it to a reconfigurable static geofence property and ❸ a *notification* sink to inform the fleet operator when the delivery robot unintentionally leaves the valid test area defined by the geofence.

After completing the pipeline authoring, we instructed the partner responsible for fleet management tasks to perform the preference-based configuration to prepare the location monitoring pipeline for the geo-distributed deployment. Figure 9.7b depicts the result of the "locality-aware" deployment option which was added to demonstrate the extensibility of the underlying concept. In contrast to the custom deployment option where nodes are assigned in a manual selection process by citizen technologists, the locality-aware deployment option uses a rudimentary system-side node mapping strategy where eligible nodes are automatically assigned. Therefore, starting from the root of the pipeline topology, namely the adapter and its deployment target node typically in close proximity to the event source (here the delivery robot), a basic bin packing algorithm was integrated. This assigns succeeding pipeline elements to the same node as long as stated resource requirements can be served before randomly assigning them to other nodes. In the given case, as the connectivity requirement for the ROS adapter (see Section 6.2) was only fulfilled by the delivery robot, the point-in-polygon processor was also assigned to the

delivery robot. Yet, the notification sink was deliberately mapped to the cloud node as a requirement by the fleet operator. Again, the ROS adapter was already pre-deployed on the delivery robot in a previous step within the connect module of StreamPipes (not shown in the figure). Additionally, global pipeline operation policies were configured by the partner. The operation policy for relays was set to "buffer" in order to configure the event stream relay manager of the node controller on the delivery robot to buffer events when passing through radio holes. Besides, preemption was enabled with a high priority class. After the deployment of the location monitoring pipeline, the delivery robot performed an autonomous package pickup and delivery scenario in the test area. Upon receiving a new order schedule, the delivery robot picked up the package at the satellite hub and autonomously performed the delivery task to the package box to deposit the package. During the delivery, the delivery robot covered a driving distance of 232 meters from satellite hub to package box over typical street terrain and performed the delivery task within roughly 7 minutes including necessary handling steps. The practical applicability of the run-time reconfiguration and offloading was assessed in this real-world setting during this pickup and delivery scenario. For the reconfiguration, the fleet operator was instructed to update the valid test area during the delivery process (Figure 9.5c). Technically, this was possible by the geofence reconfigurable static property within the point-in-polygon processor. For the offloading, we registered an offloading manifest comprising a threshold-based CPU evaluation policy (N_ϑ=2, T_{cpu}=60%) and a priority selection strategy. To provoke violations in a deterministic manner, a CPU load generator was implemented and wrapped inside the point-in-polygon processor which allowed to set the CPU load to a given target value, here 70 %. During another delivery process, multiple individual threshold violations were detected by the node controller on the delivery robot. Upon the third violation, an offloading was triggered for the only actively running processor, namely the point-in-polygon processor (Figure 9.5d). At the time, the geolocation for the package box fell within the WiFi coverage of the delivery robot. Thus, a regional offloading to the nearby fog node in the package box was performed while required event stream relays were updated accordingly.

9.2.3 Discussion

The general goal behind conducting case studies was to obtain real-world insights for (1) assessing the expressivity of our proposed node model and identify potential shortcomings or missing concepts and (2) evaluate the practical applicability of the methods and concepts entailed in the system. Yet, as a prerequisite, we need to clarify to what extent the presented case studies are capable of representing a broad spectrum of potential scenarios in the domain of IIoT and fog computing. This can be validated by assessing the case studies with regard to the fundamental fog computing characteristics from Section 2.3.2 as well as applicational and organizational needs categorized in Section 3.2. The former denotes dimensions which shape the general fog computing landscape, the



(a) case studies → fog computing characteristics **(b)** case studies → applicational/organizational needs

Figure 9.8 Mapping case studies to fog computing characteristics (a) and applicational and organizational needs (b) according to the degree of importance for the specific dimension.

latter describes dimensions of concrete demands for geo-distributed event-driven applications in fog computing entailed in typical directed and sense-process-respond models. Therefore, we evaluate each dimension with respect to the degree of *importance* within the specific case study according to a five point scale: (0) *not important*, (1) *slightly important*, (2) *moderately important*, (3) *very important*, and (4) *extremely important*. While not all dimensions need to be at the highest level to answer the question about the suitability of our case studies, the higher the importance level, the more complexity is generally attained. In turn, this induces more challenges that need to be covered by our models, concepts and methods building the foundations of the holistic application management. Figure 9.8 illustrates the results. As can be seen, both fog computing characteristics as well as applicational and organizational needs are broadly covered by the discussed case studies. Only in terms of interoperability and federation aspects both case studies only indicate a slight importance which is due to the fact that no production-grade fog service providers are present at the time of writing this thesis. Despite the fact that the quality inspection case study follows a rule-based approach for decision-making at the edge, this can be equally seen as a placeholder for more sophisticated machine learning approaches when the complexity level of the domain problem increases. Hence, we argue that these case studies are well-capable of covering the application and requirement space in the given context. This provides an essential indicator to validate whether our holistic management approach for geo-distributed event-driven applications is suitable to further excel the democratization movement and to provide support for citizen technologists.

One key to fostering practical applicability in real-word deployments results from the end-to-end support along the node and the pipeline element life cycle. From the configuration and registration of new nodes and supported pipeline elements during the setup phase,

over the pipeline deployment and operation, to pipeline adaptations in the operation phase. In both case studies, citizen technologists in the role of quality engineers or fleet operators were able to perform related tasks. The node model not only is a decisive factor when it comes to uncover heterogeneity dimensions inherent to fog computing. It also allows to obtain knowledge in a machine-interpretable fashion while extending existing domain vocabularies. At the same time, citizen technologists are abstracted from any modeling efforts by the management middleware regardless of the present infrastructure and node design which lowers technical entry barriers. The model shows its versatility for two prominent application types, namely in a stationary setting for the quality inspection case and in a mobile setting for the delivery robot case. Despite relevant geolocation information in the mobile scenario which are covered by the model, the extensible node tags concept further allows to introduce domain-specific knowledge, e.g., the cobot type or line number. In particular these user-defined domain tags are useful to prepare the deployment. In addition, guidance is provided throughout the setup and operation phase. First, the connection process to IoT devices for specific adapter and sink types is simplified by leveraging connectivity information of individual nodes. Second, the extensible set of preference-based deployment options and operation policies provide freedom of choice to citizen technologists. This gives them the flexibility to fit deployment and operation tasks to a wide range of application and business requirements as shown in the case studies. Third, citizen technologists are enabled to perform pipeline adaptation actions (when required) with system-side support. Moreover, the architectural design and the tool support allow to intuitively add new nodes as demands grow which is of importance for both case studies.

Hence, we are confident that both the expressivity of the node model and the practical applicability of our holistic application management approach is suitable to target the emerging needs in a wide variety of fog-related use cases in domains like the IIoT. Yet, the performed studies may not equally reflect all essential criteria for answering our principal research question and thus require further investigation. Next, we assess to which extent the proposed holistic application management approach meets the derived requirements to further support our first impression.

9.3 Conceptual Investigation

In this section, we evaluate the requirement fulfillment level by means of the conceptual investigation method. Therefore, elicited requirements from Chapter 5 entailing model-specific and architecture-specific as well as system-specific aspects are assessed. Accordingly, we elaborate how these requirements are addressed by our proposed models, concepts and methods towards a holistic application management system for geo-distributed event-driven applications. Figure 9.9 shows our research questions and elicited requirements.

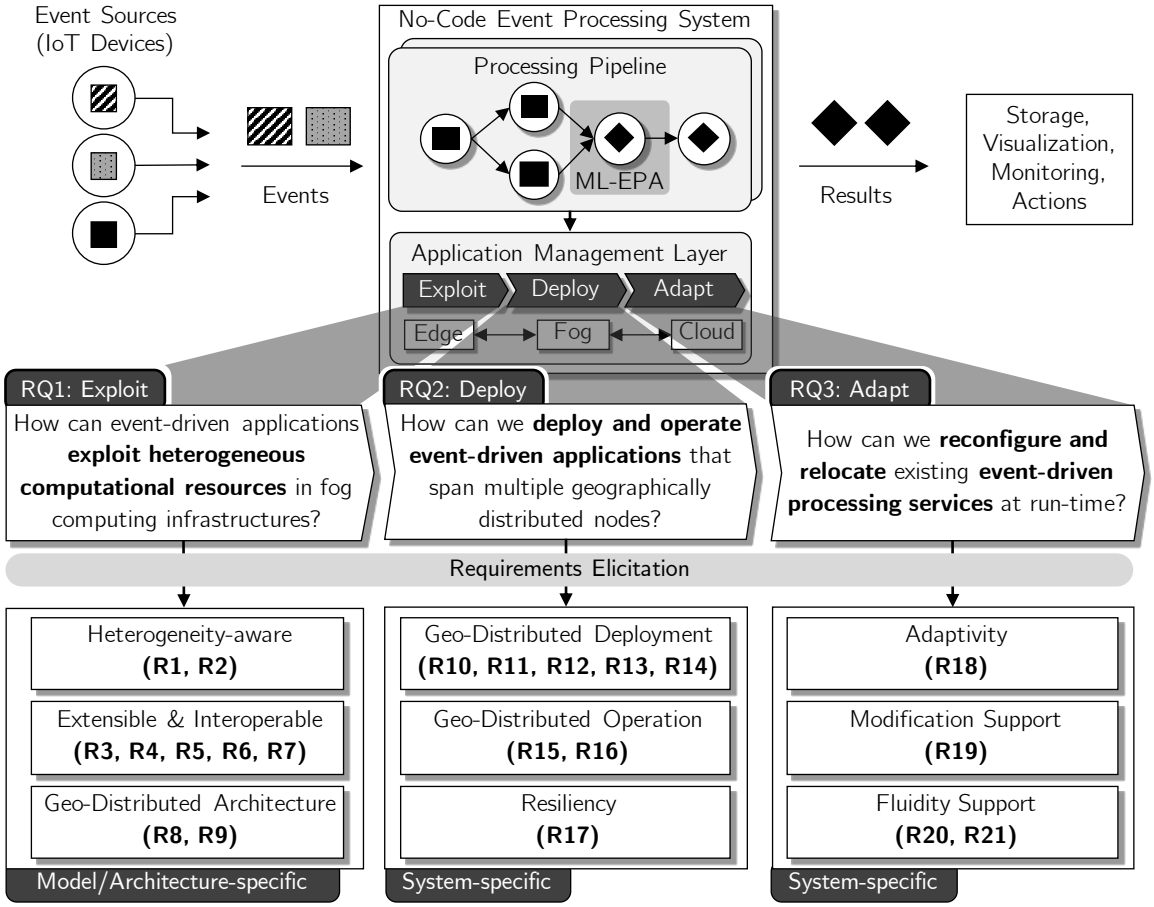


Figure 9.9 Mapping of research questions to requirements

9.3.1 Requirements Fulfillment

Following, we first elaborate on derived requirements (**R1-R9**) which build the foundation and pose decisive architectural design decisions for a holistic event-driven application management in fog computing. Table 9.1 summarizes the model and architecture-specific requirements and their fulfillment degree. After reviewing them, system-specific requirements (**R11-R21**) are investigated and discussed. In particular, the latter set of requirements deal with technical challenges and describe *what* aspects are indispensable for a holistic application management system for geo-distributed event-driven applications while aiming at the question *how* to realize respective functionalities. Table 9.2 summarizes the system-specific requirements and their fulfillment degree.

Model/Architecture-specific Requirements Fulfillment. When it comes to executing geo-distributed event-driven applications in fog infrastructures, a fundamental question that arises is how to tackle the inherent heterogeneity. Consequently, concepts are required which allow to create transparency and resource-awareness of the discussed

heterogeneity dimensions. Therefore, the first two requirements (**R1**, **R2**) are directly fulfilled by related concepts implied in the proposed node model from Section 6.4. Not only do nodes expose their generated node description via web-based standards and register them at the central application management layer to gain knowledge on hardware, software and connectivity resource dimensions in a machine-interpretable manner, the model itself is generic. Hence, it does not rely on specific node types, processor architectures such as x86, ARM32 or ARM64, or concrete implementations. Moreover, the described NodeResource concept is extensible and provides the ability to either add new subclasses for individual resource dimensions (if required) or even add new resource dimensions as a whole (**R3**). Additionally, the generic NodeTag concept fulfills the requirement for domain knowledge (**R4**) by providing the ability to structure and organize nodes by specifying meaningful domain metadata. This further facilitates the node selection and deployment process for citizen technologists as it provides additional knowledge over a potentially large set of geographically dispersed nodes. Besides, both the stationary scenario and the mobile edge scenario benefit from the node tag concept as it either allows to express relative location information in the former case, or to enrich information on the current geolocation in the latter case. The developed node model is built on well-established concepts in web technologies and is represented in RDF which fulfills the requirement for interoperability (**R5**).

The decomposition of complex analytical problems into smaller discrete and reusable computational building blocks is well-suited to meet the induced IIoT-related application demands. Moreover, it alleviates the distribution of individual processing parts, namely pipeline elements, to geo-distributed nodes along the cloud-edge continuum. Therefore, the interoperability of the proposed node model provides the ability to reuse existing vocabularies for describing EPN entities and extend them with essential concepts and properties to enhance the geo-distributed operation. Concretely, extensions to the adapter and EPA vocabulary (see Section 6.4.1) allow pipeline element developer to express necessary node resource requirements for hardware, software and connectivity resources as defined in the formal pipeline application model Section 6.2. Still such modeling efforts can be troublesome and requires advanced knowledge. To this end, foundational concepts are wrapped and integrated as extensions to the SDK as part of the tool support in StreamPipes which fulfill related requirements (**R6**, **R7**). Architecture-related requirements regarding the two-level management approach (**R8**) are fulfilled by the proposed geo-distributed master-worker architectural design introduced in Section 6.5. In addition, the general design is further refined in Sections 7.3 and 8.3 with a central coordinating application management middleware which is complemented by the local node controller. The seamless interaction between both central and local management enables the geo-distributed pipeline deployment, operation and adaptation along the node and pipeline element life cycle. Lastly, resource isolation is ensured by relying on lightweight containers to provision nodes with essential services (**R9**). In this matter, the initial node provisioning in the setup phase is supported by the generic DeploymentContainer concept from Section 6.4.3. This is incorporated in the tool support and allows pipeline

N°	Description	Fulfilled by
R1	Heterogeneity-aware	Vocabulary, NodeDescription (Section 6.4)
R2	Platform-agnostic	Vocabulary, NodeDescription (Section 6.4)
R3	Extensibility	Vocabulary, NodeResource (Section 6.4.1)
R4	Domain Knowledge	Vocabulary, NodeTag (Section 6.4.2), tool support
R5	Interoperability	Vocabulary, RDF data model (Section 6.4)
R6	Dataflow Composition	Vocabulary, pipeline application model (Section 6.2)
R7	Requirement Declaration	EPA/Adapter vocabulary extension (Section 6.4.1), SDK
R8	Two-level Management	Geo-distributed architecture (Section 6.5)
R9	Isolation	Vocabulary, DeploymentContainer (Section 6.4.3)

Table 9.1 Requirements fulfillment: Model and architecture-specific requirements

element developers to declare the respective services in generic, technology-agnostic deployment manifests which are used by the node controller alongside knowledge on the present container runtime to perform the provisioning.

System-specific Requirements Fulfillment. In contrast to alternative approaches delineated in Chapter 4, we aim at combining aspects from both the application-centric and infrastructure-centric world. This is transferred into a holistic application management approach geared towards the demands of citizen technologists. Thereby, the system leverages previously elaborated concepts and the derived geo-distributed architecture as foundational building blocks which shape the subsequent design of related components. In view of this, both the central pipeline management and the node controller component provide end-to-end support for geo-distributing and operating individual pipeline elements (Section 7.3). Thereby, the employed concepts and methods cover the complete deployment process. From the eager pipeline element validation (see Section 7.2), over the coarse to fine-grained node validation (see Section 7.3.2), to event stream relay generation (see Section 7.3.3) and final geo-distribution (see Section 7.3.4). The node controller further complements the end-to-end support in terms of node-level management capabilities (Section 7.3.5). Overall, this fulfills related requirements (**R10-R13**). Moreover, deployment and operation support are fulfilled by providing preference-based configurations in terms of pipeline deployment options and operation policies from Section 7.3.1. These allow citizen technologists to customize deployment and operation aspects, such as selecting deployment target nodes or enabling the preemption mechanism, in a self-service manner (**R14**, **R15**). Further, flexible edge-fog-cloud communication patterns in view of the event routing requirement (**R16**) are fulfilled by several concepts and methods building on top of each other. This includes the foundational EventStreamRelay concept, LAEDS for intra-node and inter-node event dissemination and the generation and execution of event stream relays. Besides, resiliency demands (**R17**) are addressed at two levels to account for the dynamics in fog computing. First, citizen technologists can configure an operation policy for relays stating how the node controller should deal

N°	Description	Fulfilled by
R10	Geo-Distribution	Pipeline/node management, node controller (Section 7.3)
R11	Node Autonomy	Node controller (Section 7.3.5)
R12	Abstraction	Tool support (Section 7.4)
R13	Matching	Validation procedure (Section 7.3.2)
R14	Deployment Support	Deployment options (Section 7.3.1)
R15	Operation Support	Operation policies (Section 7.3.1)
R16	Event Routing	Vocabulary, LAEDS, relay (Section 7.3.3)
R17	Resiliency	Relay (Section 7.3.1), node controller (Section 7.3.5)
R18	Adaptivity	Pipeline adaptation methodology (Section 8.2)
R19	Reconfiguration Support	Vocabulary, adaptation gate/event (Section 8.3.1)
R20	Migration Support	Migration scheme (Section 8.3.2)
R21	Context-aware Offloading	Offloading scheme/manifest (Section 8.3.3)

Table 9.2 Requirements fulfillment: System-specific requirements

with situations of temporary network outages. As such, events may be buffered at the source node controller for a configurable length and are send upon reestablishing the connection to the target as discussed in Section 7.3.5. Second, invoked pipeline elements and event stream relays are persistently stored at node level which provides the ability to issue local re-invocation actions in the case of unintentional node restarts. Next, the proposed pipeline adaptation methodology from Section 8.2 covers both external and internal decision-making origins to account for user-initiated and system-initiated adaptation actions at the same time (**R18**). The methodology facilitates the pipeline evolution over-time which is expressed in terms of typical adaptation types. On the one hand, user-initiated run-time reconfiguration actions (see Section 8.3.1) integrate the ReconfigurableStaticProperty concept alongside two abstractions, namely adaptation gate and adaption event, to alter design-time decisions of static properties. On the other hand, user-initiated run-time migration actions are realized on the basis of a step-wise migration scheme (see Section 8.3.2). Additional system-initiated offloading actions use the idea of offloading manifests and an offloading scheme to relocate pipeline elements without human intervention when observed context changes violate stated criteria (see Section 8.3.3). In summary, this fulfills the related requirements (**R19-R21**).

9.3.2 Discussion

The conceptual investigation shows that all derived requirements are fulfilled by the introduced models, concepts and methods in this work. Yet, it further reveals the inter-correlation among the individual requirements. Thereby, the system leverages concepts and the outlined architectural design from Chapter 6 as foundational building blocks for the subsequent component design. This equally applies for components at the central application management layer and the node level. Moreover, this manifests in employed

methods for deploying, operating and adapting event-driven applications in fog infrastructures as discussed in Chapters 7 and 8. The provided tool support completes the overall picture. Consequently, this leads to the point where managing geo-distributed event-driven applications is accessible to citizen technologists which strengthens the overall degree of fulfillment.

9.4 Performance Tests

In this section, we present results from an extensive set of performance tests regarding *operation* and *adaptation* aspects to evaluate the run-time performance of the developed system. This allows to draw conclusions on potential limitations of our proposed concepts and methods as stated in Section 9.1. To obtain insightful results, we set up a configurable fog computing testbed representing a typical real-world environment. In this regard, we use computational nodes which are commonly found in literature [Bellavista and Zanni 2017; Yigitoglu et al. 2017; Ahmed and Pierre 2018]. These nodes represent different system architectures, operating systems and hardware capabilities and are well-suited for investigating respective fog computing scenarios. Following, we describe the setup of the fog computing testbed prior to presenting individual performance test scenarios and discussing the related results.

9.4.1 Setup

The fog computing testbed comprises four nodes: two ARM32-based Raspberry Pi Model 4 (RPi4 1/2) single-board computer serving as edge nodes, one x86-based Intel NUC Mini PC (NUC) representing a fog node and one x86-based powerful virtual machine (VM) acting as a cloud node. Table 9.3 summarizes essential node specifications:

Id	Type	Model	OS	CPU	Memory	Network
RPi4 1	EN[1]	Raspberry Pi 4	Raspbian 10	4×1.5 GHz	4 GB	1 Gbit
RPi4 2	EN[1]	Raspberry Pi 4	Raspbian 10	4×1.5 GHz	4 GB	1 Gbit
NUC	FN[1]	Intel NUC	Ubuntu 18.04.6	8×2.7 GHz	32 GB	1 Gbit
VM	CN[1]	Virtual Machine	Ubuntu 16.04.7	16×2.1 GHz	236 GB	1 Gbit

[1] Edge node (EN), fog node (FN), cloud node (CN)

Table 9.3 Fog computing testbed: Node overview

Network. In compliance with our formal fog infrastructure model provided in Section 6.3, all participating nodes are capable of communicating with each other over the network. As such, all testbed nodes, except the VM, are connected to a Gigabit (Gbit)

Ethernet network switch with typical low network round-trip latencies for the single-hop distance with an average around $0.2\ ms$ over 100 consecutive ping runs. In addition, the VM resides on a server within the same local area network accessible over a Gbit network connection and an average latency around $0.5\ ms$ over 100 consecutive ping runs. In order to account for this, we create artificial network latencies between every pair of nodes by leveraging the Linux traffic control (`tc`) command to emulate wide area network delays in a configurable fashion. Table 9.4 captures the pairwise network latencies present during all performance tests. This allows to provide realistic latencies among the geo-distributed nodes, in particular to reflect larger network distances to potentially remote cloud nodes residing at dispersed data center locations. Latency information are obtained from the WonderNetwork[4] Global Ping Statistics providing average city to city ping times, ranging from $7\ ms$ for Frankfurt–Munich, over $14\ ms$ for Frankfurt–Amsterdam, to $40\ ms$ for Frankfurt–Vienna. Consequently, while *edge-cloud* and *fog-cloud* latencies are set to $25 \pm 2\ ms$ and $20 \pm 2\ ms$ respectively, *edge-fog* latencies are set to be in the low milliseconds range at $5 \pm 2\ ms$. Moreover, *edge-edge* latencies are kept at the actual measured latency as previously described. This arguably denotes a typical setup for geo-distributed fog computing infrastructures assuming no cross-continental cloud data center connections.

	RPi4 1 (edge node)	**RPi4 2** (edge node)	**NUC** (fog node)	**VM** (cloud node)
RPi4 1	0.03	0.2[1]	5 ± 2	25 ± 2
RPi4 2	0.2[1]	0.03	5 ± 2	25 ± 2
NUC	5 ± 2	5 ± 2	0.03	20 ± 2
VM	25 ± 2	25 ± 2	20 ± 2	0.03

[1] Actual avg. latency measured 100 ping runs (edge nodes only)

Table 9.4 Fog computing testbed: Pairwise network latencies (in ms)

Setting. All nodes are provisioned with Docker $20.10.7$ as the industry's de facto container runtime and its proven feasibility in the context of fog computing to realize similar IoT-related scenarios [Ismail et al. 2015; Pahl and Lee 2015; Alam et al. 2018]. To ensure synchronized clocks within the fog computing testbed for performance measurements, we set up a local time server on the cloud node (VM) using the Network Time Protocol [Mills 1991] and configured all remaining nodes as clients to sync their internal clock with the local time server. This reduces any clock offsets and drifts to a negligible minimum which is a fundamental prerequisite for accurate measurements. We provisioned the fog computing testbed with the developed software artifacts integrated as extensions into the Apache StreamPipes project to employ geo-distributed event-driven application management capabilities.

[4] https://wondernetwork.com/

Table 9.5 provides an overview of the respective containerized components and their assignment to testbed nodes. In brief, relevant containerized components include:

- **performance-test-client**—A service wrapping all performance tests allowing to programmatically interact with the backend to execute individual test runs in a configurable, reproducible and automated manner.
- **streampipes-ui**—Graphical user interface to create pipelines for individual evaluation scenarios and to specify necessary deployment options and operation policies.
- **streampipes-backend**—Core of StreamPipes comprising extensions in terms of central node and pipeline management functionality in addition to an in-memory RDF triplestore realized with Eclipse RDF4J (formerly OpenRDF Sesame) containing the description graphs for pipelines, pipeline elements and nodes.
- **Apache Kafka**—Central message broker for the cloud node realized using Apache Kafka. Although developed for large-scale deployments by partitioning and replicating topics in a distributed system, we use Kafka in a single-node broker setup.
- **Apache Zookeeper**—A distributed coordination service required by Kafka.
- **Consul**—A service-discovery solution and distributed key-value store for pipeline element and backend system configurations.
- **Apache CouchDB**—A NoSQL database to store non-RDF data.
- **Apache ActiveMQ**—A multi-protocol messaging service generally required for live data visualization in the UI, here we use it as a central endpoint to collect distributed performance measurement logs.
- **streampipes-node-controller**—Node controller management service installed on all testbed nodes and responsible for all local node and pipeline element management tasks along the pipeline element life cycle.
- **streampipes-extensions**—Run-time wrapper for Java pipeline elements (adapters, processors, sinks) including custom ones developed for the sake of the evaluation such as a CPU load generator processor to generate a reconfigurable target CPU load and a latency measurement and event logging sink.
- **Eclipse Mosquitto**—Local message broker for all edge and fog nodes realized using Eclipse Mosquitto which implements the lightweight MQTT protocol and used to demonstrate cross-transport protocol capability of event stream relays.

Moreover, we developed and integrated a custom logging module to collect run-time logs of the developed system during the performance tests. Thereby, arbitrary logging events, e.g., current resource usage metrics on container level and scenario-specific point-in-time control logs, are published to ActiveMQ from where they are subscribed, stored and analyzed. Resource metrics on container level provide a good estimate to gain insights about the impact and resource overhead a certain containerized component such as the node controller introduces. Therefore, we implemented a specific container runtime interface for Docker within the container manager component of the node controller (see Section 7.3.5) which allows to periodically collect and publish resource usage metrics for all running container instances on the present node.

N°	Component	Version	RPi4 1 (EN)[1]	RPi4 2 (EN)[1]	NUC (FN)[1]	VM (CN)[1]
1	performance-test-client[2,3]	0.68.0-SNAPSHOT	❍	❍	❍	●
2	streampipes-ui	0.68.0-SNAPSHOT	❍	❍	❍	●
3	streampipes-backend[3]	0.68.0-SNAPSHOT	❍	❍	❍	●
4	Apache Kafka	2.2.0	❍	❍	❍	●
5	Apache Zookeeper	3.4.13	❍	❍	❍	●
6	Consul	1.7.1	❍	❍	❍	●
7	Apache CouchDB	2.3.1	❍	❍	❍	●
8	Apache ActiveMQ	5.15.9	❍	❍	❍	●
9	streampipes-node-controller[3]	0.68.0-SNAPSHOT	●	●	●	●
10	streampipes-extensions[3]	0.68.0-SNAPSHOT	●	●	●	●
11	Eclipse Mosquitto	1.6.12	●	●	●	❍

[1] Edge node (EN), fog node (FN), cloud node (CN)
[2] The performance-test-client container only runs for the duration of the executed tests
[3] Component extended with the logging module
●/❍ Component deployed/not deployed on this node

Table 9.5 Fog computing testbed: Assignment of components to nodes

9.4.2 Evaluations and Results

In the following, we briefly give an overview of the conducted performance tests and state pursued goals to assess the run-time performance of the node controller and our developed system with regard to *operation* and *adaptation* aspects. Afterwards, we provide an in-depth description for each performance test, elaborate scenario-specific evaluation details and discuss results.

First, we evaluate the impact of the node controller and quantify the *resource overhead* due to extended management capabilities in a real-world IIoT setting for various resource consumption tests. Second, we investigate *end-to-end latencies* to examine the performance of our proposed LAEDS approach for intra-node and inter-node communication in geo-distributed fog infrastructures for different deployment topologies and pipeline sizes and compare results against vanilla StreamPipes within defined latency tests. Third, we demonstrate the *feasibility* of the context-aware offloading behavior employed in the node controller for varying offloading manifest configurations in different offloading tests. Hereby, we analyze the *offloading time* to complete an offloading action which marks the time span from the offloading trigger to the final completion of the task. To this extent, we also quantify the fraction of *migration time*, as the offloading problem is translated into a migration problem at the central application management layer. In addition, we determine the order of magnitude of the *downtime* that is the time span when the event stream is temporarily interrupted in the course of the step-wise migration scheme. Table 9.6 summarizes the individual performance tests.

N°	Test	Goal	Metric	OP[1]	AP[1]
1	Resource (**R**)	Node controller overhead	CPU, memory usage	●	❍
2	Latency (**L**)	LAEDS latencies	Latency measurements	●	❍
3	Offloading (**O**)	Feasibility, duration	Offloading measurements	❍	●

[1] Operation performance (OP), adaptation performance (AP)
●/❍ Performance test maps to/does not map to category

Table 9.6 Performance tests: Overview

Node Controller: Resource Consumption and Overhead

In the following, we investigate the overhead of the node controller and its run-time impact for two node types in our fog computing testbed, namely the resource-constrained *edge node* (RPi4 1) and the resource-rich *cloud node* (VM). To this extent, we use the cobot-based product quality inspection and KPI analytics pipeline $\mathcal{P}_{\text{Q,KPI}}$ (see Figure 9.4 in Section 9.2.1) as a representative real-world IIoT-scenario for our investigation. Moreover, we use data from the cobot-based product quality inspection testbed. This allows to create a dataset of reproducible cobot events during the active and inactive test periods which embraces real-world shop floor data and is thus well-suited in fulfilling our requirement.

In brief, this dataset represents recordings of raw cobot events gathered via ROS during the quality inspection conducted parallel to the pick-and-place operation. On the one hand, this includes sensor measurements (*gripper event stream*) from the gripper module, on the other hand, cobot state machine actions (*state event stream*) which are both stored in a so-called *bag*[5]. For the sake of the evaluation, we cut out a slice of roughly $27\ min$ of the original bag which contains 100 performed quality checks including gripper and state events. Table 9.7 gives an overview of the data contained in the ROS bag.

ROS Topic	Event Type	Event Count	Event Rate	Event Size
`/gripper`	gripper event	45.458 events	28 events/s	285 Bytes
`/state`	state event	200 events[1]	every 7s	200 Bytes

[1] Translates to 100 quality checks for start/end signals

Table 9.7 Cobot-based product quality inspection: ROS bag overview

The ROS bag is deployed on a RPi Model 3 (not part of the fog computing testbed) in order to replace the actual cobot during the performance test. We call this the *cobot RPi*. The cobot RPi is connected to the Gbit network switch as parts of the fog computing testbed with no artificial delays employed and solely acts as a data provider within close network proximity to the edge layer.

[5] A *bag* represents an efficient file format for storing ROS message data (see http://wiki.ros.org/Bags).

Scenario Description. To assess the resource consumption of the node controller and evaluate its feasibility in resource-constrained environments, we analyze its CPU and memory usage on the edge node (RPi4 1) and cloud node (VM) over a course of $27\ min$ where cobot data is replayed via the cobot RPi. Therefore, we define three resource consumption scenarios, namely for executing one $\mathcal{P}_{\text{Q,KPI}}$ pipeline instance (**RS1**), for executing two $\mathcal{P}_{\text{Q,KPI}}$ pipeline instances (**RS2**) and for executing three $\mathcal{P}_{\text{Q,KPI}}$ pipeline instances (**RS3**). According to specified deployment target nodes for the pipeline elements in $\mathcal{P}_{\text{Q,KPI}}$ this involves one *edge node* (RPi4 1) and one *cloud node* (VM) requiring the edge node controller to run one, two and three event stream relay instances respectively.

Results. Figure 9.10 shows the results for all three resource consumption scenarios while illustrating both raw CPU and memory usage for the edge and cloud node controller containers. Additionally, we show $Q_1/Q_2/Q_3$ statistics as well as a $10\ s$ rolling mean of the collected resource metrics to smooth out occasional spikes.

For all scenarios (RS1, RS2, RS3), the node controllers on both the edge and the cloud node remain stable over time with regard to CPU and memory usage with no significant variance. Despite some regular spikes in CPU usage on the edge node controller peaking at around 10% due to periodic resource monitoring and health check activities and an increasing number of event stream relays, the overall introduced footprint remains low with $1.4\%/1.4\%/1.3\%$ in Q_2 values for RS1/RS2/RS3. Similar observations in CPU usage can be made for the cloud node controller with periodic spikes at around $2-3\%$ and $0.1\%/0.2\%/0.2\%$ in Q_2 values for RS1/RS2/RS3. However, the cloud node controller is not busy with executing event stream relays resulting in a low additional overhead. This provides a good estimate in quantifying the base load of the node controller in view of the monitored metrics. Regarding the memory usage, differences in management duties between edge and cloud node controller are observable for all scenarios. While the memory usage of the edge node controller slightly increases with $195.5/196.2/197.2$ Mebibyte (MiB) in Q_2 values for RS1/RS2/RS3 due to a growing number of executed event stream relays, the memory usage of cloud node controller remains at a steady level with $182.4/179/179.4$ MiB in Q_2 values. Still, the memory footprint of the node controller is noticeable. A basic explanation for this rather high impact on memory results from the fact that the node controller is implemented in Java and thus requires a Java Virtual Machine (JVM) for execution. At run-time, the memory footprint is primarily induced by the JVM itself. Though configurable, we did not fine-tune the JVM memory parameters and used the default values for the JVM heap size.

We conclude that the overall resource consumption and impact of node controller is almost negligible in terms of CPU usage. Furthermore, the slight growth in memory usage for an increasing number of event stream relays is tolerable in comparison to the absolute memory footprint predominantly resulting from the JVM which requires additional investigation.

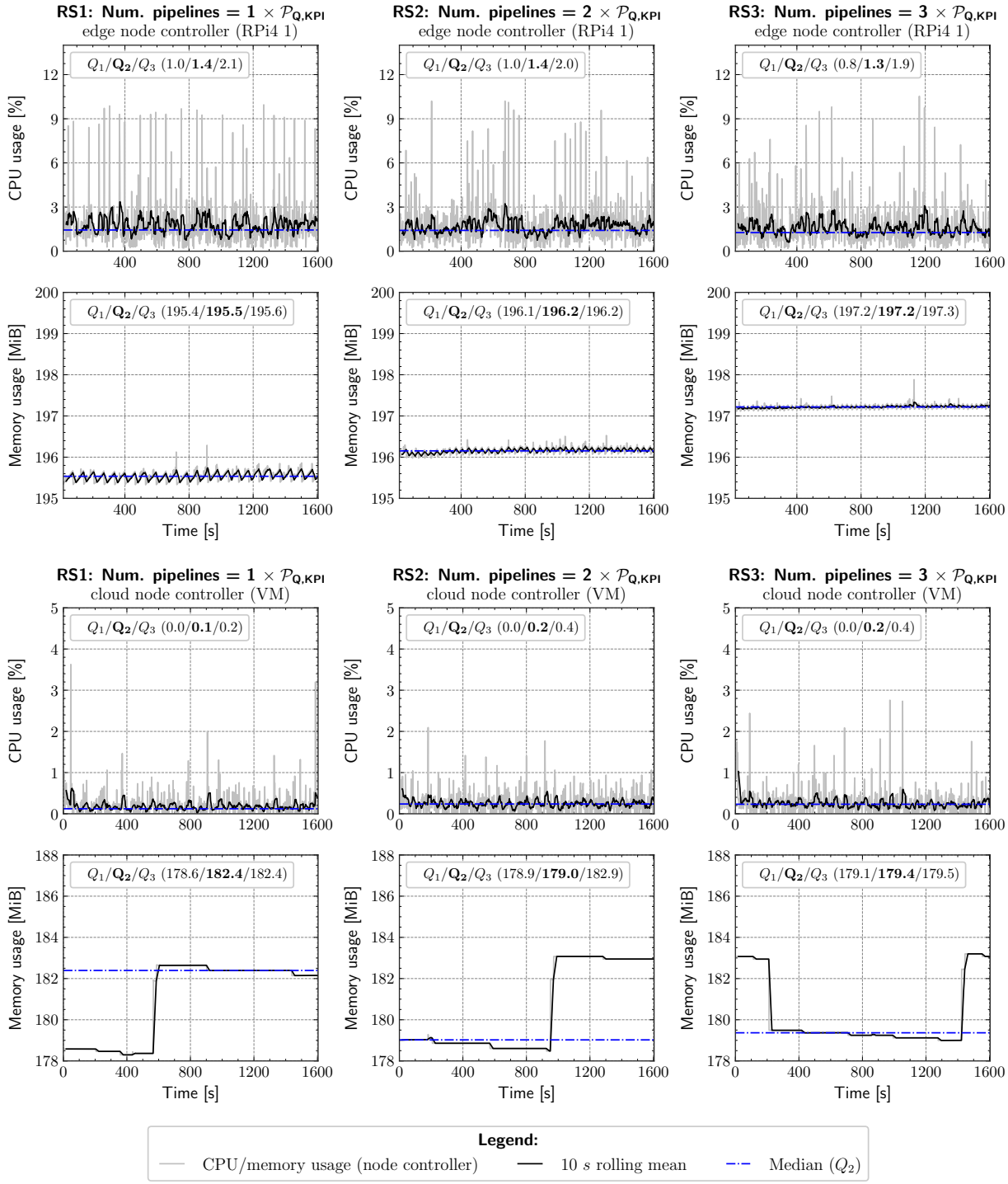


Figure 9.10 Performance evaluation: Resource consumption scenarios. CPU and memory usage by the node controller containers of the edge node (RPi4 1) and the cloud node (VM) at run-time for one $\mathcal{P}_{\text{Q,KPI}}$ pipeline instance (RS1), two $\mathcal{P}_{\text{Q,KPI}}$ pipeline instances (RS2) and three $\mathcal{P}_{\text{Q,KPI}}$ pipeline instances (RS3).

LAEDS: End-to-End Latencies

Following, we evaluate the proposed *locality-aware event dissemination strategy* (LAEDS) for intra-node and inter-node communication between adjacent pipeline elements. The goal is to evaluate end-to-end latencies of LAEDS in latency and reliability-sensitive use cases. This implies both use cases related to the *sense-process-respond* (SPR) model and use cases related to the *directed* model. In this regard, we understand end-to-end latency as the time it takes for an event to travel through the processing pipeline from its source (adapter) over intermittent steps (processor) to its destination (sink). That means, we limit our scope to the system boundary knowingly ignoring additional external I/O connections.

For this performance test, we use a single-connected[6] *latency measurement pipeline* $\mathcal{P}_{\text{lat}}$ which comprises the following pipeline element types:

1. A **random event generator adapter** which produces random events at 100 events per second and 300 Bytes per event entailing a timestamp of the event generation.
2. A **projection processor** which solely forwards received events without applying any additional event processing logic to keep processing latency at a minimum.
3. A **latency logger sink** which first calculates the end-to-end latency for a received event as the time delta between the event generation and the event receiving time and then publishes the result to the logging broker for analyses.

Due to highly synchronized wall clocks in our fog computing testbed, the effect of clock offset, i.e., the time difference between individual nodes, is negligible.

Scenario Description. We define four different latency scenarios evolving around the SPR model (**LS1**, **LS2**) and the directed model (**LS3**, **LS4**) with different processor and sink deployment locations (edge, fog, or cloud node) in addition to varying $\mathcal{P}_{\text{lat}}$ pipeline sizes (3, 4 and 5 pipeline elements). Here, we assume a minimum pipeline size of 3 as there exists exactly one pipeline element per type, i.e., adapter, processor, sink. For the remaining pipeline sizes, we increase the number of projection processors connected in a serial manner—two processors for a pipeline size of 4, three processors for a pipeline sizes 5. Each scenario for each pipeline size is performed over a duration of $10\ min$ which lead to 60.000 generated events. Furthermore, we compare the results of LAEDS against the results of the latest version of vanilla StreamPipes in order to show potential performance gains and assess the limitations of our approach. Yet, it is noteworthy that vanilla StreamPipes relies on a single centralized message broker (Apache Kafka) which obviously limits its applicability in geo-distributed edge and fog scenarios and has to be considered. When discussing the results, we refer to the extended version of StreamPipes with LAEDS as SP^{+}, while using SP^{v} to indicate the latest version of vanilla StreamPipes without LAEDS.

[6] A *single-connected* pipeline is a pipeline with no branches.

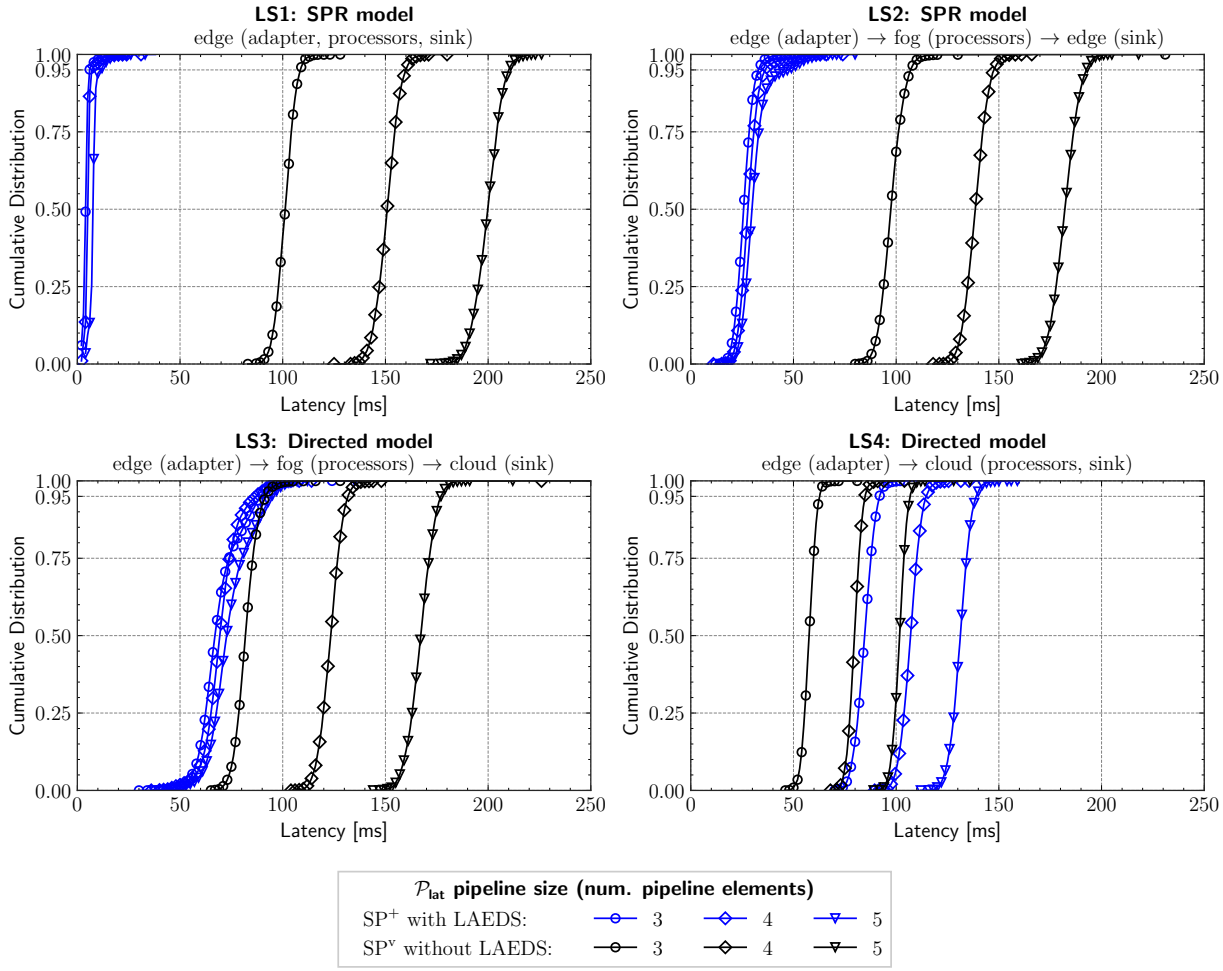


Figure 9.11 Performance evaluation: Latency scenarios. Cumulative distributions of latency measurements for SP$^+$ with LAEDS (—) and SP^v without LAEDS (—) over four latency scenarios reflecting the SPR model (LS1, LS2) as well as the directed model (LS3, LS4) for different processor and sink deployment locations, i.e., edge (RPi4 1), fog (NUC) and cloud (VM), and varying $\mathcal{P}_{\text{lat}}$ pipeline sizes (3, 4 and 5).

Results. Figure 9.11 depicts the cumulative distributions of latency measurements for SP$^+$ with LAEDS and SP^v without LAEDS over the four latency scenarios. Further, Table 9.8 shows $25^{th}/50^{th}/75^{th}/95^{th}$ percentiles in addition to derived performance improvements on median values for SP$^+$ and SP^v.

Unsurprisingly, SP$^+$ with LAEDS clearly outperforms SP^v in three out of four scenarios, namely for LS1, LS2 and LS3. In the case of LS1, SP$^+$ performs best with little overall variability for an increasing pipeline size. This is due to the ability of LAEDS to apply intra-node communication over the local message broker between adjacent and colocated pipeline elements as it is the case in this edge only scenario. When looking at 50^{th} percentile in LS1 (also referred to as median which we use hereafter), this results in latencies of $5/6/8\ ms$ for pipeline sizes of $3/4/5$. Moreover, for the 95^{th} percentile in

N°	PS[1]	**Median** latency ($25^{th}/75^{th}/95^{th}$ percentile) SP^{v} *without* LAEDS		SP^{+} *with* LAEDS		**Median** latency PI[1] SP^{+} over SP^{v}
LS1	3	**102**	(98/105/109)	**5**	(4/5/6)	+95%
	4	**151**	(148/155/160)	**6**	(5/6/7)	+96%
	5	**200**	(196/205/211)	**8**	(7/9/12)	+96%
LS2	3	**98**	(94/102/108)	**26**	(24/29/33)	+73%
	4	**139**	(135/143/148)	**28**	(26/31/37)	+80%
	5	**183**	(178/187/193)	**30**	(27/34/48)	+84%
LS3	3	**82**	(79/86/92)	**67**	(63/74/90)	+18%
	4	**124**	(120/127/132)	**70**	(66/75/88)	+43%
	5	**168**	(163/172/177)	**73**	(68/81/95)	+56%
LS4	3	**58**	(56/60/63)	**85**	(82/88/92)	−47%
	4	**80**	(78/82/85)	**108**	(105/111/116)	−34%
	5	**102**	(100/104/107)	**132**	(129/135/139)	−29%

[1] Pipeline size (PS), performance improvement (PI)

Table 9.8 Performance tests: Latency statistics (in ms) and performance improvements

LS1, we still achieve latencies in low milliseconds range between $6/7/12\ ms$ for pipeline sizes of $3/4/5$, i.e., 95% of all measured end-to-end event latencies are equal or below the stated values. In contrast, SP^{v} shows significant growth of median latencies in LS1, with values of $102/151/200\ ms$ in addition to $109/160/211\ ms$ for the 95^{th} percentile, for the pipeline sizes $3/4/5$. This behavior is systematic and also visible in LS2 and LS3 and is explained by the excessive network round trips to exchange events between adjacent pipeline elements to and from the central message broker deployed on the cloud node. Consequently, the specified pairwise network latencies of the underlying fog computing testbed presented in Table 9.4 provide a rough estimate of the minimum achievable latency in a given configuration. Network latencies are further supplemented by some minor processing latency overhead. For instance in LS1 at pipeline size 3, with all pipeline elements of $\mathcal{P}_{\text{lat}}$ being deployed on the edge node, the median of $102\ ms$ latency in SP^{v} is the result of four individual network trips from the edge node (RPi4 1) to the cloud node (VM) and back with a one-way latency of $25 \pm 2\ ms$ between these nodes. This is due to the fact that a single cloud transport protocol is used among all pipeline elements. Noticeably, LS2 and LS3 with processors outsourced to the fog node show an increase in overall end-to-end latencies for SP^{+} in contrast to the local communication in LS1. Yet, this is still not overly impacted by the additional network communication and the increasing pipeline size. Similar to the edge node, LAEDS also implies local event exchange for adjacent and colocated pipeline elements on fog nodes to achieve better performance leading to median latencies of $26/28/30\ ms$ and $33/37/48\ ms$ for the 95^{th} percentile. This also includes two additional event stream relays, namely an *edge-fog* and a *fog-edge* relay for inter-node communication. In the case of LS3, despite still outperforming SP^{v} in all cases, SP^{+} with LAEDS is affected by the additional network latency from the fog node to the cloud resulting in median latencies of $67/70/73\ ms$ and

$90/88/95\ ms$ for the 95^{th} percentile. Only in LS4, we observe SP^{v} outperforming SP^{+} in terms of end-to-end latencies. When looking at the involved deployment locations for executing individual pipeline elements employed in $\mathcal{P}_{\text{lat}}$, namely the edge node for running the adapter and the cloud node for remaining processors and sinks, we see that LS4 resembles the traditional cloud computing approach. Consequently, events from remote locations (here, the edge node) are sent unchanged to the central cloud layer where the typical processing takes place. Hence, SP^{v} performs better as the effect of excessive network round trips to and from the central message broker between geo-distributed pipeline elements is reduced to a single transfer. More interestingly, two key observations can be made in LS4: (1) differences in cumulative latency distributions between SP^{+} and SP^{v} are roughly off by one pipeline element and (2) the offset remains in the same order of magnitude. For a pipeline size of 4, SP^{v} achieves a median latency of $80\ ms$ which is in the same order of magnitude accomplished by SP^{+} for a pipeline size of 3, namely $85\ ms$. This demonstrates the impact of LAEDS which requires to instantiate an event stream relay for inter-node communication. Hence, a relay can be understood as an additional, system-internal routing pipeline element which adds auxiliary processing latency.

Table 9.8 reveals that SP^{+} with LAEDS clearly outperforms the centralized message broker approach of SP^{v} with maximum achievable median performance improvements of $+96\%/+84\%/+56\%$ for LS1, LS2 and LS3 for a pipeline size of 5. Yet in the case of LS4, we also see that LAEDS performs worse, at most -47%, due to the added overhead of the event stream relay. However, this is put into perspective as LS4 represents the traditional centralized cloud computing model with its known limitations in the context of IIoT-related scenarios. Still, LAEDS demonstrates its overall feasibility and provide low latency results in view of typical SPR and directed models for geo-distributed event driven applications.

Context-aware Offloading: Offloading Time, Migration Time, Downtime

The following performance test allows to evaluate multiple adaptation aspects at once. Thereby, the proposed offloading steps not only describe the ability of a node controller to perform offloading decisions in a reactive fashion, but also involve centrally coordinated migration activities to dynamically readjust previous pipeline deployment topologies. Hence, we begin the tests by investigating the node controller's ability to react to context changes according to registered offloading manifests. After evaluating the general feasibility of this policy-driven approach, we analyze the total incurred *offloading time*. Here, the offloading time is the time it takes for the offloading manager to issue an offloading action encompassing an offloading candidate until the centrally coordinated migration action is completed. Consequently, another part of this assessment is to analyze the fraction of the offloading time spend on the migration from one node to another, called *migration time*. As the migration implies temporal interrupts, it is important to determine the cost of a migration in terms of the *downtime* and quantify its order of magnitude.

For the sake of this performance test, we use the following offloading manifest configurations which are implemented and integrated in the offloading manager of the node controller:

- *Resource property*—CPU usage in percent as an indicator for the degree of utilization
- *Evaluation policy*—Threshold-based CPU evaluation policy
- *Selection strategy*—Random selection strategy, priority selection strategy

We use a single-connected *offloading pipeline* $\mathcal{P}_{\text{off}}$ comprising three pipeline elements:

1. A **random event generator adapter** which produces random events at 10 events per second and 300 Bytes per event.
2. A **CPU load generator processor** which allows to generate an arbitrary and reconfigurable target CPU load on the underlying node.
3. A **logger sink** which publishes the received events to the logging broker.

As such, the most crucial pipeline element within $\mathcal{P}_{\text{off}}$ is the CPU load generator processor which allows to configure the following three parameters: (1) a ramp up duration, (2) a ramp up delay and (3) a target CPU load. The former two parameters are statically provided during the initial pipeline element configuration. As such, ramp up duration refers to the time it takes to reach the specified target load while the ramp up delay refers to an initial temporal lag until the load generation begins. We keep these two parameters constant throughout the offloading scenarios with a ramp up duration of $30\ s$ and no ramp up delay. Lastly, the target CPU load configuration is implemented using the proposed ReconfigurableStaticProperty concept and extensions provided for the run-time wrappers in StreamPipes to facilitate run-time reconfigurations. This allows to dynamically set new target load values in order to generate defined load profiles. In addition, it guarantees a predictable and reproducible behavior in the course of our evaluations.

Scenario Description. We define four offloading scenarios that broadly differentiate in horizontal offloading with multiple, consecutively performed offloading actions (**OS1**, **OS2**), i.e., *edge-edge* offloading and re-offloading (onloading) between two edge nodes (RPi4 1, RPi4 2), and vertical offloading with a single offloading action (**OS3**, **OS4**), i.e., *edge-fog* offloading between an edge node (RPi4 1) and a fog node (NUC). We investigate different combinations in CPU threshold values, maximum number of permissible violations and varying selection strategies. For the random selection strategy, we configure one instance of $\mathcal{P}_{\text{off}}$ with preemption enabled. For the priority selection strategy, we duplicate $\mathcal{P}_{\text{off}}$ with preemption enabled for both instances. Additionally, the priority class for one instance is set to "high" while the priority class for the other one is set to "low". Moreover, we vary the evaluation interval τ_{eval} which specifies the time period between two evaluation policy checks. Each scenario is performed over a duration of $1800\ s$ (or $30\ min$) while externally providing target CPU load values through our performance test client to achieve a characteristic load profile.

Moreover, we divide each offloading scenario into three time intervals:

1. The **ramp up interval** ($0-600\ s$). Here, we provoke a step-wise increase in CPU load from an initial base load to a subsequent moderate load by issuing a reconfiguration at $300\ s$. This interval ends with another reconfiguration to further raise the load to a defined maximum knowingly above the evaluation threshold.
2. The **offloading interval** ($600 - 1500\ s$). In this interval, the processor offloading actions are performed according to the scenario-specific configurations. The interval ends with a final reconfiguration to decrease the CPU load.
3. The **fade out interval** ($1500 - 1800\ s$). Lastly, the target CPU load remains steady at the level of the initial base load for the remainder of the time.

For each scenario, all $\mathcal{P}_{\text{off}}$ pipeline elements are initially deployed on the *edge node* (RPi4 1) while the corresponding partner nodes, either the *edge node* (RPi4 2) or the *fog node* (NUC), do not execute any pipeline elements in the beginning of the ramp up interval.

Results. Figure 9.12 illustrates the results for all four offloading scenarios including the scenario-specific configurations. In the following, we elaborate individual offloading characteristics along the individual time intervals in chronological order.

During **ramp up interval**, we clearly see the characteristic step-wise increase in CPU usage on the origin edge node (RPi4 1). This is provoked by the CPU load generator processor of $\mathcal{P}_{\text{off}}$ whereby the ramp up delay of $30\ s$ constitutes to the steepness of the CPU usage slope in all four scenarios. In addition, the respective offloading partner node, namely edge node (RPi4 2) and fog node (NUC), remains at a low and stable base load. As previously shown, especially the run-time base load overhead of the node controller in terms of CPU usage is negligible (see Figure 9.10). In the case of the vertical offloading scenarios (OS3, OS4), we further segment the artificial load profile in the fraction induced by the CPU load generator processor of the "high" priority $\mathcal{P}_{\text{off}}$ and the fraction caused by the "low" priority $\mathcal{P}_{\text{off}}$. With the start of the **offloading interval**, there is a noticeable increase in CPU load provoked by the executed reconfiguration action on the CPU load generator processor which further occupies resources and leads to a CPU usage well above 80% (OS1, OS2, OS3, OS4). This allows to investigate subsequenet offloading actions in a predictable manner. Overall, the anticipated run-time behavior of offloading actions performed by the node controller is confirmed. Within the *horizontal offloading* scenarios, multiple consecutive offloading and re-offloading actions are performed: a total of 7 for OS1, and 3 for OS2. The results show the typical alternating no-load, high-load pattern in the load profile. Thereby, the values range from a low base load to well above 80% on both the origin edge node (RPi4 1) and the partner edge node (RPi4 2) after the CPU load generator processor is selected as the offloading candidate by the random selection strategy and subsequently migrated. The number of offloading actions vary due to scenario-specific configurations. Concretely, decisive factors are the maximum number of permissible CPU threshold violations N_{ϑ} in addition to the evaluation interval τ_{eval}

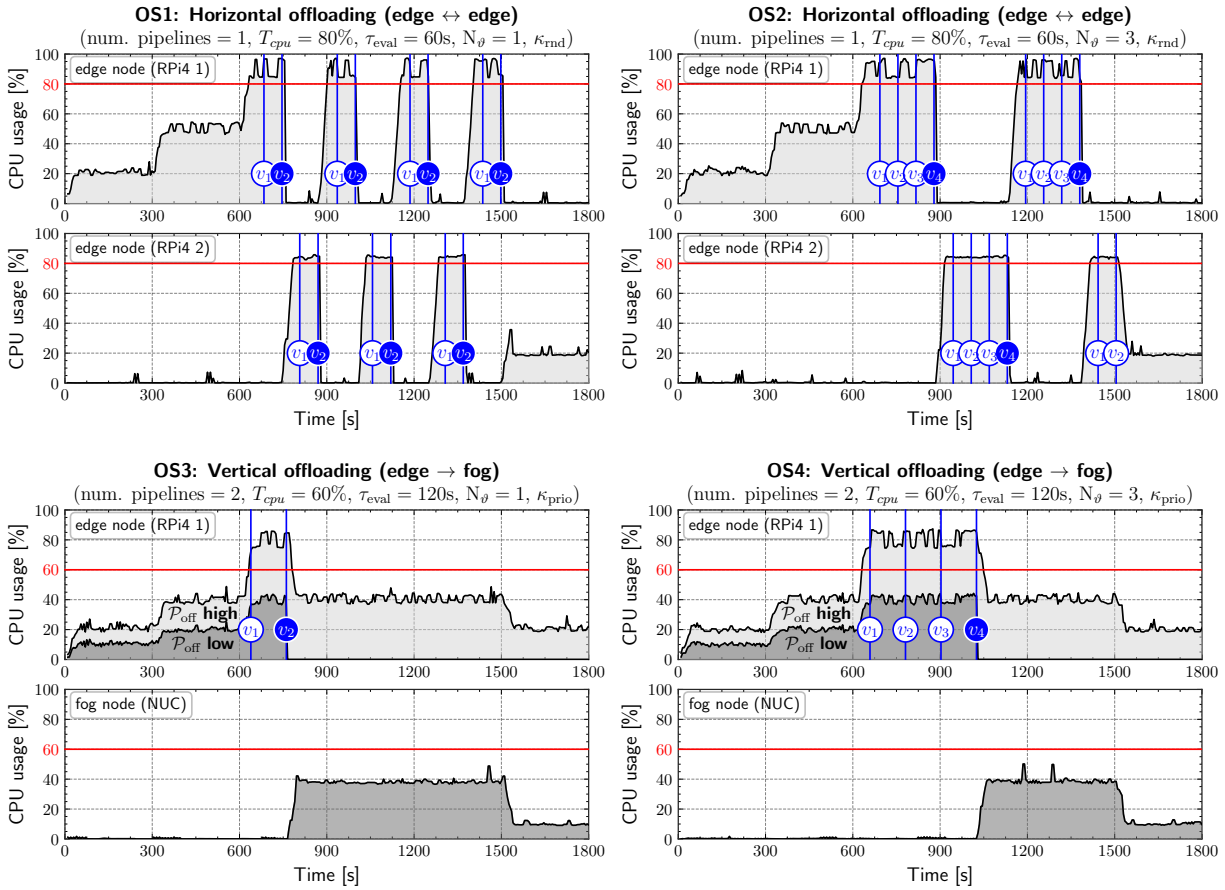


Figure 9.12 Performance evaluation: Offloading scenarios. Horizontal offloading scenarios (OS1, OS2) and vertical offloading scenarios (OS3, OS4) using 1 or 2 deployed $\mathcal{P}_{\text{off}}$ pipeline instances with different offloading manifest configurations: CPU threshold value T_{cpu} (60%, 80%), maximum number of permissible violations N_ϑ (1, 3), random and priority selection strategies (κ_{rnd}, κ_{prio}), evaluation interval τ_{eval} (60 s, 120 s). Horizontal line depicts CPU threshold (—) while vertical lines indicate violation counts (—) due to active evaluation policy with subsequent processor offloading for values exceeding N_ϑ.

which allow to control the sensitivity of the offloading approach. In view of OS1 and OS2 the evaluation interval was configured with 60 s. Within the *vertical offloading* scenarios, single offloading actions are performed at run-time for a modified set of configurations in the offloading manifest, including a lowered threshold for CPU violations ($T_{cpu} = 60\%$) and the priority selection strategy for candidate selection. Apart from that, the offloading manager is configured with a larger evaluation interval ($\tau_{\text{eval}} = 120$ s). For both cases OS3 and OS4, the foreseen offloading behavior is also confirmed. With two differently prioritized $\mathcal{P}_{\text{off}}$ pipeline instances deployed on the edge node (RPi4 1), upon exceeding the maximum number of permissible violations, the low-prioritized CPU load generator processor is selected as an offloading candidate and subsequently relocated to the fog node (NUC). Thereby, the occupied CPU resources on the edge node are instantaneously released when the processor is stopped as a result of the step-wise migration scheme.

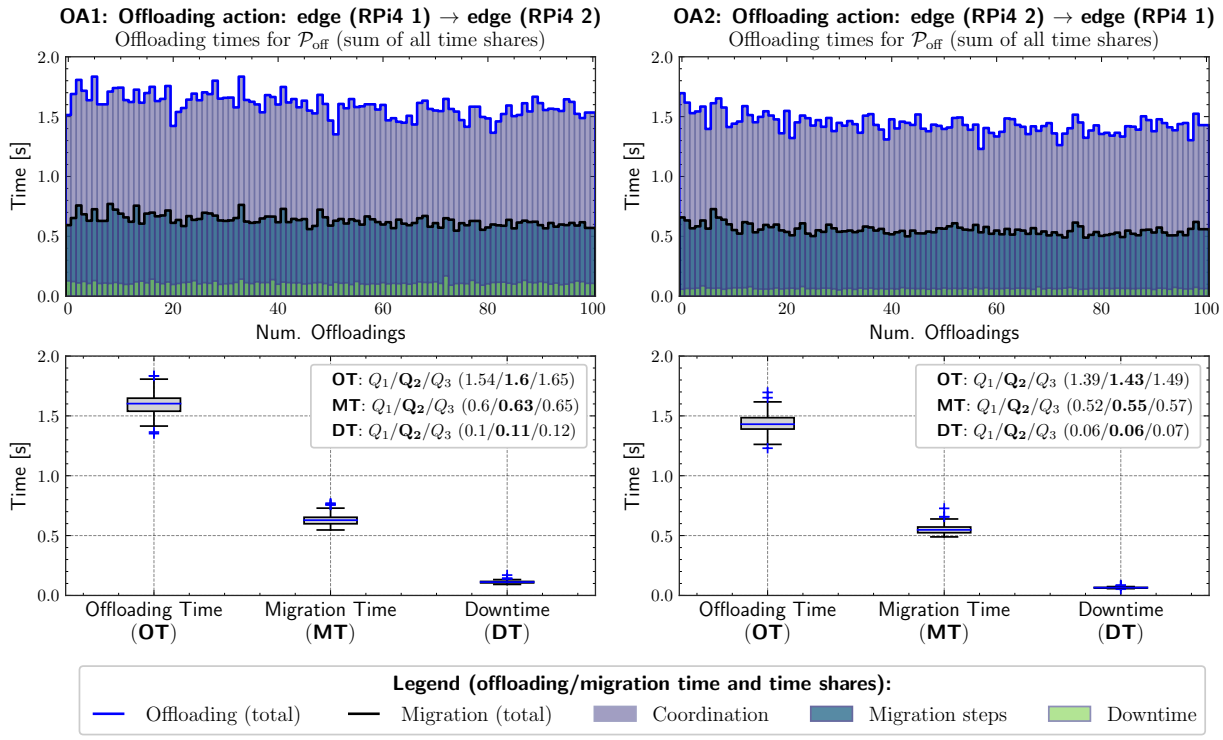


Figure 9.13 Performance evaluation: Offloading time, migration time, downtime. Extended horizontal offloading actions (OA1, OA1) on the basis of the offloading scenario (OS1) show offloading time (—), migration time (—) entailed downtime time share (■) of 100 consecutive offloading and re-offloading actions per direction, either from edge node (RPi4 1) to edge node (RPi4 2) or vice versa.

This leads to a relaxation of the induced load to around 40% constituted by the CPU load generator processor in the high-prioritized pipeline until the end of the offloading interval. On the partner fog node (NUC), the CPU load generator processor of the low-prioritized $\mathcal{P}_{\text{off}}$ pipeline instance starts to occupy resources, yet not violating the threshold. In the **fade out interval**, the artificial load is reduced by reconfiguring the target CPU load of the designated CPU load generator processor. Therefore, a reconfiguration action is sent to the specific node controller instance where the processor is executed at that given point in time and resets potential violation counters as is the case in OS2. Analogously in case of OS3 and OS4, both node controllers are informed to reconfigure the CPU load processors to their original base load.

Next, we discuss results with regard to the *offloading time*, *migration time* and the incurred temporary *downtime* as depicted in Figure 9.13. Therefore, we apply the parameters of OS1 and perform 100 consecutive horizontal offloading actions per direction, i.e., from edge node (RPi4 1) to the edge node (RPi4 2) and vice versa to obtain representative results (**OA1**, **OA2**). In general, we broadly categorize respective time shares constituting to the overall offloading duration into: (1) *coordination* and (2) *migration*. The former describes all relevant preparation and coordination tasks from initiating the offloading

actions over all required preparation steps prior to the migration to receiving the final offloading confirmation. The latter contains all operations as required by the step-wise migration scheme (see Section 8.3.2). This also implies temporary downtime which we measure and plot separately.

Over the course of the executed offloading actions, no significant and noticeable changes for all individual time shares are observed in both cases (OA1, OA2). Hereby, the coordination activities constitute the most to the overall offloading time with an average of roughly 60% in both cases (OA1, OA2), such that migration related tasks are accountable for the remaining 40%. The coordination cannot only be attributed to network communication from geo-distributed edge nodes to the central application management layer, but also contains the sum of various operations (see Algorithm 10 in Section 8.3.3). This includes finding and retrieving the current pipeline description associated with the offloading candidate, validating and selecting a new eligible deployment target node and generating a newly configured pipeline with updated deployment targets. Given the setup of the fog computing testbed, the order of magnitude for total offloading durations is in the low seconds range with median values at $1.6\ s$ (OA1) and $1.43\ s$ (OA2) and also remains stable over time resulting in low variability in the measurements. Similar observations can be made regarding the migration time with overall median values at roughly half a second, $0.63\ s$ in OA1 and $0.55\ s$ in OA2. Looking at the step-wise migration procedures (see Section 8.3.2), the analyses reveals the order of magnitude for the downtime is on average around $0.09\ s$. Here, it highly depends on the actual applicational use case and business requirements whether or not such a downtime is tolerable. In addition, small but noticeable differences exist in absolute downtime values between the two investigated offloading directions, analogously for the migration and offloading time as a result. This can be explained by the fact that all pipeline elements of $\mathcal{P}_{\text{off}}$ are deployed on the edge node (RPi4 1) with only local communication given the initial deployment setting in OA1. When relocating the CPU load generator processor from this initial deployment state to a new destination, here from edge node (RPi4 1) to edge node (RPi4 2), new event stream relays need to be instantiated according to the step-wise migration scheme. Now, when restoring the initial deployment situation in the course of OA2, these relays are obsolete as the communication model falls back to the intra-node one according to LAEDS. Hereby, removing relays are less expensive than their instantiation.

In summary, offloading involves coordination, but the two-staged coordination approach with local and global decision-making ability does not affect the running application until migrations are triggered. In contrast to the overall offloading time, the migration portion plays a more critical role. Yet, the order of magnitude for the induced downtime is tolerable for most real-world IoT applications where approximate instead of exact results are sufficient [Wen et al. 2018]. To this extent, the processor migration time and thus the downtime depends on the logical pipeline model in combination with its deployment topology with respect to the execution location of adjacent pipeline elements in the moment of migration.

9.4.3 Discussion

The conducted performance tests show that the operational overhead and impact of the node controller is minimal making it suitable to also run on resource-constrained edge including single-board computers such as Raspberry Pi even when managing multiple pipeline instances. The predominant factor in question is the overall memory footprint which is the result of the chosen implementation in Java and the default JVM settings. Yet, as we expect to see more powerful small-scale industrial edge PCs in the future due to the ongoing hardware specialization mainly driven by emerging applicational requirements in the area of AI and IoT, the overhead is acceptable. In addition, the performance of LAEDS demonstrates its feasibility for a manifold of application models and deployment situations for geo-distributed event-driven applications with latencies in the low milliseconds. Still, in the typical case of geo-distributed pipeline elements over multiple nodes, using event stream relays for inter-node communication resemble intermediary virtual routing pipeline elements which impact the achievable end-to-end latencies as they introduce additional processing latency. However, the overall impact is negligible for the most common SPR and directed application models apart from the traditional cloud processing one. Moreover, this design decision presents a well-balanced trade-off between performance and flexibility with respect to mediating event streams between potential heterogeneous broker technologies. Further, it is shown that the node controller is capable of detecting contextual changes in its operational context in a reactive manner. Thereby, the configurable detection interval characterizes how sensitive the node controller is with respect to potential contextual changes leading to violations of stated criteria. This detection interval is determined by the evaluation interval and the maximum number of permissible violations. The shorter the detection interval is, the faster the node controller reacts to changes. Yet, this comes at the cost of additional load on the underlying system due to excessive metrics collection and policy evaluation operations. Apart from that, this also gives room for potential false-positive offloading actions in the case of rapidly changing and fluctuating metrics which presents clear limitations of the exemplified threshold-based approach. Yet, all components in the offloading manifest are generic and can be enhanced depending on the actual requirements evolving around the operational goals of the respective use case. Lastly, the demonstrated offloading time and its comprised migration related aspects show an overall small, but measurable, impact on the running pipeline in particular in the matter of the incurred downtime which has to be considered. In practice, losing a small fraction of potentially high-frequent IoT event streams has limited effect on the conveyed information and can also be compensated by message broker technologies, where events are not transient but can be temporarily buffered. Though out of scope of this thesis, one limitation of the current migration approach stems from the fact that only stateless pipeline element migrations are considered. State management concepts [To et al. 2018] can be added to extend the step-wise migration scheme which we leave for future work.

10 Conclusion

In the context of this thesis, models, concepts and methods were introduced which form the basis for the realization of a holistic application management for geo-distributed event-driven applications in heterogeneous fog infrastructures, in which application-centric and infrastructure-centric realms converge. A reference implementation of the application management approach has been developed as an extension for the Apache StreamPipes project to demonstrate its practical applicability and evaluate its performance. This thesis is concluded with a summary of the main contributions alongside the stated research questions in Section 10.1. Afterwards, the significance of our results are discussed in view of newly emerging geo-distributed application scenarios in Section 10.2. Finally, Section 10.3 elaborates on future work to further improve the support of citizen technologists in managing geo-distributed event-driven applications.

10.1 Summary

With the proliferation of the IoT and digitalization initiatives evolving around the Industry 4.0 and the IIoT in industrial settings, a massive amount of real-time data is generated at unprecedented pace. The promise to realize data-driven decision-making by harvesting this data and turning it into useful insights has led to an ever increasing willingness to invest across many industries. Thereby, fog computing acts as an enabler to extend the capabilities of the traditional cloud model and to facilitate reliable and low latency event processing at the the edge of the network. The rapid changes in digital transformation are accommodated by the convergence of previously separate business and IT worlds as a new class of individuals rise in today's enterprises. These citizen technologists play a critical role in acquiring a data-driven culture, yet with the need to be provided with essential tool support which allows them to create and manage geo-distributed event-driven applications.

Therefore, this thesis investigated the following principle research question:

How can citizen technologists be enabled to manage event-driven applications in geographically distributed fog computing infrastructures?

Derived from the principal research question, the following three research questions were identified which we briefly recapitulate while summarizing the main contributions together with the results to answer them.

Research Question 1 (Exploit)**.** *How can event-driven applications exploit heterogeneous computational resources in fog computing infrastructures?*

The first research question concerned with resource management matters of event-driven applications in heterogeneous fog infrastructures is answered in Chapter 6. Building the fundament of this thesis, a primary step is related to the creation of understanding of the heterogeneity in fog infrastructures. The first contribution is a generic and extensible node model on the basis of identified heterogeneity dimensions. On the one hand, this allows to semantically describe node characteristics in terms of node resources besides node-specific metadata including domain-relevant information. On the other hand, it covers concepts for describing event-driven applications and their management by re-using and extending existing state-of-the-art vocabularies in view of platform and technology-agnostic deployment, operation and adaptation. The main advantage results from the symbiosis of infrastructure-centric and application-centric realms. The second contribution is a geo-distributed architecture manifested in a two-level management approach with a central node management and a local node controller. This facilitates the deployment of event-driven applications over heterogeneous computational resources. A reference implementation of the stated concepts is integrated into the Apache StreamPipes project in order to provide necessary tool support for citizen technologists. The evaluation of our node model is based on two case studies resulting from real-world deployments to demonstrate the practical applicability and expressivity. In addition, derived model and architecture-specific requirements are assessed as part of the conceptual investigation.

Research Question 2 (Deploy)**.** *How can we deploy and operate event-driven applications that span multiple geographically distributed nodes?*

The second research question covers aspects related to geo-distributed deployment and operation of event-driven applications and is answered in Chapter 7. As a first contribution, concepts for geo-distributed pipeline management are introduced and added to the geo-distributed system architecture with the goal to provide assistance to citizen technologists when preparing pipelines for their deployment in the beginning of the operation phase. On the basis of publish/subscribe, a second contribution is related to a generic approach for geo-distributed event stream management in order to account for arbitrary pipeline deployment topologies. The suggested locality-aware event dissemination strategy uses the notion of location to decide the respective event dissemination channel between logically adjacent pipeline elements regardless of their physical execution location. On the technical side, event stream relays are presented as a solution for a flexible inter-node communication mechanism providing publish/subscribe-based event dissemination in a technology-agnostic manner. Lastly, the third contribution is a node local

management entity referred to as node controller. This is a system-inherent management service deployed on each node along the cloud-edge continuum which ensures to execute and reliably manage pipeline elements and event stream relays along the application life cycle. The introduced concepts for geo-distributed pipeline deployment and operation including the locality-aware event dissemination strategy are integrated into Apache StreamPipes enhancing the pipeline management capabilities at a central level which are complemented by a newly developed standalone node controller service at the node level. In an extension to the web interface, citizen technologists prepare modeled pipelines for geo-distributed deployment and operation while the system provides necessary support. The approach is evaluated by conceptually investigating related system-specific requirements. Moreover, operation aspects, specifically the node controller overhead and the end-to-end pipeline latencies, are evaluated on the basis of performance tests.

Research Question 3 (Adapt)**.** *How can we reconfigure and relocate existing event-driven processing services at run-time?*

The third research question focuses on aspects of run-time evolution in terms of reconfiguration and relocation of running event-driven processing services which is answered in Chapter 8. The main contribution is an adaptation methodology for event-driven applications which attaches to the sense-process-respond principles of event-driven architectures. The methodology centers around the idea of continuously refining processing results by performing specific adaptations. On the one hand, this implies adaptations triggered by citizen technologists on the basis of their domain expert knowledge, and, on the other hand, adaptations triggered by the system using context knowledge. In this matter, two fundamental abstractions are suggested, namely adaptation gates and adaptation events. Moreover, three adaptation types for event-driven applications are presented and discussed. User-initiated reconfiguration actions allow citizen technologists to modify pipeline element configurations based on the concept of reconfigurable static properties. User-initiated migration actions provide citizen technologists with the ability to relocate certain pipeline elements from their original node to another node which is realized by a step-wise migration scheme. As this effectively changes the pipeline execution topology, the system automatically restores respective event stream edges among logically adjacent pipeline elements. Lastly, system-initiated offloading actions use an offloading scheme to employ decision-making ability within the node controller based on the MAPE-K pattern. The initial decentralized approach is then combined with a central coordination to complete the relocation task by transforming the offloading problem into a migration problem. All adaptation types are integrated into Apache StreamPipes as extensions to both the central pipeline management and the node controller. The adaptation methodology and related adaptation types including reconfiguration, migration and offloading are evaluated by conceptually investigating corresponding system-specific requirements. Moreover, adaptation aspects, specifically the offloading time, migration time and downtime, is evaluated on the basis performance tests.

10.2 Significance

Our holistic management approach simplifies and improves the deployment and operation aspects of geo-distributed event-driven applications in heterogeneous fog infrastructures. The proposed models not only aid in creating understanding on available resources but allow to link related application-side requirements beyond typical hardware characteristics to provide better support throughout the application life cycle. In many application areas and industries, data-driven decision-making is increasingly democratized and non-technical experts are empowered to solve emerging application and business problems on their own. Especially the IoT and its industrial adoption in the IIoT pose new requirements to flexibly orchestrate event-driven applications in a geo-distributed manner in order to bring event processing capability to the edge of the network. From a technical point of view, our approach allows nodes to expose a node description which is leveraged for enhanced resource and application management. In addition, as node descriptions are extensible and configurable, e.g., in terms of generic domain-specific node tags, this allows to quickly adjust to new situations. From an application point of view, our holistic management approach is well suited for realizing a wide range of application models in fog computing from edge-based preprocessing to novel edge artificial intelligence applications. Our proposed preference-based deployment options and operation policies offer citizen technologists generic configuration options to self-reliantly configure and deploy processing pipelines that span the cloud-edge continuum to best account for application-specific needs. At the same time, the ability for pipeline adaptations at run-time ensures business continuity and allows for the event-driven application to evolve over time in view of occurring changes. Technical details on the geo-distribution, operation, adaptation and run-time management are completely hidden in the underlying managing middleware which facilitates to further drive the democratization movement.

10.3 Outlook

This thesis lays the foundations towards establishing a holistic application management for geo-distributed event-driven applications in heterogeneous fog infrastructures. Our introduced models, concepts and methods simplify and improve aspects of geo-distributed deployment, operation and adaptation for event-driven applications created by non-technical citizen technologists. Potential future research to bring further improvements are as follows:

Elastic Fog Pipelines. Once a pipeline is configured, i.e., individual pipeline elements are assigned to nodes, the pipeline graph and its respective elements alongside potential event stream relays are distributed in the fog infrastructure. In our current execution

model, every user-modeled pipeline element within a processing pipeline on the logical level is represented by exactly one run-time instance on the physical level. This can lead to situations where congested or slow operating run-time instances turn into bottlenecks which in turn slow down or even compromise the whole event-driven application. To mitigate such circumstances, concepts evolving around resource elasticity in resource-constrained fog environments need to be investigated. Thereby, elastic run-time instances that allow to grow or shrink in a load-dependent manner can be realized by means of instance replication or by allocating more node resources if feasible, e.g., by evicting other lower-prioritized run-time instances. Yet, unlike the cloud, the fog and edge layer mostly comprise physical nodes where dynamic provisioning and releasing of additional resources is time-consuming, inefficient and costly. Therefore, elastic event-driven applications in federated fog architectures present a promising research area that needs to be explored from various viewpoints.

Autonomic Self-Management. Our current application management approach follows a two-level design with a global, centralized coordinator and local, decentralized node controller. While it is arguably not feasible to completely decentralize all coordination and management activities, especially in view of the centralized pipeline modeling and orchestration by citizen technologists, specific activities can be outsourced and delegated to respective node controllers. Our provided offloading approach has shown the potential of node-local decision-making in the event of context changes without the need for human interaction. Yet, this requires further investigation. One idea is to extend node controllers with the ability to coordinate, observe and manage themselves in clusters of neighboring nodes by using decentralized gossip-based membership approaches in combination with network coordinates for proximity estimation. Individual clusters can elect leader node controllers serving the purpose of the central coordinator at a smaller scale in order to improve the coordination and run-time management aspects of pipeline elements, including autonomic reconfiguration or element handover in mobile scenarios. Therefore, more research is needed that centers around autonomic self-management concepts for event-driven applications in fog infrastructures.

Bibliography

Abadi, Daniel J.; Ahmad, Yanif; Balazinska, Magdalena; Cherniack, Mitch; Hwang, Jeong-hyon; Lindner, Wolfgang; Maskey, Anurag S.; Rasin, Er; Ryvkina, Esther; Tatbul, Nesime; Xing, Ying; Zdonik, Stan (2005). 'The Design of the Borealis Stream Processing Engine'. In: *In CIDR*, pp. 277–289.

Abadi, Daniel J.; Carney, Don; Çetintemel, Ugur; Cherniack, Mitch; Convey, Christian; Lee, Sangdon; Stonebraker, Michael; Tatbul, Nesime; Zdonik, Stan (2003). 'Aurora: A New Model and Architecture for Data Stream Management'. In: *The VLDB Journal — The International Journal on Very Large Data Bases* 12 (2), pp. 120–139. DOI: 10.1007/s00778-003-0095-z.

Ahmed, Arif; Pierre, Guillaume (2018). 'Docker Container Deployment in Fog Computing Infrastructures'. In: *2018 IEEE International Conference on Edge Computing (EDGE)*, pp. 1–8. DOI: 10.1109/EDGE.2018.00008.

Alam, Muhammad; Rufino, Joao; Ferreira, Joaquim; Ahmed, Syed Hassan; Shah, Nadir; Chen, Yuanfang (2018). 'Orchestration of Microservices for IoT Using Docker and Edge Computing'. In: *IEEE Communications Magazine* 56 (9), pp. 118–123. DOI: 10.1109/MCOM.2018.1701233.

Alcaraz, Cristina (2019). 'Secure Interconnection of IT-OT Networks in Industry 4.0'. In: *Critical Infrastructure Security and Resilience: Theories, Methods, Tools and Technologies*. Ed. by Dimitris Gritzalis; Marianthi Theocharidou; George Stergiopoulos. Advanced Sciences and Technologies for Security Applications. Cham: Springer International Publishing, pp. 201–217. ISBN: 978-3-030-00024-0. DOI: 10.1007/978-3-030-00024-0_11.

Alexander, Christopher (1979). *The Timeless Way of Building*. New York: Oxford University Press. ISBN: 0-19-502402-8.

Alexander, Christopher; Ishikawa, Sara; Silverstein, Murray (1977). *A Pattern Language: Towns, Buildings, Construction*. New York: Oxford University Press. ISBN: 0-19-501919-9.

Alshuqayran, N.; Ali, N.; Evans, R. (2016). 'A Systematic Mapping Study in Microservice Architecture'. In: *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pp. 44–51. DOI: 10.1109/SOCA.2016.15.

An, Kyoungho; Khare, Shweta; Gokhale, Aniruddha; Hakiri, Akram (2017). 'An Autonomous and Dynamic Coordination and Discovery Service for Wide-Area Peer-to-Peer Publish/Subscribe: Experience Paper'. In: *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*. DEBS '17. New York, NY, USA: Association for Computing Machinery, pp. 239–248. DOI: 10.1145/3093742.3093910.

Andrade, Henrique C. M.; Gedik, Bugra; Turaga, Deepak S. (2014). *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. 1st. USA: Cambridge University Press. ISBN: 978-1-107-01554-8.

Arasu, Arvind; Babu, Shivnath; Widom, Jennifer (2006). 'The CQL Continuous Query Language: Semantic Foundations and Query Execution'. In: *The VLDB Journal — The*

International Journal on Very Large Data Bases 15 (2), pp. 121–142. DOI: 10.1007/s00778-004-0147-z.

Armbrust, Michael; Xin, Reynold S.; Lian, Cheng; Huai, Yin; Liu, Davies; Bradley, Joseph K.; Meng, Xiangrui; Kaftan, Tomer; Franklin, Michael J.; Ghodsi, Ali; Zaharia, Matei (2015). 'Spark SQL: Relational Data Processing in Spark'. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, pp. 1383–1394. DOI: 10.1145/2723372.2742797.

Babcock, Brian; Babu, Shivnath; Datar, Mayur; Motwani, Rajeev; Widom, Jennifer (2002). 'Models and Issues in Data Stream Systems'. In: *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS '02. New York, NY, USA: Association for Computing Machinery, pp. 1–16. DOI: 10.1145/543613.543615.

Babu, Shivnath; Widom, Jennifer (2001). 'Continuous Queries over Data Streams'. In: *ACM SIGMOD Record* 30 (3), pp. 109–120. DOI: 10.1145/603867.603884.

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

Balan, Rajesh; Flinn, Jason; Satyanarayanan, M.; Sinnamohideen, Shafeeq; Yang, Hen-I (2002). 'The Case for Cyber Foraging'. In: *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*. EW 10. New York, NY, USA: Association for Computing Machinery, pp. 87–92. DOI: 10.1145/1133373.1133390.

Baldini, Ioana; Castro, Paul; Chang, Kerry; Cheng, Perry; Fink, Stephen; Ishakian, Vatche; Mitchell, Nick; Muthusamy, Vinod; Rabbah, Rodric; Slominski, Aleksander; Suter, Philippe (2017). 'Serverless Computing: Current Trends and Open Problems'. In: *Research Advances in Cloud Computing*. Ed. by Sanjay Chaudhary; Gaurav Somani; Rajkumar Buyya. Singapore: Springer, pp. 1–20. ISBN: 978-981-10-5026-8. DOI: 10.1007/978-981-10-5026-8_1.

Barr, Jeff (2006). *Amazon EC2 Beta*. https://aws.amazon.com/blogs/aws/amazon_ec2_beta/. (Online: accessed 2021-03-01).

Bates, John; Bacon, Jean; Moody, Ken; Spiteri, Mark (1998). 'Using Events for the Scalable Federation of Heterogeneous Components'. In: *Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*. EW 8. New York, NY, USA: Association for Computing Machinery, pp. 58–65. DOI: 10.1145/319195.319205.

Baun, Christian; Kunze, Marcel; Nimis, Jens; Tai, Stefan (2011). *Cloud Computing: Web-Based Dynamic IT Services*. Springer Verlag. ISBN: 978-3-642-20916-1. DOI: 10.1007/978-3-642-20917-8.

Begoli, Edmon; Akidau, Tyler; Hueske, Fabian; Hyde, Julian; Knight, Kathryn; Knowles, Kenneth (2019). 'One SQL to Rule Them All - an Efficient and Syntactically Idiomatic Approach to Management of Streams and Tables'. In: *Proceedings of the 2019 International*

Conference on Management of Data. SIGMOD '19. New York, NY, USA: Association for Computing Machinery, pp. 1757–1772. DOI: 10.1145/3299869.3314040.

Bellavista, Paolo; Zanni, Alessandro (2017). 'Feasibility of Fog Computing Deployment Based on Docker Containerization over RaspberryPi'. In: *Proceedings of the 18th International Conference on Distributed Computing and Networking - ICDCN '17*, pp. 1–10. DOI: 10.1145/3007748.3007777.

Bergmayr, Alexander; Breitenbücher, Uwe; Ferry, Nicolas; Rossini, Alessandro; Solberg, Arnor; Wimmer, Manuel; Kappel, Gerti; Leymann, Frank (2018). 'A Systematic Review of Cloud Modeling Languages'. In: *ACM Computing Surveys* 51 (1), 22:1–22:38. DOI: 10.1145/3150227.

Bermbach, David; Pallas, Frank; Pérez, David García; Plebani, Pierluigi; Anderson, Maya; Kat, Ronen; Tai, Stefan (2018). 'A Research Perspective on Fog Computing'. In: *Service-Oriented Computing – ICSOC 2017 Workshops*. Ed. by Lars Braubach; Juan M. Murillo; Nima Kaviani; Manuel Lama; Loli Burgueño; Naouel Moha; Marc Oriol. Lecture Notes in Computer Science. Cham: Springer International Publishing, pp. 198–210. DOI: 10.1007/978-3-319-91764-1_16.

Bernstein, D. (2014). 'Containers and Cloud: From LXC to Docker to Kubernetes'. In: *IEEE Cloud Computing* 1 (3), pp. 81–84. DOI: 10.1109/MCC.2014.51.

Bhattacharjee, S. (2018). *Practical Industrial Internet of Things Security: A Practitioner's Guide to Securing Connected Industries*. Packt Publishing. ISBN: 978-1-78883-085-0.

Binz, Tobias; Breitenbücher, Uwe; Haupt, Florian; Kopp, Oliver; Leymann, Frank; Nowak, Alexander; Wagner, Sebastian (2013). 'OpenTOSCA – A Runtime for TOSCA-Based Cloud Applications'. In: *Service-Oriented Computing*. Ed. by Samik Basu; Cesare Pautasso; Liang Zhang; Xiang Fu. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 692–695. DOI: 10.1007/978-3-642-45005-1_62.

Birman, K.; Joseph, T. (1987). 'Exploiting Virtual Synchrony in Distributed Systems'. In: *ACM SIGOPS Operating Systems Review* 21 (5), pp. 123–138. DOI: 10.1145/37499.37515.

Birman, Kenneth P. (1993). 'The Process Group Approach to Reliable Distributed Computing'. In: *Communications of the ACM* 36 (12), pp. 37–53. DOI: 10.1145/163298.163303.

Bittencourt, L. F.; Lopes, M. M.; Petri, I.; Rana, O. F. (2015). 'Towards Virtual Machine Migration in Fog Computing'. In: *2015 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*, pp. 1–8. DOI: 10.1109/3PGCIC.2015.85.

Blackstock, Michael; Lea, Rodger (2014). 'Toward a Distributed Data Flow Platform for the Web of Things (Distributed Node-RED)'. In: *Proceedings of the 5th International Workshop on Web of Things*. WoT '14. New York, NY, USA: Association for Computing Machinery, pp. 34–39. DOI: 10.1145/2684432.2684439.

Bonomi, Flavio; Milito, Rodolfo; Natarajan, Preethi; Zhu, Jiang (2014). 'Fog Computing: A Platform for Internet of Things and Analytics'. In: *Studies in Computational Intelligence* 546, pp. 169–186. DOI: 10.1007/978-3-319-05029-4_7.

Bonomi, Flavio; Milito, Rodolfo; Zhu, Jiang; Addepalli, Sateesh (2012). 'Fog Computing and Its Role in the Internet of Things'. In: *Proceedings of the first edition of the MCC*

workshop on Mobile cloud computing, pp. 13–16. DOI: 10.1145/2342509.2342513. arXiv: 1502.01815v3.

Brinker, Scott (2018). *Democratizing Martech: Distributing Power from IT to Marketing Technologists to Everyone*. https://chiefmartec.com/2018/05/democratizing-martech-marketing-technologists/. (Online: accessed 2021-03-19).

Brogi, Antonio; Mencagli, Gabriele; Neri, Davide; Soldani, Jacopo; Torquati, Massimo (2018). 'Container-Based Support for Autonomic Data Stream Processing Through the Fog'. In: *Euro-Par 2017: Parallel Processing Workshops*, pp. 17–28. DOI: 10.1007/978-3-319-75178-8_2.

Brun, Yuriy; Di Marzo Serugendo, Giovanna; Gacek, Cristina; Giese, Holger; Kienle, Holger; Litoiu, Marin; Müller, Hausi; Pezzè, Mauro; Shaw, Mary (2009). 'Engineering Self-Adaptive Systems through Feedback Loops'. In: *Software Engineering for Self-Adaptive Systems*. Ed. by Betty H. C. Cheng; Rogério de Lemos; Holger Giese; Paola Inverardi; Jeff Magee. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 48–70. ISBN: 978-3-642-02161-9. DOI: 10.1007/978-3-642-02161-9_3.

Bruns, Ralf; Dunkel, Jürgen (2010). *Event-Driven Architecture: Softwarearchitektur für ereignisgesteuerte Geschäftsprozesse*. Xpert.press. Berlin Heidelberg: Springer-Verlag. ISBN: 978-3-642-02438-2. DOI: 10.1007/978-3-642-02439-9.

Buchmann, Alejandro; Koldehofe, Boris (2009). 'Complex Event Processing'. In: *it - Information Technology* 51 (5), pp. 241–242. DOI: 10.1524/itit.2009.9058.

Burns, Brendan; Grant, Brian; Oppenheimer, David; Brewer, Eric; Wilkes, John (2016). 'Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade'. In: *Queue* 14 (1), pp. 70–93. DOI: 10.1145/2898442.2898444.

Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter; Stal, Michael (1996). *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing. ISBN: 0-471-95869-7.

Byers, C. C. (2017). 'Architectural Imperatives for Fog Computing: Use Cases, Requirements, and Architectural Techniques for Fog-Enabled IoT Networks'. In: *IEEE Communications Magazine* 55 (8), pp. 14–20. DOI: 10.1109/MCOM.2017.1600885.

Cardellini, Valeria; Lo Presti, Francesco; Nardelli, Matteo; Russo Russo, Gabriele (2018). 'Decentralized Self-Adaptation for Elastic Data Stream Processing'. In: *Future Generation Computer Systems* 87, pp. 171–185. DOI: 10.1016/J.FUTURE.2018.05.025.

Carney, Don; Çetintemel, Uğur; Cherniack, Mitch; Convey, Christian; Lee, Sangdon; Seidman, Greg; Stonebraker, Michael; Tatbul, Nesime; Zdonik, Stan (2002). 'Monitoring Streams: A New Class of Data Management Applications'. In: *Proceedings of the 28th International Conference on Very Large Data Bases*. VLDB '02. Hong Kong, China: VLDB Endowment, pp. 215–226.

Carzaniga, Antonio; Di Nitto, Elisabetta; Rosenblum, David S.; Wolf, Alexander L. (1998). 'Issues in Supporting Event-Based Architectural Styles'. In: *Proceedings of the Third International Workshop on Software Architecture*. ISAW '98. New York, NY, USA: Association for Computing Machinery, pp. 17–20. DOI: 10.1145/288408.288413.

Casale, G.; Artač, M.; van den Heuvel, W.-J.; van Hoorn, A.; Jakovits, P.; Leymann, F.; Long, M.; Papanikolaou, V.; Presenza, D.; Russo, A.; Srirama, S. N.; Tamburri, D. A.; Wurster, M.; Zhu, L. (2020). 'RADON: Rational Decomposition and Orchestration for Serverless Computing'. In: *SICS Software-Intensive Cyber-Physical Systems* 35 (1), pp. 77–87. DOI: 10.1007/s00450-019-00413-w.

Challita, Stéphanie; Korte, Fabian; Erbel, Johannes; Zalila, Faiez; Grabowski, Jens; Merle, Philippe (2021). 'Model-Based Cloud Resource Management with TOSCA and OCCI'. In: *Software and Systems Modeling*. DOI: 10.1007/s10270-021-00869-y.

Chandrasekaran, Sirish; Cooper, Owen; Deshpande, Amol; Franklin, Michael J.; Hellerstein, Joseph M.; Hong, Wei; Krishnamurthy, Sailesh; Madden, Samuel R.; Reiss, Fred; Shah, Mehul A. (2003). 'TelegraphCQ: Continuous Dataflow Processing'. In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. SIGMOD '03. New York, NY, USA: Association for Computing Machinery, p. 668. DOI: 10.1145/872757.872857.

Chandy, K.; Schulte, W. (2009). *Event Processing: Designing IT Systems for Agile Companies*. First. USA: McGraw-Hill, Inc. ISBN: 978-0-07-163350-5.

Chandy, Mani K.; Schulte, Roy (2007). *What Is Event Driven Architecture (EDA) and Why Does It Matter?* https://complexevents.com/2007/07/17/what-is-event-driven-architecture-eda-and-why-does-it-matter/. (Online: accessed 2021-01-30).

Chang, Chii; Srirama, Satish Narayana; Buyya, Rajkumar (2017). 'Indie Fog: An Efficient Fog-Computing Infrastructure for the Internet of Things'. In: *Computer* 50 (9), pp. 92–98. DOI: 10.1109/MC.2017.3571049.

Chen, B.; Wan, J.; Celesti, A.; Li, D.; Abbas, H.; Zhang, Q. (2018). 'Edge Computing in IoT-Based Manufacturing'. In: *IEEE Communications Magazine* 56 (9), pp. 103–109. DOI: 10.1109/MCOM.2018.1701231.

Cheng, B.; Papageorgiou, A.; Cirillo, F.; Kovacs, E. (2015). 'GeeLytics: Geo-Distributed Edge Analytics for Large Scale IoT Systems Based on Dynamic Topology'. In: *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pp. 565–570. DOI: 10.1109/WF-IoT.2015.7389116.

Cheng, Bin; Solmaz, Gürkan; Cirillo, Flavio; Kovacs, Ernö; Terasawa, Kazuyuki; Kitazawa, Atsushi (2018). 'FogFlow: Easy Programming of IoT Services Over Cloud and Edges for Smart Cities'. In: *IEEE Internet of Things Journal* 5 (2), pp. 696–707. DOI: 10.1109/JIOT.2017.2747214.

Chiang, M.; Ha, S.; Risso, F.; Zhang, T.; Chih-Lin, I. (2017). 'Clarifying Fog Computing and Networking: 10 Questions and Answers'. In: *IEEE Communications Magazine* 55 (4), pp. 18–20. DOI: 10.1109/MCOM.2017.7901470.

Costa, P.; Picco, G. P.; Rossetto, S. (2005). 'Publish-Subscribe on Sensor Networks: A Semi-Probabilistic Approach'. In: *IEEE International Conference on Mobile Adhoc and Sensor Systems Conference, 2005.* 10 pp.-332. DOI: 10.1109/MAHSS.2005.1542816.

Cristian, Flavin (1991). 'Understanding Fault-Tolerant Distributed Systems'. In: *Communications of the ACM* 34 (2), pp. 56–78. DOI: 10.1145/102792.102801.

Cugola, Gianpaolo; Margara, Alessandro (2012). 'Processing Flows of Information: From Data Stream to Complex Event Processing'. In: *ACM Computing Surveys* 44 (3), 15:1–15:62. DOI: 10.1145/2187671.2187677.

Dabek, Frank; Cox, Russ; Kaashoek, Frans; Morris, Robert (2004). 'Vivaldi: A Decentralized Network Coordinate System'. In: *ACM SIGCOMM Computer Communication Review* 34 (4), pp. 15–26. DOI: 10.1145/1030194.1015471.

Dasher, Richard B. (2019). *The Present and Future of Edge Computing from an International Perspective*. https://asia.stanford.edu/wp-content/uploads/190926-RBD-402a-slides.pdf. (Online: accessed 2021-11-11).

Dastjerdi, Amir Vahid; Gupta, Harshit; Calheiros, Rodrigo N.; Ghosh, Soumya K.; Buyya, Rajkumar (2016). 'Fog Computing: Principles, Architectures, and Applications'. In: *Internet of Things*. Ed. by Rajkumar Buyya; Amir Vahid Dastjerdi. Morgan Kaufmann, pp. 61–75. ISBN: 978-0-12-805395-9. DOI: 10.1016/B978-0-12-805395-9.00004-6.

Dautov, R.; Distefano, S.; Bruneo, D.; Longo, F.; Merlino, G.; Puliafito, A. (2018). 'Data Processing in Cyber-Physical-Social Systems Through Edge Computing'. In: *IEEE Access* 6, pp. 29822–29835. DOI: 10.1109/ACCESS.2018.2839915.

Dayal, Umeshwar; Buchmann, Alejandro P.; McCarthy, Dennis R. (1988). 'Rules Are Objects Too: A Knowledge Model for an Active, Object-Oriented Database System'. In: *Advances in Object-Oriented Database Systems*. Ed. by Klaus R. Dittrich. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 129–143. DOI: 10.1007/3-540-50345-5_9.

Dayarathna, Miyuru; Perera, Srinath (2018). 'Recent Advancements in Event Processing'. In: *ACM Computing Surveys* 51 (2), 33:1–33:36. DOI: 10.1145/3170432.

Dean, Jeffrey; Ghemawat, Sanjay (2004). 'MapReduce: Simplified Data Processing on Large Clusters'. In: *Proc. of the OSDI - Symp. on Operating Systems Design and Implementation*. USENIX, pp. 137–149.

Dennis, Jack B.; Misunas, David P. (1974). 'A Preliminary Architecture for a Basic Data-Flow Processor'. In: *ACM SIGARCH Computer Architecture News* 3 (4), pp. 126–132. DOI: 10.1145/641675.642111.

Dias de Assunção, Marcos; da Silva Veith, Alexandre; Buyya, Rajkumar (2018). 'Distributed Data Stream Processing and Edge Computing: A Survey on Resource Elasticity and Future Directions'. In: *Journal of Network and Computer Applications* 103, pp. 1–17. DOI: 10.1016/j.jnca.2017.12.001.

Díaz-de-Arcaya, Josu; Miñón, Raúl; Torre-Bastida, Ana I.; Del Ser, Javier; Almeida, Aitor (2020). 'PADL: A Modeling and Deployment Language for Advanced Analytical Services'. In: *Sensors* 20 (23), p. 6712. DOI: 10.3390/s20236712.

Dilley, John; Maggs, Bruce; Parikh, Jay; Prokop, Harald; Sitaraman, Ramesh; Weihl, Bill (2002). 'Globally Distributed Content Delivery'. In: *IEEE Internet Computing* 6 (5), pp. 50–58. DOI: 10.1109/MIC.2002.1036038.

Dupont, Corentin; Giaffreda, Raffaele; Capra, Luca (2017). 'Edge Computing in IoT Context: Horizontal and Vertical Linux Container Migration'. In: *GIoTS 2017 - Global Internet of Things Summit, Proceedings*. DOI: 10.1109/GIOTS.2017.8016218.

ElMaraghy, Hoda; Wiendahl, Hans-Peter (2014). 'Changeable Manufacturing'. In: *CIRP Encyclopedia of Production Engineering*. Ed. by Luc Laperrière; Gunther Reinhart. Berlin, Heidelberg: Springer, pp. 157–163. ISBN: 978-3-642-20617-7. DOI: 10.1007/978-3-642-20617-7_6674.

Endres, Christian; Breitenbücher, Uwe; Falkenthal, Michael; Kopp, Oliver; Leymann, Frank; Wettinger, Johannes (2017). 'Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications'. In: *Proceedings of the 9th International Conference on Pervasive Patterns and Applications*. Xpert Publishing Services (XPS), pp. 22–27.

Etzion, Opher; Niblett, Peter (2010). *Event Processing in Action*. Manning Publications Co. ISBN: 978-1-935182-21-4.

Eugster, Patrick Th.; Felber, Pascal A.; Guerraoui, Rachid; Kermarrec, Anne-Marie (2003). 'The Many Faces of Publish/Subscribe'. In: *ACM Computing Surveys* 35 (2), pp. 114–131. DOI: 10.1145/857076.857078.

Fiege, Ludger; Mühl, Gero; Gärtner, Felix C. (2002). 'Modular Event-Based Systems'. In: *The Knowledge Engineering Review* 17 (4), pp. 359–388. DOI: 10.1017/S0269888903000559.

Fischer, G.; Nakakoji, K.; Ye, Y. (2009). 'Metadesign: Guidelines for Supporting Domain Experts in Software Development'. In: *IEEE Software* 26 (5), pp. 37–44. DOI: 10.1109/MS.2009.134.

Flinn, Jason; Satyanarayanan, M. (1999). 'Energy-Aware Adaptation for Mobile Applications'. In: *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*. SOSP '99. New York, NY, USA: Association for Computing Machinery, pp. 48–63. DOI: 10.1145/319151.319155.

Franklin, Michael; Zdonik, Stan (1998). '"Data in Your Face": Push Technology in Perspective'. In: *ACM SIGMOD Record* 27 (2), pp. 516–519. DOI: 10.1145/276305.276360.

Franklin, Michael; Zdonik, Stanley (1997). 'A Framework for Scalable Dissemination-Based Systems'. In: *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '97. New York, NY, USA: Association for Computing Machinery, pp. 94–105. DOI: 10.1145/263698.263725.

Al-Fuqaha, A.; Guizani, M.; Mohammadi, M.; Aledhari, M.; Ayyash, M. (Fourthquarter 2015). 'Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications'. In: *IEEE Communications Surveys Tutorials* 17 (4), pp. 2347–2376. DOI: 10.1109/COMST.2015.2444095.

Gamma, Erich; Helm, Richard; Johnson, Ralph E.; Vlissides, John M. (1993). 'Design Patterns: Abstraction and Reuse of Object-Oriented Design'. In: *Proceedings of the 7th European Conference on Object-Oriented Programming*. ECOOP '93. Berlin, Heidelberg: Springer-Verlag, pp. 406–431. ISBN: 978-3-540-57120-9.

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John M. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional. ISBN: 0-201-63361-2.

Garcia Lopez, Pedro; Montresor, Alberto; Epema, Dick; Datta, Anwitaman; Higashino, Teruo; Iamnitchi, Adriana; Barcellos, Marinho; Felber, Pascal; Riviere, Etienne (2015).

'Edge-Centric Computing: Vision and Challenges'. In: *ACM SIGCOMM Computer Communication Review* 45 (5), pp. 37–42. DOI: 10.1145/2831347.2831354.

Giang, Nam Ky (2019). 'Context-Dependent Exogenous Coordination for Building Large Scale, Dynamic Fog Computing Applications'. PhD thesis. University of British Columbia. DOI: 10.14288/1.0383394.

Giang, Nam Ky; Blackstock, Michael; Lea, Rodger; Leung, Victor C.M. (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, pp. 155–162. DOI: 10.1109/IOT.2015.7356560.

Greengard, Samuel (2020). 'AI on Edge'. In: *Communications of the ACM* 63 (9), pp. 18–20. DOI: 10.1145/3409977.

Gröger, Christoph (2018). 'Building an Industry 4.0 Analytics Platform'. In: *Datenbank-Spektrum* 18 (1), pp. 5–14. DOI: 10.1007/s13222-018-0273-1.

Guazzelli, Alex; Zeller, Michael; Lin, Wen-Ching; Williams, Graham (2009). 'PMML: An Open Standard for Sharing Models'. In: *The R Journal* 1 (1), pp. 60–65. DOI: 10.32614/RJ-2009-010.

Hao, Z.; Novak, E.; Yi, S.; Li, Q. (2017). 'Challenges and Software Architecture for Fog Computing'. In: *IEEE Internet Computing* 21 (2), pp. 44–53. DOI: 10.1109/MIC.2017.26.

Hasenburg, Jonathan; Bermbach, David (2020). 'DisGB: Using Geo-Context Information for Efficient Routing in Geo-Distributed Pub/Sub Systems'. In: *2020 IEEE/ACM International Conference on Utility and Cloud Computing*. Leicester, United Kingdom: IEEE.

Heinze, Thomas; Aniello, Leonardo; Querzoni, Leonardo; Jerzak, Zbigniew (2014). 'Cloud-Based Data Stream Processing'. In: *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. DEBS '14. New York, NY, USA: Association for Computing Machinery, pp. 238–245. DOI: 10.1145/2611286.2611309.

Henke, Nicolaus; Bughin, Jacques; Chui, Michael; Manyika, James; Saleh, Tamim; Wiseman, Bill; Sethupathy, Guru (2016). *The Age of Analytics: Competing in a Data-Driven World*. Tech. rep. McKinsey Global Institute.

Herle, Stefan; Blankenbach, Jörg (2016). 'GeoPipes Using GeoMQTT'. In: *Geospatial Data in a Changing World*. Ed. by Tapani Sarjakoski; Maribel Yasmina Santos; L. Tiina Sarjakoski. Lecture Notes in Geoinformation and Cartography. Cham: Springer International Publishing, pp. 383–398. DOI: 10.1007/978-3-319-33783-8_22.

Hevner, Alan R.; March, Salvatore T.; Park, Jinsoo; Ram, Sudha (2004). 'Design Science in Information Systems Research'. In: *MIS Quarterly* 28 (1), pp. 75–105. DOI: 10.2307/25148625.

Hießl, Thomas; Hochreiner, Christoph; Schulte, Stefan (2019). 'Towards a Framework for Data Stream Processing in the Fog'. In: *Informatik Spektrum* 42 (4), pp. 256–265. DOI: 10.1007/s00287-019-01192-z.

Hirzel, Martin (2012). 'Partition and Compose: Parallel Complex Event Processing'. In: *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*.

DEBS '12. New York, NY, USA: Association for Computing Machinery, pp. 191–200. DOI: 10.1145/2335484.2335506.

Hochreiner, C.; Schulte, S.; Dustdar, S.; Lecue, F. (2015). 'Elastic Stream Processing for Distributed Environments'. In: *IEEE Internet Computing* 19 (6), pp. 54–59. DOI: 10.1109/MIC.2015.118.

Hochreiner, Christoph; Vogler, Michael; Schulte, Stefan; Dustdar, Schahram (2016). 'Elastic Stream Processing for the Internet of Things'. In: *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. IEEE, pp. 100–107. DOI: 10.1109/CLOUD.2016.0023.

Hong, Cheol-Ho; Varghese, Blesson (2019). 'Resource Management in Fog/Edge Computing: A Survey on Architectures, Infrastructure, and Algorithms'. In: *ACM Computing Surveys* 52 (5), pp. 1–37. DOI: 10.1145/3326066.

Hong, Kirak; Lillethun, David; Ramachandran, Umakishore; Ottenwälder, Beate; Koldehofe, Boris (2013). *Mobile Fog: A Programming Model for Large-Scale Applications on the Internet of Things*. ISBN: 978-1-4503-2180-8.

Hou, X.; Li, Y.; Chen, M.; Wu, D.; Jin, D.; Chen, S. (2016). 'Vehicular Fog Computing: A Viewpoint of Vehicles as the Infrastructures'. In: *IEEE Transactions on Vehicular Technology* 65 (6), pp. 3860–3873. DOI: 10.1109/TVT.2016.2532863.

Hu, Pengfei; Dhelim, Sahraoui; Ning, Huansheng; Qiu, Tie (2017). 'Survey on Fog Computing: Architecture, Key Technologies, Applications and Open Issues'. In: *Journal of Network and Computer Applications* 98, pp. 27–42. DOI: 10.1016/j.jnca.2017.09.002.

Hu, Yun Chao; Patel, Milan; Sabella, Dario; Sprecher, Nurit; Young, Valerie (2015). 'Mobile Edge Computing: A Key Technology towards 5G'. In: *ETSI White Paper* 11 (1).

Huang, Yongqiang; Garcia-Molina, Hector (2004). 'Publish/Subscribe in a Mobile Environment'. In: *Wireless Networks* 10 (6), pp. 643–652. DOI: 10.1023/B:WINE.0000044025.64654.65.

Hueske, F.; Kalavri, V. (2019). *Stream Processing with Apache Flink: Fundamentals, Implementation, and Operation of Streaming Applications*. O'Reilly Media, Incorporated. ISBN: 978-1-4919-7429-2.

Hunkeler, U.; Truong, H. L.; Stanford-Clark, A. (2008). 'MQTT-S — A Publish/Subscribe Protocol for Wireless Sensor Networks'. In: *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE '08)*, pp. 791–798. DOI: 10.1109/COMSWA.2008.4554519.

Hutter, Frank; Kotthoff, Lars; Vanschoren, Joaquin, eds. (2019). *Automated Machine Learning: Methods, Systems, Challenges*. The Springer Series on Challenges in Machine Learning. Springer International Publishing. ISBN: 978-3-030-05317-8. DOI: 10.1007/978-3-030-05318-5.

Iorga, Michaela; Feldman, Larry; Barton, Robert; Martin, Michael J; Goren, Ned; Mahmoudi, Charif (2018). 'Fog Computing Conceptual Model'. In: *NIST Special Publication 500-325*. DOI: 10.6028/NIST.SP.500-325.

Ismail, Bukhary Ikhwan; Mostajeran Goortani, Ehsan; Ab Karim, Mohd Bazli; Ming Tat, Wong; Setapa, Sharipah; Luke, Jing Yuan; Hong Hoe, Ong (2015). 'Evaluation of Docker

as Edge Computing Platform'. In: *ICOS 2015 - 2015 IEEE Conference on Open Systems*. IEEE, pp. 130–135. DOI: 10.1109/ICOS.2015.7377291.

Issarny, Valérie; Bouloukakis, Georgios; Georgantas, Nikolaos; Billet, Benjamin (2016). 'Revisiting Service-Oriented Architecture for the IoT: A Middleware Perspective'. In: *Service-Oriented Computing*. Ed. by Quan Z. Sheng; Eleni Stroulia; Samir Tata; Sami Bhiri. Lecture Notes in Computer Science. Cham: Springer International Publishing, pp. 3–17. DOI: 10.1007/978-3-319-46295-0_1.

Jeschke, Sabina; Brecher, Christian; Meisen, Tobias; Özdemir, Denis; Eschert, Tim (2017). 'Industrial Internet of Things and Cyber Manufacturing Systems'. In: *Industrial Internet of Things: Cybermanufacturing Systems*. Ed. by Sabina Jeschke; Christian Brecher; Houbing Song; Danda B. Rawat. Springer Series in Wireless Technology. Cham: Springer International Publishing, pp. 3–19. ISBN: 978-3-319-42559-7. DOI: 10.1007/978-3-319-42559-7_1.

Joerss, Martin; Schröder, Jürgen; Neuhaus, Florian; Klink, Christoph; Mann, Florian (2016). *Parcel Delivery: The Future of Last Mile*. Tech. rep. McKinsey & Company.

Kai, Kang; Cong, Wang; Tao, Luo (2016). 'Fog Computing for Vehicular Ad-Hoc Networks: Paradigms, Scenarios, and Issues'. In: *The Journal of China Universities of Posts and Telecommunications* 23 (2), pp. 56–96. DOI: 10.1016/S1005-8885(16)60021-3.

Kang, H.; Le, M.; Tao, S. (2016). 'Container and Microservice Driven Design for Cloud Infrastructure DevOps'. In: *2016 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 202–211. DOI: 10.1109/IC2E.2016.26.

Karagiannis, V.; Schulte, S. (2020). 'Comparison of Alternative Architectures in Fog Computing'. In: *2020 IEEE 4th International Conference on Fog and Edge Computing (ICFEC)*, pp. 19–28. DOI: 10.1109/ICFEC50348.2020.00010.

Karamoozian, Amir; Hafid, Abdelhakim; Aboulhamid, El Mostapha (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)*. Rome, Italy: IEEE, pp. 166–172. DOI: 10.1109/FMEC.2019.8795320.

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

Kephart, J. O.; Chess, D. M. (2003). 'The Vision of Autonomic Computing'. In: *Computer* 36 (1), pp. 41–50. DOI: 10.1109/MC.2003.1160055.

Kleinfeld, Robert; Steglich, Stephan; Radziwonowicz, Lukasz; Doukas, Charalampos (2014). 'Glue.Things: A Mashup Platform for Wiring the Internet of Things with the Internet of Services'. In: *Proceedings of the 5th International Workshop on Web of Things*. WoT '14. New York, NY, USA: Association for Computing Machinery, pp. 16–21. DOI: 10.1145/2684432.2684436.

Kruchten, Philippe (1995). 'Architectural Blueprints–The "4+1" View Model of Software Architecture'. In: *IEEE Software* 12 (6), pp. 42–50.

Lee, E. A. (2008). 'Cyber Physical Systems: Design Challenges'. In: *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pp. 363–369. DOI: 10.1109/ISORC.2008.25.

Lewis, James; Fowler, Martin (2014). *Microservices: A Definition of This New Architectural Term*. https://martinfowler.com/articles/microservices.html. (Online: accessed 2021-02-01).

Liang, K.; Zhao, L.; Chu, X.; Chen, H. (2017). 'An Integrated Architecture for Software Defined and Virtualized Radio Access Networks with Fog Computing'. In: *IEEE Network* 31 (1), pp. 80–87. DOI: 10.1109/MNET.2017.1600027NM.

Luan, Tom H.; Gao, Longxiang; Li, Zhi; Xiang, Yang; Wei, Guiyi; Sun, Limin (2016). 'Fog Computing: Focusing on Mobile Users at the Edge'. In: *arXiv:1502.01815 [cs]*. arXiv: 1502.01815 [cs].

Luckham, D. C.; Vera, J. (1995). 'An Event-Based Architecture Definition Language'. In: *IEEE Transactions on Software Engineering* 21 (9), pp. 717–734. DOI: 10.1109/32.464548.

Luckham, David (2020). *What's the Difference Between ESP and CEP?* https://complexevents.com/2020/06/15/whats-the-difference-between-esp-and-cep-2/. (Online: accessed 2021-01-17).

Luckham, David C. (2002). *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 978-0-201-72789-0.

Luthra, Manisha; Koldehofe, Boris (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*. Davis, CA, USA: ACM Press, pp. 7–12. DOI: 10.1145/3366613.3368121.

Madakam, Somayya; Bhagat, Pratima (2018). 'Fog Computing in the IoT Environment: Principles, Features, and Models'. In: *Fog Computing: Concepts, Frameworks and Technologies*. Ed. by Zaigham Mahmood. Cham: Springer International Publishing, pp. 23–43. ISBN: 978-3-319-94890-4. DOI: 10.1007/978-3-319-94890-4_2.

Mahapatra, Tanmaya (2019). 'High-Level Graphical Programming for Big Data Applications'. PhD thesis. München: Technische Universität München.

Mahapatra, Tanmaya; Prehofer, Christian (2019). 'aFlux: Graphical Flow-Based Data Analytics'. In: *Software Impacts* 2, p. 100007. DOI: 10.1016/j.simpa.2019.100007.

Mahmood, Zaigham; Ramachandran, Muthu (2018). 'Fog Computing: Concepts, Principles and Related Paradigms'. In: *Fog Computing: Concepts, Frameworks and Technologies*. Ed. by Zaigham Mahmood. Cham: Springer International Publishing, pp. 3–21. ISBN: 978-3-319-94890-4. DOI: 10.1007/978-3-319-94890-4_1.

Mahmud, Redowan; Ramamohanarao, Kotagiri; Buyya, Rajkumar (2020). 'Application Management in Fog Computing Environments: A Taxonomy, Review and Future Directions'. In: *ACM Computing Surveys* 53 (4), 88:1–88:43. DOI: 10.1145/3403955.

Masip-Bruin, X.; Marín-Tordera, E.; Tashakor, G.; Jukan, A.; Ren, G. (2016). 'Foggy Clouds and Cloudy Fogs: A Real Need for Coordinated Management of Fog-to-Cloud Com-

puting Systems'. In: *IEEE Wireless Communications* 23 (5), pp. 120–128. DOI: 10.1109/MWC.2016.7721750.

Matheson, Eloise; Minto, Riccardo; Zampieri, Emanuele G. G.; Faccio, Maurizio; Rosati, Giulio (2019). 'Human–Robot Collaboration in Manufacturing Applications: A Review'. In: *Robotics* 8 (4), p. 100. DOI: 10.3390/robotics8040100.

Mayer, Ruben; Gupta, Harshit; Saurez, Enrique; Ramachandran, Umakishore (2017). 'FogStore: Toward a Distributed Data Store for Fog Computing'. In: *2017 IEEE Fog World Congress (FWC)*, pp. 1–6. DOI: 10.1109/FWC.2017.8368524.

McCarthy, Dennis; Dayal, Umeshwar (1989). 'The Architecture of an Active Database Management System'. In: *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*. SIGMOD '89. New York, NY, USA: Association for Computing Machinery, pp. 215–224. DOI: 10.1145/67544.66946.

Mcllroy, Doug (1964). *The Origin of Unix Pipes*. http://doc.cat-v.org/unix/pipes/. (Online: accessed 2021-02-03).

Mills, D.L. (1991). 'Internet Time Synchronization: The Network Time Protocol'. In: *IEEE Transactions on Communications* 39 (10), pp. 1482–1493. DOI: 10.1109/26.103043.

Morrison, J.P. (1994). *Flow-Based Programming: A New Approach to Application Development*. Computer Science. Van Nostrand Reinhold. ISBN: 978-0-442-01771-2.

Mouradian, C.; Naboulsi, D.; Yangui, S.; Glitho, R. H.; Morrow, M. J.; Polakos, P. A. (2018). 'A Comprehensive Survey on Fog Computing: State-of-the-Art and Research Challenges'. In: *IEEE Communications Surveys Tutorials* 20 (1), pp. 416–464. DOI: 10.1109/COMST.2017.2771153.

Mozzaquatro, Bruno A.; Jardim-Goncalves, Ricardo; Agostinho, Carlos (2017). 'Situation Awareness in the Internet of Things'. In: *2017 International Conference on Engineering, Technology and Innovation (ICE/ITMC)*, pp. 982–990. DOI: 10.1109/ICE.2017.8279988.

Mühl, Gero (2001). 'Generic Constraints for Content-Based Publish/Subscribe'. In: *Cooperative Information Systems*. Ed. by Carlo Batini; Fausto Giunchiglia; Paolo Giorgini; Massimo Mecella. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 211–225. DOI: 10.1007/3-540-44751-2_17.

Mühl, Gero; Fiege, Ludger; Pietzuch, Peter (2006). *Distributed Event-Based Systems*. Berlin Heidelberg: Springer-Verlag. ISBN: 978-3-540-32651-9. DOI: 10.1007/3-540-32653-7.

Noble, Brian D.; Satyanarayanan, M.; Narayanan, Dushyanth; Tilton, James Eric; Flinn, Jason; Walker, Kevin R. (1997). 'Agile Application-Aware Adaptation for Mobility'. In: *ACM SIGOPS Operating Systems Review* 31 (5), pp. 276–287. DOI: 10.1145/269005.266708.

Noor, Joseph; Tseng, Hsiao-Yun; Garcia, Luis; Srivastava, Mani (2019). 'DDFlow: Visualized Declarative Programming for Heterogeneous IoT Networks'. In: *Proceedings of the International Conference on Internet of Things Design and Implementation*. IoTDI '19. New York, NY, USA: Association for Computing Machinery, pp. 172–177. DOI: 10.1145/3302505.3310079.

Oki, Brian; Pfluegl, Manfred; Siegel, Alex; Skeen, Dale (1993). 'The Information Bus: An Architecture for Extensible Distributed Systems'. In: *ACM SIGOPS Operating Systems Review* 27 (5), pp. 58–68. DOI: 10.1145/173668.168624.

OpenStreetMap contributors (2021). *Planet Dump (Data File from 10 September 2021 of Database Geofabrik)*. Distributed under the Open Data Commons Open Database License (ODbL). Retrieved from https://planet.osm.org.

Pahl, C.; Lee, B. (2015). 'Containers and Clusters for Edge Cloud Architectures – A Technology Review'. In: *2015 3rd International Conference on Future Internet of Things and Cloud*, pp. 379–386. DOI: 10.1109/FiCloud.2015.35.

Pahl, Claus; Jamshidi, Pooyan; Zimmermann, Olaf (2020). *Microservices and Containers*. Gesellschaft für Informatik e.V. ISBN: 978-3-88579-694-7. DOI: 10.18420/SE2020_34.

Perrochon, Louis; Mann, Walter; Kasriel, Stephane; Luckham, David C. (1999). 'Event Mining with Event Processing Networks'. In: *Methodologies for Knowledge Discovery and Data Mining*. Ed. by Ning Zhong; Lizhu Zhou. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 474–478. DOI: 10.1007/3-540-48912-6_63.

Pietzuch, P. R.; Bacon, J. M. (2002). 'Hermes: A Distributed Event-Based Middleware Architecture'. In: *Proceedings 22nd International Conference on Distributed Computing Systems Workshops*, pp. 611–618. DOI: 10.1109/ICDCSW.2002.1030837.

Pinto, H Sofia; Martins, JP (2000). 'Reusing Ontologies'. In: *AAAI 2000 Spring Symposium on Bringing Knowledge to Business Processes*. Vol. 2. Karlsruhe, Germany: AAAI, p. 7.

Pivarski, Jim; Bennett, Collin; Grossman, Robert L. (2016). 'Deploying Analytics with the Portable Format for Analytics (PFA)'. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. New York, NY, USA: Association for Computing Machinery, pp. 579–588. DOI: 10.1145/2939672.2939731.

Pop, Paul; Zarrin, Bahram; Barzegaran, Mohammadreza; Schulte, Stefan; Punnekkat, Sasikumar; Ruh, Jan; Steiner, Wilfried (2021). 'The FORA Fog Computing Platform for Industrial IoT'. In: *Information Systems* 98, p. 101727. DOI: 10.1016/j.is.2021.101727.

Powell, David (1996). 'Group Communication'. In: *Communications of the ACM* 39 (4), pp. 50–53. DOI: 10.1145/227210.227225.

Preden, J. S.; Tammemäe, K.; Jantsch, A.; Leier, M.; Riid, A.; Calis, E. (2015). 'The Benefits of Self-Awareness and Attention in Fog and Mist Computing'. In: *Computer* 48 (7), pp. 37–45. DOI: 10.1109/MC.2015.207.

Puliafito, Carlo; Mingozzi, Enzo; Vallati, Carlo; Longo, Francesco; Merlino, Giovanni (2018). 'Companion Fog Computing: Supporting Things Mobility Through Container Migration at the Edge'. In: *2018 IEEE International Conference on Smart Computing (SMARTCOMP)*. Taormina: IEEE, pp. 97–105. DOI: 10.1109/SMARTCOMP.2018.00079.

Rausch, T.; Dustdar, S. (2019). 'Edge Intelligence: The Convergence of Humans, Things, and AI'. In: *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 86–96. DOI: 10.1109/IC2E.2019.00022.

Rausch, Thomas; Nastic, Stefan; Dustdar, Schahram (2018). 'EMMA: Distributed QoS-Aware MQTT Middleware for Edge Computing Applications'. In: *Proceedings - 2018 IEEE International Conference on Cloud Engineering, IC2E 2018*. IEEE, pp. 191–197. DOI: 10.1109/IC2E.2018.00043.

Ravindra, Pushkara; Khochare, Aakash; Reddy, Siva Prakash; Sharma, Sarthak; Varshney, Prateeksha; Simmhan, Yogesh (2017). '𝔼ℂℍ𝕆: An Adaptive Orchestration Platform for Hybrid Dataflows across Cloud and Edge'. In: *Service-Oriented Computing*. Ed. by Michael Maximilien; Antonio Vallecillo; Jianmin Wang; Marc Oriol. Lecture Notes in Computer Science. Cham: Springer International Publishing, pp. 395–410. DOI: 10.1007/978-3-319-69035-3_28.

Reclus, Fabrice; Drouard, Kristen (2009). 'Geofencing for Fleet & Freight Management'. In: *2009 9th International Conference on Intelligent Transport Systems Telecommunications, (ITST)*, pp. 353–356. DOI: 10.1109/ITST.2009.5399328.

Rehman, Muhammad Habib; Yaqoob, Ibrar; Salah, Khaled; Imran, Muhammad; Jayaraman, Prem Prakash; Perera, Charith (2019). 'The Role of Big Data Analytics in Industrial Internet of Things'. In: *Future Generation Computer Systems* 99, pp. 247–259. DOI: 10.1016/j.future.2019.04.020.

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

Riemer, Dominik (2016). 'Methods and Tools for Management of Distributed Event Processing Applications'. PhD thesis. Karlsruher Institut für Technologie (KIT). DOI: 10.5445/IR/1000070005.

Riemer, Dominik; Kaulfersch, Florian; Hutmacher, Robin; Stojanovic, Ljiljana (2015). 'StreamPipes: Solving the Challenge with Semantic Stream Processing Pipelines'. In: *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. DEBS '15. New York, NY, USA: Association for Computing Machinery, pp. 330–331. DOI: 10.1145/2675743.2776765.

Riemer, Dominik; Ljiljana, Stojanovic; Nenad, Stojanovic; Zehnder, Philipp; Wiener, Patrick (2019). 'Fog for Everyone: Modeling of Distributed Data Processing Pipelines for the Industrial Internet of Things'. In: *19th Annual IEEE/ACM International Symposium in Cluster, Cloud, and Grid Computing*. CCGrid. Larnaca, Cyprus.

Riemer, Dominik; Stojanovic, Ljiljana; Stojanovic, Nenad (2014). 'SEPP: Semantics-Based Management of Fast Data Streams'. In: *Proceedings - IEEE 7th International Conference on Service-Oriented Computing and Applications, SOCA 2014*. IEEE, pp. 113–118. DOI: 10.1109/SOCA.2014.52.

Rosenblum, David S.; Wolf, Alexander L. (1997). 'A Design Framework for Internet-Scale Event Observation and Notification'. In: *ACM SIGSOFT Software Engineering Notes* 22 (6), pp. 344–360. DOI: 10.1145/267896.267920.

Sajjad, H. P.; Danniswara, K.; Al-Shishtawy, A.; Vlassov, V. (2016). 'SpanEdge: Towards Unifying Stream Processing over Central and Near-the-Edge Data Centers'. In: *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, pp. 168–178. DOI: 10.1109/SEC.2016.17.

Santoro, Daniele; Zozin, Daniel; Pizzolli, Daniele; De Pellegrini, Francesco; Cretti, Silvio (2017). 'Foggy: A Platform for Workload Orchestration in a Fog Computing Environment'. In: *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 231–234. DOI: 10.1109/CloudCom.2017.62.

Satyanarayanan, M. (2001). 'Pervasive Computing: Vision and Challenges'. In: *IEEE Personal Communications* 8 (4), pp. 10–17. DOI: 10.1109/98.943998.

Satyanarayanan, M. (2017). 'The Emergence of Edge Computing'. In: *Computer* 50 (1), pp. 30–39. DOI: 10.1109/MC.2017.9.

Satyanarayanan, M.; Bahl, P.; Caceres, R.; Davies, N. (2009). 'The Case for VM-Based Cloudlets in Mobile Computing'. In: *IEEE Pervasive Computing* 8 (4), pp. 14–23. DOI: 10.1109/MPRV.2009.82.

Satyanarayanan, M.; Simoens, P.; Xiao, Y.; Pillai, P.; Chen, Z.; Ha, K.; Hu, W.; Amos, B. (2015). 'Edge Analytics in the Internet of Things'. In: *IEEE Pervasive Computing* 14 (2), pp. 24–31. DOI: 10.1109/MPRV.2015.32.

Saurez, Enrique; Hong, Kirak; Lillethun, Dave; Ramachandran, Umakishore; Ottenwälder, Beate (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*. New York, New York, USA: ACM Press, pp. 258–269. DOI: 10.1145/2933267.2933317.

Schiefer, Josef; Rozsnyai, Szabolcs; Rauscher, Christian; Saurer, Gerd (2007). 'Event-Driven Rules for Sensing and Responding to Business Situations'. In: *Proceedings of the 2007 Inaugural International Conference on Distributed Event-Based Systems*. DEBS '07. New York, NY, USA: Association for Computing Machinery, pp. 198–205. DOI: 10.1145/1266894.1266934.

Schilit, B.; Adams, N.; Want, R. (1994). 'Context-Aware Computing Applications'. In: *1994 First Workshop on Mobile Computing Systems and Applications*. IEEE, pp. 85–90. DOI: 10.1109/WMCSA.1994.16.

Schilling, Björn; Koldehofe, Boris; Pletat, Udo; Rothermel, Kurt (2010). 'Distributed Heterogeneous Event Processing: Enhancing Scalability and Interoperability of CEP in an Industrial Context'. In: *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*. DEBS '10. New York, NY, USA: Association for Computing Machinery, pp. 150–159. DOI: 10.1145/1827418.1827453.

Schwab, Klaus (2016). *The Fourth Industrial Revolution*. World Economic Forum.

Schwarzkopf, Malte; Konwinski, Andy; Abd-El-Malek, Michael; Wilkes, John (2013). 'Omega: Flexible, Scalable Schedulers for Large Compute Clusters'. In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. New York, NY, USA: Association for Computing Machinery, pp. 351–364. DOI: 10.1145/2465351.2465386.

Segall, Bill; Arnold, David (1997). 'Elvin Has Left the Building: A Publish/Subscribe Notification Service with Quenching'. In: *Proceedings of the 1997 Australian UNIX Users Group Technical Conference*, pp. 243–255.

Sharon, Guy; Etzion, Opher (2007). *Event Processing Network - A Conceptual Model*. Technion-Israel Institute of Technology, Faculty of Industrial and Management Engineering.

Shi, Cong; Lakafosis, Vasileios; Ammar, Mostafa H.; Zegura, Ellen W. (2012). 'Serendipity: Enabling Remote Computing among Intermittently Connected Mobile Devices'. In: *Proceedings of the Thirteenth ACM International Symposium on Mobile Ad Hoc Network-*

ing and Computing. MobiHoc '12. New York, NY, USA: Association for Computing Machinery, pp. 145–154. DOI: 10.1145/2248371.2248394.

Shi, Ke; Deng, Zhancheng; Qin, Xuan (2011). 'TinyMQ: A Content-Based Publish/Subscribe Middleware for Wireless Sensor Networks'. In: *SENSORCOMM 2011, The Fifth International Conference on Sensor Technologies and Applications*, pp. 12–17. ISBN: 978-1-61208-144-1.

Shi, W.; Cao, J.; Zhang, Q.; Li, Y.; Xu, L. (2016). 'Edge Computing: Vision and Challenges'. In: *IEEE Internet of Things Journal* 3 (5), pp. 637–646. DOI: 10.1109/JIOT.2016.2579198.

Shi, W.; Dustdar, S. (2016). 'The Promise of Edge Computing'. In: *Computer* 49 (5), pp. 78–81. DOI: 10.1109/MC.2016.145.

Shimrat, M. (1962). 'Algorithm 112: Position of Point Relative to Polygon'. In: *Communications of the ACM* 5 (8), p. 434. DOI: 10.1145/368637.368653.

Shin, K. G.; Chang, Y.- (1989). 'Load Sharing in Distributed Real-Time Systems with State-Change Broadcasts'. In: *IEEE Transactions on Computers* 38 (8), pp. 1124–1142. DOI: 10.1109/12.30867.

Skarlat, Olena; Karagiannis, Vasileios; Rausch, Thomas; Bachmann, Kevin; Schulte, Stefan (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, pp. 164–173. DOI: 10.1109/UCC.2018.00025.

Skarlat, Olena; Schulte, Stefan; Borkowski, Michael; Leitner, Philipp (2016). 'Resource Provisioning for IoT Services in the Fog'. In: *Proceedings - 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications, SOCA 2016*, pp. 32–39. DOI: 10.1109/SOCA.2016.10.

Steiner, Wilfried; Poledna, Stefan (2016). 'Fog Computing as Enabler for the Industrial Internet of Things'. In: *e & i Elektrotechnik und Informationstechnik* 133 (7), pp. 310–314. DOI: 10.1007/s00502-016-0438-2.

Stojmenovic, I.; Wen, S. (2014). 'The Fog Computing Paradigm: Scenarios and Security Issues'. In: *2014 Federated Conference on Computer Science and Information Systems*, pp. 1–8. DOI: 10.15439/2014F503.

Stonebraker, Michael; Çetintemel, Uğur; Zdonik, Stan (2005). 'The 8 Requirements of Real-Time Stream Processing'. In: *ACM SIGMOD Record* 34 (4), pp. 42–47. DOI: 10.1145/1107499.1107504.

Sullivan, Kevin; Notkin, David (1990). 'Reconciling Environment Integration and Component Independence'. In: *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*. SDE 4. New York, NY, USA: Association for Computing Machinery, pp. 22–33. DOI: 10.1145/99277.99281.

Sunyaev, Ali (2020). *Internet Computing: Principles of Distributed Systems and Emerging Internet-Based Technologies*. Springer International Publishing. ISBN: 978-3-030-34956-1. DOI: 10.1007/978-3-030-34957-8.

Taherizadeh, Salman; Stankovski, Vlado; Grobelnik, Marko (2018). 'A Capillary Computing Architecture for Dynamic Internet of Things: Orchestration of Microservices

from Edge Devices to Fog and Cloud Providers'. In: *Sensors (Switzerland)* 18 (9). DOI: 10.3390/s18092938.

Tai, Stefan; Nimis, Jens; Lenk, Alexander; Klems, Markus (2010). 'Cloud Service Engineering'. In: *2010 ACM/IEEE 32nd International Conference on Software Engineering*. Vol. 2, pp. 475–476. DOI: 10.1145/1810295.1810437.

Tammemäe, Kalle; Jantsch, Axel; Kuusik, Alar; Preden, Jürgo-Sören; Õunapuu, Enn (2018). 'Self-Aware Fog Computing in Private and Secure Spheres'. In: *Fog Computing in the Internet of Things: Intelligence at the Edge*. Ed. by Amir M. Rahmani; Pasi Liljeberg; Jürgo-Sören Preden; Axel Jantsch. Cham: Springer International Publishing, pp. 71–99. ISBN: 978-3-319-57639-8. DOI: 10.1007/978-3-319-57639-8_5.

Tapadinhas, Joao; Idoine, Carlie (2016). *Citizen Data Science Augments Data Discovery and Simplifies Data Science*. Gartner Research G00314599.

Terry, Douglas; Goldberg, David; Nichols, David; Oki, Brian (1992). 'Continuous Queries over Append-Only Databases'. In: *ACM SIGMOD Record* 21 (2), pp. 321–330. DOI: 10.1145/141484.130333.

Terzo, Olivier; Djemame, Karim; Scionti, Alberto; Pezuela, Clara (2019). *Heterogeneous Computing Architectures: Challenges and Vision*. CRC Press. ISBN: 978-0-429-68004-5.

Thornton, Chris; Hutter, Frank; Hoos, Holger H.; Leyton-Brown, Kevin (2013). 'Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms'. In: *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '13. New York, NY, USA: Association for Computing Machinery, pp. 847–855. DOI: 10.1145/2487575.2487629.

TIBCO (1999). *TIB/Rendezvous*. White Paper. TIBCO, Palo Alto, CA.

To, Quoc-Cuong; Soto, Juan; Markl, Volker (2018). 'A Survey of State Management in Big Data Processing Systems'. In: *The VLDB Journal — The International Journal on Very Large Data Bases* 27 (6), pp. 847–872. DOI: 10.1007/s00778-018-0514-9.

Tsagkaropoulos, Andreas; Verginadis, Yiannis; Compastié, Maxime; Apostolou, Dimitris; Mentzas, Gregoris (2021). 'Extending TOSCA for Edge and Fog Deployment Support'. In: *Electronics* 10 (6), p. 737. DOI: 10.3390/electronics10060737.

Tziouvaras, Athanasios; Foukalas, Fotis (2020). 'Edge AI for Industry 4.0: An Internet of Things Approach'. In: *24th Pan-Hellenic Conference on Informatics*. PCI 2020. New York, NY, USA: Association for Computing Machinery, pp. 121–126. DOI: 10.1145/3437120.3437289.

Vaquero, Luis M.; Rodero-Merino, Luis (2014). 'Finding Your Way in the Fog: Towards a Comprehensive Definition of Fog Computing'. In: *ACM SIGCOMM Computer Communication Review* 44 (5), pp. 27–32. DOI: 10.1145/2677046.2677052.

Varghese, Blesson; Buyya, Rajkumar (2018). 'Next Generation Cloud Computing: New Trends and Research Directions'. In: *Future Generation Computer Systems* 79, pp. 849–861. DOI: 10.1016/j.future.2017.09.020.

Varshney, Prateeksha; Simmhan, Yogesh (2017). 'Demystifying Fog Computing: Characterizing Architectures, Applications and Abstractions'. In: *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, pp. 115–124. DOI: 10.1109/ICFEC.2017.20.

Verginadis, Yiannis; Alshabani, Iyad; Mentzas, Gregoris; Stojanovic, Nenad (2017). 'PrEsto-Cloud: Proactive Cloud Resources Management at the Edge for Efficient Real-Time Big Data Processing'. In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science*. CLOSER 2017. Setubal, PRT: SCITEPRESS - Science and Technology Publications, Lda, pp. 611–617. DOI: 10.5220/0006359106110617.

Verma, Abhishek; Pedrosa, Luis; Korupolu, Madhukar; Oppenheimer, David; Tune, Eric; Wilkes, John (2015). 'Large-Scale Cluster Management at Google with Borg'. In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. New York, NY, USA: Association for Computing Machinery, pp. 1–17. DOI: 10.1145/2741948.2741964.

Villari, Massimo; Fazio, Maria; Dustdar, Schahram; Rana, Omer; Ranjan, Rajiv (2016). 'Osmotic Computing: A New Paradigm for Edge/Cloud Integration'. In: *IEEE Cloud Computing* 3 (6), pp. 76–83. DOI: 10.1109/MCC.2016.124.

Vögler, M.; Schleicher, J.; Inzinger, C.; Nastic, S.; Sehic, S.; Dustdar, S. (2015). 'LEONORE – Large-Scale Provisioning of Resource-Constrained IoT Deployments'. In: *2015 IEEE Symposium on Service-Oriented System Engineering*, pp. 78–87. DOI: 10.1109/SOSE.2015.23.

Wang, Shiqiang; Zafer, Murtaza; Leung, Kin K. (2017). 'Online Placement of Multi-Component Applications in Edge Computing Environments'. In: *IEEE Access* 5, pp. 2514–2533. DOI: 10.1109/ACCESS.2017.2665971.

Weisenburger, P.; Luthra, M.; Koldehofe, B.; Salvaneschi, G. (2017). 'Quality-Aware Runtime Adaptation in Complex Event Processing'. In: *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pp. 140–151. DOI: 10.1109/SEAMS.2017.10.

Wen, Zhenyu; Quoc, Do Le; Bhatotia, Pramod; Chen, Ruichuan; Lee, Myungjin (2018). 'Approximate Edge Analytics for the IoT Ecosystem'. In: arXiv: 1805.05674.

Wiener, Patrick (2018). 'Dynamic Management of Distributed Stream Processing Pipelines in Fog Computing Infrastructures'. In: *Doctoral Symposium of the 19th International Middleware Conference*. Middleware. Rennes, Britanny, France.

Wiener, Patrick; Simko, Viliam; Nimis, Jens (2017). 'Taming the Evolution of Big Data and Its Technologies in BigGIS - A Conceptual Architectural Framework for Spatio-Temporal Analytics at Scale'. In: *Proceedings of the 3rd International Conference on Geographical Information Systems Theory, Applications and Management*. Vol. 1. GISTAM. Porto, Portugal, pp. 90–101. DOI: 10.5220/0006334200900101.

Wiener, Patrick; Stein, Manuel; Seebacher, Daniel; Bruns, Julian; Frank, Matthias; Simko, Viliam; Zander, Stefan; Nimis, Jens (2016). 'BigGIS: A Continuous Refinement Approach to Master Heterogeneity and Uncertainty in Spatio-Temporal Big Data (Vision Paper)'. In: *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. SIGSPACIAL '16. New York, NY, USA: Association for Computing Machinery, pp. 1–4. DOI: 10.1145/2996913.2996931.

Wiener, Patrick; Zehnder, Philipp; Heyden, Marco; Philipp, Patrick; Riemer, Dominik (2020a). 'Fogsy: Towards Holistic Industrial AI Management in Fog and Edge Environments'. In: *KuVS-Fachgespräch Fog Computing*. KuVS. Wien, Essen, pp. 16–19. DOI: 10.34726/kuvs2020.

Wiener, Patrick; Zehnder, Philipp; Riemer, Dominik (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. Larnaca, Cyprus, pp. 1–6. DOI: 10.1109/CFEC.2019.8733145.

Wiener, Patrick; Zehnder, Philipp; Riemer, Dominik (2020b). 'Managing Geo-Distributed Stream Processing Pipelines for the IIoT with StreamPipes Edge Extensions'. In: *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*. DEBS. New York, NY, USA: Association for Computing Machinery, pp. 165–176. DOI: 10.1145/3401025.3401764.

Wöbker, Cecil; Seitz, Andreas; Mueller, Harald; Bruegge, Bernd (2018). 'Fogernetes: Deployment and Management of Fog Computing Applications'. In: *IEEE/IFIP Network Operations and Management Symposium: Cognitive Management in a Cyber World, NOMS 2018*. IEEE, pp. 1–7. DOI: 10.1109/NOMS.2018.8406321.

Wong, Jason; West, Mike; Howard, Chris; Driver, Mark (2015). *Citizen Development Is Fundamental to the Digital Workplace*. Gartner Research G00278137.

Wurster, Michael; Breitenbücher, Uwe; Falkenthal, Michael; Krieger, Christoph; Leymann, Frank; Saatkamp, Karoline; Soldani, Jacopo (2020). 'The Essential Deployment Metamodel: A Systematic Review of Deployment Automation Technologies'. In: *SICS Software-Intensive Cyber-Physical Systems* 35 (1), pp. 63–75. DOI: 10.1007/s00450-019-00412-x.

Yigitoglu, Emre; Mohamed, Mohamed; Liu, Ling; Ludwig, Heiko (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*, pp. 38–45. DOI: 10.1109/AIMS.2017.14.

Yousefpour, A.; Ishigaki, G.; Gour, R.; Jue, J. P. (2018). 'On Reducing IoT Service Delay via Fog Offloading'. In: *IEEE Internet of Things Journal* 5 (2), pp. 998–1010. DOI: 10.1109/JIOT.2017.2788802.

Yousefpour, Ashkan; Fung, Caleb; Nguyen, Tam; Kadiyala, Krishna; Jalali, Fatemeh; Niakanlahiji, Amirreza; Kong, Jian; Jue, Jason P. (2019). 'All One Needs to Know about Fog Computing and Related Edge Computing Paradigms: A Complete Survey'. In: *Journal of Systems Architecture* 98, pp. 289–330. DOI: 10.1016/j.sysarc.2019.02.009.

Zehnder, Philipp; Wiener, Patrick; Riemer, Dominik (2019). 'Using Virtual Events for Edge-Based Data Stream Reduction in Distributed Publish/Subscribe Systems'. In: *2019 IEEE 3rd International Conference on Fog and Edge Computing (ICFEC)*. ICFEC '19. Larnaca, Cyprus, pp. 1–10. DOI: 10.1109/CFEC.2019.8733146.

Zehnder, Philipp; Wiener, Patrick; Straub, Tim; Riemer, Dominik (2020). 'StreamPipes Connect: Semantics-Based Edge Adapters for the IIoT'. In: *The Semantic Web*. Ed. by Andreas Harth; Sabrina Kirrane; Axel-Cyrille Ngonga Ngomo; Heiko Paulheim; Anisa Rula; Anna Lisa Gentile; Peter Haase; Michael Cochez. Lecture Notes in Computer Science. Cham: Springer International Publishing, pp. 665–680. DOI: 10.1007/978-3-030-49461-2_39.

Zhou, Jun; Cao, Zhenfu; Dong, Xiaolei; Vasilakos, Athanasios V. (2017). 'Security and Privacy for Cloud-Based IoT: Challenges'. In: *IEEE Communications Magazine* 55 (1), pp. 26–33. DOI: 10.1109/MCOM.2017.1600363CM.

Zhou, Z.; Chen, X.; Li, E.; Zeng, L.; Luo, K.; Zhang, J. (2019). 'Edge Intelligence: Paving the Last Mile of Artificial Intelligence With Edge Computing'. In: *Proceedings of the IEEE* 107 (8), pp. 1738–1762. DOI: 10.1109/JPROC.2019.2918951.

Zowghi, Didar; Coulin, Chad (2005). 'Requirements Elicitation: A Survey of Techniques, Approaches, and Tools'. In: *Engineering and Managing Software Requirements*. Ed. by Aybüke Aurum; Claes Wohlin. Berlin, Heidelberg: Springer, pp. 19–46. ISBN: 978-3-540-28244-0. DOI: 10.1007/3-540-28244-0_2.