FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

An Architectural Style for Fog Computing: Formalization and Application

Andreas Horst Nikolaus Seitz

FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Forschungs- und Lehreinheit 1
Angewandte Softwaretechnik

An Architectural Style for Fog Computing: Formalization and Application

Andreas Horst Nikolaus Seitz

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Prof. Dr.-Ing. Jörg Ott
Prüfer der Dissertation:	1. Prof. Bernd Brügge, Ph.D.
	2. Prof. Dr. Dirk Riehle

Die Dissertation wurde am 17.01.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 10.03.2019 angenommen.

Abstract

The simultaneous realization of non-functional requirements such as real-time access, synchronization, scalability, and availability represents a challenge in existing software architectures. In particular, current architectures do not allow the easy formulation of trade-offs that include real-time access and synchronization. This dissertation presents the Fogxy architectural style to address these problems. It allows to address these challenges during the different phases of a software engineering lifecycle from analysis to build- and release-management. Fogxy is based on the Fog Meta Model to identify objects and packages that are critical for the realization of non-functional requirements. Fogxy provides abstractions for a variety of application domains, such as manufacturing, digital health, and smart environments. To evaluate Fogxy, we introduce a method called Review for Intermediate Architectural Patterns (RIAP). To validate the Fog Meta Model and Fogxy we applied them in two industrial Fog Computing applications. In each of these applications, software engineers were able to simultaneously realize real-time access and synchronization by using the formalizations based on the Fog Meta Model and Fogxy. With Seamless Computing, we present a build- and release-management concept that enables the homogeneous distribution of Fogxy components to heterogeneous hardware nodes.

Zusammenfassung

Nicht-funktionale Anforderungen wie Echtzeitzugriff, Synchronisation, Skalierbarkeit und Verfügbarkeit stellen eine Herausforderung für bestehende Softwarearchitekturen dar. Insbesondere erlauben diese Architekturen nicht die gleichzeitige Realisierung von Echtzeitzugriff und Synchronisation. Diese Dissertation präsentiert den Fogxy Architekturstil, der diese Herausforderungen in den verschiedenen Phasen der Software Entwicklung von der Analyse bis zum Build- und Release-Management bewältigt. Fogxy basiert auf dem Fog Meta Model, um Objekte und Pakete zu identifizieren, die für die Realisierung nichtfunktionaler Anforderungen entscheidend sind. Fogxy bietet Abstraktionen für eine Vielzahl von Anwendungsbereichen wie industrielle Produktion, Digitale Gesundheit und Intelligente Umgebungen. Zur Evaluierung von Fogxy führen wir eine neue Methode RIAP (Review for Intermediate Architectural Patterns) ein. Das Fog Meta Model und der Fogxy Architekturstil wurden in zwei industriellen Fog Computing-Anwendungen eingesetzt. In jeder dieser Anwendungen konnten die Entwickler gleichzeitig Echtzeitzugriff und Synchronisation mit Hilfe des Fog Meta Models und Fogxy realisieren. Mit Seamless Computing präsentieren wir ein Build- und Release-Management Konzept, das die homogene Verteilung von Fogxy-Komponenten auf heterogene Hardwareknoten ermöglicht.

Acknowledgments

This dissertation would not have been possible without the inspiration, ideas, encouragement, and support of many people. I would like to thank all the people that accompanied me on my journey.

First of all, thank you, Bernd Brügge. You taught me a lot in your inimitable way; you continuously inspired and motivated me. Many thanks for the opportunities and the great time, which I will certainly never forget. Thank you also to Dirk Riehle as second advisor of this dissertation.

Furthermore, I would like to thank my colleagues and co-authors for the collaboration as well as the exciting and inspiring discussions on the most diverse topics. Thank you very much for your support and the time I was allowed to spend with you. Many thanks especially to Constantin Scheuermann, Stephan Krusche, Jan Ole Johanßen, Lukas Alperowitz, Dora Dzvonyar, Dominic Henze, and Nadine von Frankenberg. Not to forget all the other members of the chair who made the experience so wonderful and exciting. Thanks to all the students whose final theses I advised or worked within the iPraktikum and lectures.

My friends played a crucial role in this dissertation. The combination of research, teaching, leisure time, and time spent with friends made life enjoyable. Unfortunately, I cannot name all of them here personally, but I am grateful to each of you for your trust, support and our time together.

I would like to express my deep gratitude and love for my family. My sincere thanks to my parents Reinhard and Evelyn and my sister Lisa. You are always there for me and you support me in all my projects. Thank you also for your patience, which I try from time to time. Without you, none of this would have happened.

Finally and most importantly, thank you, Sandra. The past months have certainly not been easy, but you always encouraged and motivated me. I am profoundly grateful for your support, your trust, and your love.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

Fog Computing is a new architectural style based on Cloud Computing. Cloud Computing as a replacement for the client-server architectural style has established itself as a computing paradigm to synchronize data among multiple clients. However, it is not appropriate for clients with real-time constraints, in particular for low latency requirements. Fog Computing, also known as Edge Computing or Cloudlets, proposes to simultaneously address these requirements—synchronization of data, real-time access, and availability—and enables the realization of design goals previously considered trade-offs.

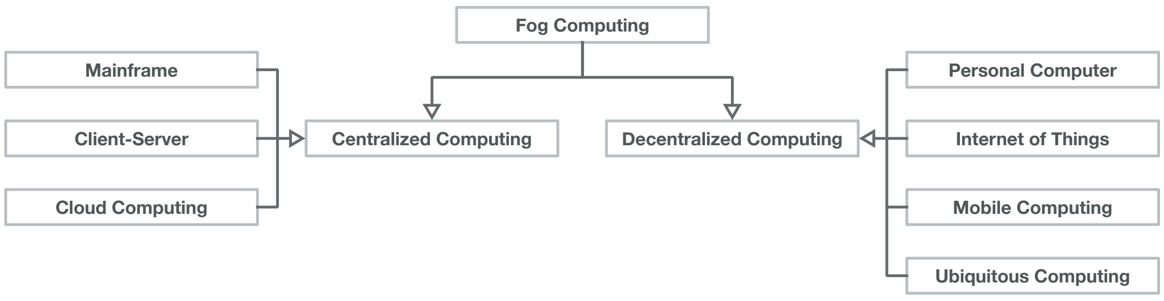

Figure 1.1: Fog Computing multiple inheritance

The pendulum effect between decentralization and centralization has already occurred several times in the history of software architectures. In the early 1950s, the mainframe computer was state-of-the-art. Machines were used to perform mathematical calculations "that were mostly run by experts behind closed doors" [WB97]. With the establishment of the personal computer (PC) and the client-server architectural style in the 1960s and 1970s, the mainframe was replaced and the shift from centralization to decentralization occurred. The invention of the Internet combined elements of the PC and the mainframe, linking millions of people and information, a paradigm that Weiser describes as "client-server computing on a massive scale" [WB97]. The penetration and versatility of this approach led to the development of Cloud Computing in the 1990s and, in turn, once more a centralized approach. During the 1990s, Weiser formulated his vision of Ubiquitous Computing. The concept has been integrated into the paradigms of Mobile Computing, Internet of Things (IoT), and Cyber-Physical Systems (CPSs) [Wei99]. The penetration of applications, an increasing number of devices, and the amount of data generated pose challenges for existing software architectures and in particular for Cloud Computing. “Purely centralized environments are ill-suited for applications that have soft and hard real-time requirements” [WYG$^+$17]. Fog Computing eases the decision-making between decentralized and centralized architectures because it inherits features from both (c.f. Figure 1.1). It combines centralized Cloud Computing with decentralized IoT to eliminate cloud constraints and to meet real-time access and synchronization requirements [BMZA12, SBCD09, Sat17a].

The following forces and reasons lead to the establishment of Fog Computing:

Low Latency and Real-Time Requirements: IoT applications have real-time requirements and require low and predictable latency to make decisions and perform actuations [DD17]. Unpredictable delays ruin the user experience of delay-sensitive applications [HNYL17].

Standardization, Heterogeneity, and Interoperability: For Fog Computing no standardization exists yet. The hardware ranges from embedded devices without operating systems to virtualized solutions in the cloud.

Resource Limitation: Devices in IoT have special purposes and are limited in memory, computing power, network connectivity, and battery capacity [HNYL17].

Geographical Distribution: The cloud is located in central data centers, while the nodes in Fog Computing are decentralized, allowing applications to be deployed remotely with unreliable Internet connection. Orchestration of the distributed components is required to enable their collaboration.

Missing and Inadequate Infrastructure: The requirements for network speed and availability are constantly increasing. Data is being collected and processed, and the number of requests being transmitted poses a challenge to the infrastructure. This results in high network loads and possible restrictions due to insufficient or unavailable infrastructure.

Interplay with the Cloud: Cloud components are dynamically scalable, while embedded devices meet real-time requirements. The combination of both approaches eliminates their drawbacks.

Access Control: Large numbers of Smart Objects[1] give rise to the challenge of controlling access to cloud components. Data can be potentially harmful when transferred to the cloud and can lead to security problems. An authentication and authorization solution is required to control Smart Objects' access.

Location Awareness and Mobility: The multitude of devices and their mobility pose a challenge. Devices are used and set up at the edge of the network for IoT scenarios. For orchestration, task distribution, and service provisioning, we must verify which devices are located in a given environment and track the ones entering or leaving it.

1.1 Problem

Yacoub states that the "most difficult part of building software is not coding; it is the decisions you make early at the design level. Those design decisions live with the system for the rest of its lifetime" [YA03]. Typically, these decisions are made at development time, making subsequent changes to the chosen architecture expensive or impossible. The goal is to postpone the decision between the architecture to be able to dynamically adapt the architecture to changing non-functional requirements and quality attributes. We therefore hypothesize that the Fogxy architectural style minimizes trade-offs between centralized and decentralized computing. For example, the analysis of sensor data must be done in real-time while historical data must be stored centrally and synchronized with all interested parties. We describe the Fog Meta Model and Seamless Computing formalizations that enable architecture decisions to be made at runtime. Software engineers no longer need to choose between a centralized or decentralized architecture. Depending on the context, application components can be dynamically allocated to fulfill the functionality within the required quality criteria.

Furthermore, there is "no common picture on what Fog Computing and a fog node, as its main building block, really is" [MMA+16]. Fog Computing, its architecture, and its impact on a software engineering process are still in its infancy. Non-functional requirements affecting the design, implementation, distribution, and integration of distributed systems on heterogeneous hardware nodes must be considered. Fog Computing requires engineered systems, definitions, and formalizations to achieve the promised benefits. The convergence of Cloud Computing and IoT leads to challenges and the need for methodologies, tools, and models to effectively design, develop, and deploy Fog Computing applications.

[1]Smart Objects are defined in Chapter 4.

1.2 Research Objectives

The primary objective is to validate the hypothesis we set out in the previous section. By applying the Fogxy architectural style, previously exclusive requirements can be fulfilled simultaneously. The decision for or against a decentralized architecture can be made at runtime. To take advantage of both decentralized and centralized paradigms, it is not sufficient to use the style alone. Fogxy affects all phases of a software engineering lifecycle and there is a need for further methods. Through the practical implementation of Fog Computing applications, we derive formalizations for the phases. Chen et al. highlight challenges regarding programming abstracts and models, fog architecture, and resource provisioning and management [CZS17]. We present a tailored lifecycle, namely Pattern-Based Development (PBD), which builds on the intensive use of patterns (cf. Section 2.3). For the phases analysis, system design, object design, and build- and release-management, we derive three formalizations. Figure 1.2 shows the allocation of the formalizations to the phases, as well as to the chapters.

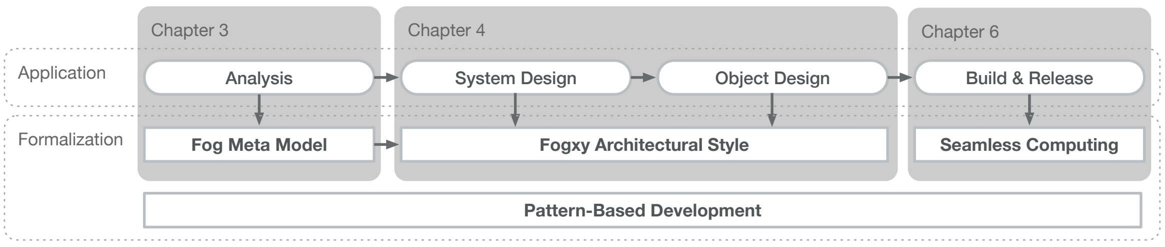

Figure 1.2: Breakdown of formalizations by chapter

1.3 Outline

This dissertation is structured as follows:

Chapter 2 introduces relevant foundations. We discuss the history of Fog Computing and the history of software architecture and patterns in software engineering. The terminology is clarified, and we illustrate the relation between the terms IoT, CPS, Industrial Internet of Things (IIoT), and Industry 4.0. We present similarities and differences regarding the terms Fog Computing, Edge Computing, Cloudlets, and related concepts, and classify them under the generalization Fog and Edge Computing. We discuss the Pattern-Based Development (PBD) lifecycle and delimit it from Pattern-Oriented Analysis and Design (POAD) and Pattern-Based Engineering (PBE).

Chapter 3 describes the Fog Meta Model as an extension of the UML metamodel by three stereotypes. The stereotypes allow the analysis of requirements and characteristics of Fog Computing applications as early as the analysis phase. We present objectives and motivation, and describe the Fog Meta Model in detail. We demonstrate the applicability of the Fog Meta Model in application examples, which we carried out in the context of our research.

Chapter 4 presents the Fogxy architectural style using a schema that includes the problem, context, and solution. In the solution, we present the style with the 4+1 view model for software architectures [Kru95]. By applying the Fogxy architectural style in practice, we prove its applicability to Fog Computing applications in different domains.

Chapter 5 describes the Review of Intermediate Architectural Patterns (RIAP) method as part of the Architectural Pattern Evaluation Process (APEP). We apply RIAP to evaluate the Fogxy architectural style. Through the practical application of the methodologies, we were able to evaluate and formatively improve the Fogxy architectural style and gain insights into the processes that we present in this chapter. RIAP uses a scenario-based approach in which an architectural pattern for a specific scenario is instantiated as a software architecture to check whether the non-functional requirements derived from the scenario can be met.

Chapter 6 deals with the build- and release-management of Fog Computing applications. The multitude of components and their distribution poses integration and deployment challenges. With Seamless Computing, we describe a concept in which a homogeneous development environment is established to enable the static and dynamic assignment of components to different layers. With Fogernetes and DYSCO, we present two implementations of Seamless Computing.

Chapter 7 presents the two case studies AIIoT and IIoT Bazaar. They validate the Fog Meta Model, Fogxy architectural style, and Seamless Computing methods and formalizations that have been presented. As application domains, we focus on two applications from manufacturing in which we demonstrate the simultaneous fulfillment of previously exclusive requirements. The chapter provides an overview of the other case studies presented in the form of publications.

Chapter 8 concludes the dissertation, summarizes the contributions, and provides an outlook on future work.

The dissertation is based on published journal, conference, and workshop papers: [SB18], [STB18b], [STB17], [MGSB17], [WSMB18], [GMP+18], [MGH+16], [KMS+18], [SHS+18], [SBB18], [SJB+17], [SHM+18], [HSHB18], [KMS+18], [STB18a], and [ASB19].

Moreover, the following bachelor and master theses were supervised in the context of this research: [Wan16], [Woe17], [Bec17], [Roh17], [Thi17], [Sye17], [Buc17], [Gaß17], [Hel18], [Kra18], [Kat18], and [Bod18].

1.4 Research Approach

We applied a formative approach to extend and improve the three formalizations, namely Fog Meta Model, Fogxy architectural style, and Seamless Computing. The starting point of each investigation was always an application example with a problem statement. This problem was brought to us by cooperation partners and partly constructed for the application examples. The application examples applied agile development methods and an adapted software lifecycle from analysis to build- and release-management [KABW14]. With PBD we focus on a lifecycle that forces the intensive use of patterns in each of these phases.

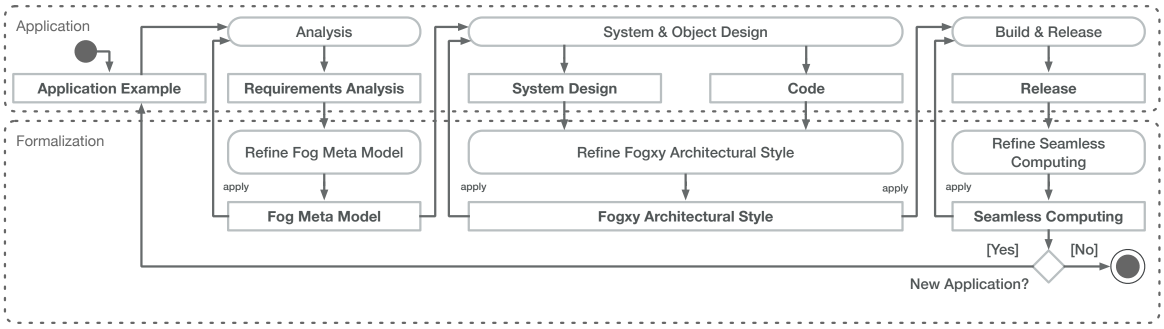

Figure 1.3: Formative and iterative research approach

From the practical application and implementation of the application examples, we derive formalizations for the respective phases. Figure 1.3 depicts the research process. We derive the Fog Meta Model for the analysis phase, the Fogxy architectural style for the system design and object design, and Seamless Computing for the build- and release-management phase. All problems refer to Fog Computing applications. After completion of each application example, we initiated a new project and applied the adapted formalizations, leading to an iterative improvement and extension of the formalizations. The analysis of and reflection on the implementation and application of formalizations allows them to be improved continuously.

Figure 1.4 shows the allocation of application examples to the domains. The examples, which we discuss in detail in the following chapters, are divided into the three application domains *Manufacturing*, *Smart Environments*, and *Digital Health*. According to [CZS17], these domains benefit from the application of Fog Computing.

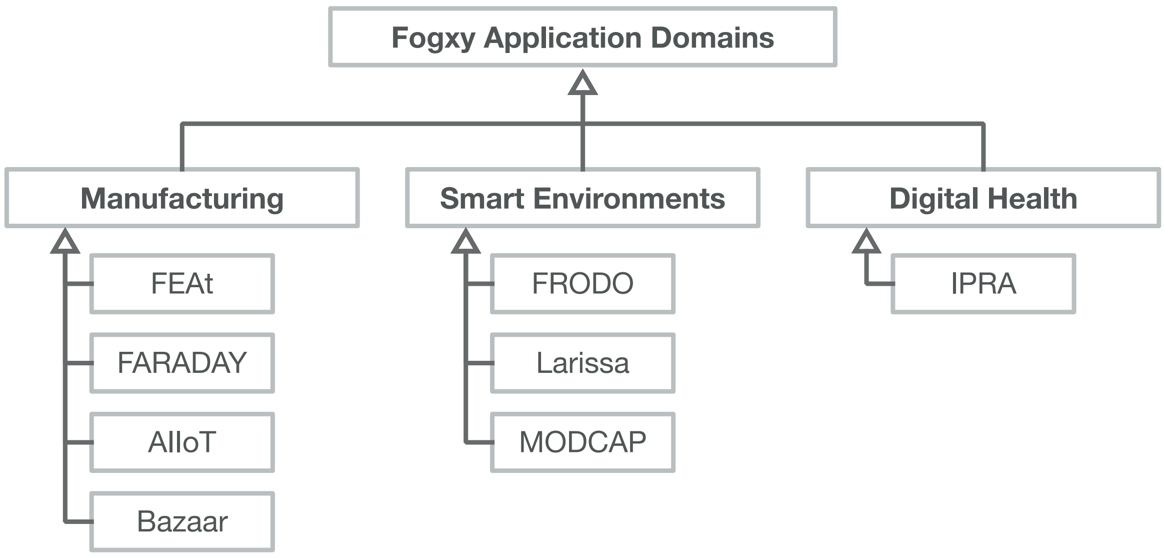

Figure 1.4: Allocation of application examples to domains

Chapter 2
Foundations

This chapter addresses relevant technologies, concepts and their definitions. In Section 2.1, we discuss the term IoT and separate it from CPS and IIoT. Section 2.2 considers software architectures, their history, quality characteristics, and evaluation methods. With Pattern-Based Development in Section 2.3, we present the applied lifecycle. Section 2.4 introduces the concept of Fog Computing, distinguishes related terms and shows both differences and similarities.

2.1 Internet of Things (IoT)

A variety of technologies, such as embedded systems, wireless sensor networks, and distributed computing influenced IoT. Technological advances and the establishment of the Internet made physical devices and everyday objects addressable. The connected sensors and actuators can be monitored and controlled remotely. The IoT makes the Ubiquitous Computing vision a reality and enables a multitude of new applications in a wide variety of domains. IoT is defined as a worldwide network of objects that are interconnected and uniquely addressable [GBMP13]. By enabling objects to communicate via intelligent interfaces, they become active participants in business, information, and social processes [SGFW10, SL08]. They interact with each other and with the environment by exchanging information and reacting to events with or without human intervention. The objective of IoT is to minimize the gap between the physical and the virtual world [FM05]. Especially in the consumer market, this trend has gained momentum in the past years with the ubiquitous presence of IoT devices in our everyday lives.

Further relevant terms were established in industry and research around IoT. We define a taxonomy with the terms *Internet of Things*, *Industrial Internet of Things*, *Cyber-Physical System*, *Industry 4.0*, *Industrial Internet*, and *Smart Objects* in Figure 2.1. Researchers have not reached consensus over the relation of IoT and CPS. Organizations and researchers use the term synonymously or see one as a subclass of the other [ISO14, Sto14, SSZ15]. Nunes et al. describe the emergence of differ-

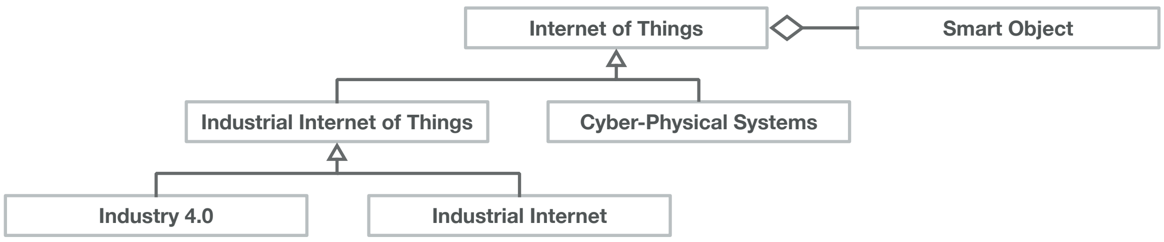

Figure 2.1: Internet of Things taxonomy

ent concepts driven by different groups: IoT was initially driven by the computer science community and more commonly used in Europe. CPSs were brought up by the engineering perspective, being supported by the US National Science Foundation [NZS15].

Industrial Internet of Things (IIoT)

In parallel to the *IoT*, the *Industrial Internet of Things (IIoT)* emerged, a paradigm that "brings together the advances of two transformative revolutions: the myriad machines, facilities, fleets and networks that arose from the Industrial Revolution, and the more recent powerful advances in computing, information and communication systems" [EA12]. The objective of IIoT is the adoption of IoT within the industrial domain to increase flexibility and productivity while reducing production cost. IIoT and IoT are characterized by a large number of smart objects [SW18] that are interconnected via a network and interact and coordinate to achieve a goal. We consider IIoT as a subclass of IoT (cf. Figure 2.1). The application of IoT in the industrial domain can have a great impact, but also faces challenges. IIoT, in contrast to IoT, represents stringent requirements regarding availability, security, and real-time requirements. In IoT, it is not decisive whether a device reacts or turns on with a delay. In industrial plants however, this can have devastating consequences. Special non-functional requirements and legacy systems pose challenges to the application of IIoT. Industrial processes, their organizations, and systems converge with advanced computing, analytics, low-cost sensing, and new levels of connectivity permitted by the Internet.

Industry 4.0 and *Industrial Internet* are subclasses of IIoT (cf. Figure 2.1) that apply proposed technologies and concepts. The ability to impact production lines and consequently business performance in real-time is one of the key promises of Industrial Internet. The integration of data acquisition by sensors, data processing by computers, and direct physical actions by actuators facilitates this [EA12]. The Industrial Internet is "a technological enabler of significant advances in the efficiency of industrial processes" [GLM+15]. Industry 4.0[1] is a primarily German initiative that is used as a generic term for a project to digitize industrial production. The aim is to prepare and digitize industrial production for the future by using IoT. The term was introduced at the Hannover Fair 2011 and describes the application of IIoT in the production environment [Ste16]. For both terms, the use of IIoT promises competitive advantages. Efficiency and productivity can be increased by analyzing measured data and applying the knowledge gained. Fog Computing offers promising solutions for the challenges of processing large amounts of data, security aspects, and real-time access.

Cyber-Physical Systems (CPS)

CPSs emerged from embedded systems, which are technical systems that use sensors and actuators to interact with their environment for a specific purpose [Lee08]. Embedded systems equipped with network capabilities result in CPSs [Bro10, RLSS10, Lee08]. CPSs include sensors and actuators the way embedded systems do. CPSs take "sensor data (measuring properties of the physical world) from a variety of sources, transform it into information in the cyber world, process it, understand it and then transform it into appropriate actions in the physical world" [SSZ15]. CPSs close the gap between physical objects in the real world and their digital representation [Sch17]. CPSs enable the control logic of hardware devices to be changed at runtime and allow computation-intensive tasks to be outsourced to remote systems. CPSs are therefore suited for integration into Fog Computing applications.

2.2 Software Architecture and Patterns

In 1972, Parnas proposed the idea of modularization and information hiding to decompose systems on a high level to improve flexibility and understandability of a software system [Par72]. The software architecture of a system is the set of structures needed to reason about the system, which comprises software elements, their interrelations, and the properties of both [BCK12]. As an architecture defines a system in terms of computational components and interactions among those components [SG96], this is particularly important for Fog Computing applications. In the following, Section 2.2.1 deals with the terminology related to software architecture. Section 2.2.2 defines software quality attributes that are evaluated with the methods presented in Section 2.2.3.

[1] `https://www.bmbf.de/de/zukunftsprojekt-industrie-4-0-848.html`

2.2.1 Terminology

Figure 2.2 shows the relationship between *Reference Model*, *Reference Architecture*, *Software Architecture*, and *Architectural Patterns*. A reference model is related to application domains, while an architectural pattern is part of the solution domain. Together they form a reference architecture for a specific domain. A concrete software architecture represents the instantiation of a reference architecture. Reference models, architectural patterns and reference architectures are not yet architectures. They are useful concepts that capture elements of an architecture.

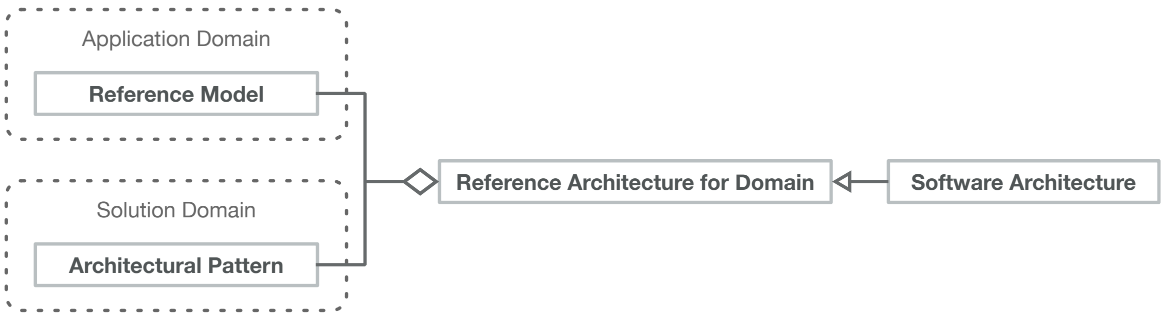

Figure 2.2: Relationship of software architecture key concepts (adapted from [BCK98])

Reference Model. A reference model is a decomposition of a known problem into parts that solve the problem together. It allows a quick overview of the problem space and the discussion of it. Bass defines them as "a division of functionality together with data flow between the pieces" [BCK12]. They enable the development of specific reference architectures and concrete software architectures. A reference model consists of unifying concepts, axioms, and relationships within an application domain, and is independent of technology or implementation [Sta06].

Reference Architecture. A reference architecture is a standardized, generic software architecture that is valid for a particular domain. It is composed of architectural patterns and a reference model and is also considered itself to be a pattern [SFI16]. Whereas a reference model divides the functionality, a reference architecture is the mapping of that functionality onto a system decomposition [BCK12]. A reference architecture allows the design of applications before implementation and creates a shared mental model for the architecture.

Software Architecture. A software architecture is an instantiation of a reference architecture. It describes the structure of the system, the relationship between its components and their externally visible properties [BCK12]. Fundamental structural decisions for a system are made in a software architecture. Changes to these after implementation are costly. Every element of an architecture should have a well-defined interface that encapsulates or hides changeable aspects, such as implementation.

Architectural Patterns and Styles. Patterns in software engineering have a nearly 30-year history. In 1991, Erich Gamma published his doctoral thesis on patterns for GUIs. In 1992 and 1993, the Gang of Four[2] participated in the Towards an Architecture Handbook workshop at OOPSLA. They published their first joint paper at the ECOOP conference and the standard pattern literature—Design Patterns: Elements of Reusable Object-Oriented Software. In 1993, the first PLoP (Pattern Languages of Programs) conference took place in Monticello, Illinois, USA. Between 1996 and 2007, the authors affiliated with Frank Buschmann published five volumes of the book Pattern-Oriented Software Architecture, which describes a multitude of architectural patterns.

A pattern is defined as "a solution to a recurring problem in a given context" [AIS77]. The term pattern was established in 1977 by Christopher Alexander, a building architect. Pattern techniques were established as being valuable architectural design techniques in the area of building architectures. This concept was adopted for software engineering and has also established itself. Since the publication of [GHJV95], many articles and books have been published. Patterns are described as simple and elegant solutions to specific problems in object-oriented software design, or as Fowler says, patterns are "an idea that has been useful in one practical context and will probably be useful in others" [Fow97]. Initially, the focus was on design patterns, which has expanded over the years to architectural patterns. Perry and Wolf introduced architectural styles in 1992: "If architecture is a formal arrangement of architectural elements, then architectural style is that which abstracts elements and formal aspects from various specific architectures" [PW92]. An architectural style is a recurring pattern that can be extracted from concrete architectures and expresses fundamental structural organization schemas for software systems [BMR+96, HA10].

While Bass et al. use the terms architectural pattern and architectural style interchangeably [BCK12], Buschmann et al. mention differences between architectural styles and patterns [BMR+96]. An architectural style describes the overall structure of an application, whereas a pattern may be found at different scales, ranging from patterns defining the overall structure of the application to patterns giving solutions to smaller design issues. Furthermore, an architectural style exists by itself as it is not dependent on other styles. A pattern depends on smaller contained patterns and other patterns that it interacts with [Ale79].

[2]Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides

Different structures exist to describe a pattern: the Alexandrian Form [AIS77], the Gang of Four Scheme [GHJV95] for design patterns and the Gang of Five Scheme [BMR+96] for architectural patterns.

2.2.2 Software Quality Attributes

A software architecture must meet quality criteria such as usability, performance, or modifiability [LPR03]. To address corresponding quality attributes, a software architect applies architectural patterns. There are different models for quality attributes [KRM16], such as McCall [MRW77], Boehm's quality model [BBL76], the FURPS quality model [Gra92], or ISO 9126 [ISO00]. The quality models include different quality features. Maintainability and reliability, for instance, are equally considered in all models, while only FURPS or McCall mention performance.

2.2.3 Software Evaluation Methods

Several ways to evaluate software architectures exist. Abowd et al. divide them into two categories: qualitative questions and quantitative measurements [ABC+97]. Qualitative questions use scenarios and checklists, while quantitative measurements use metrics or simulations and experiments to assess the quality of a system's architecture. Software Architecture Analysis Method (SAAM) [KBAW94] and Architecture Trade-Off Analysis Method (ATAM) [KKC00] are scenario-based evaluation methods. [BZJ04] and [BBM13] compare these methods. SAAM and ATAM target fully specified architectures that develop throughout a project. Clements introduced Active Reviews for Intermediate Designs (ARID) as a more lightweight method that concentrates on the viability of the software architecture [Cle00]. ARID is applied at the beginning of a design process to discover errors or inconsistencies. Different stakeholders are more likely to accept the design if, as with ARID, they are involved in advance. ARID is a combination of Active Design Reviews (ADR) [PW85] and ATAM. It is a method that actively challenges reviewers to solve review tasks using the design in relevant scenarios instead of asking questions [CKK02]. The general outline of ARID is based on the structure of ATAM and is divided into two phases: rehearsal and review. In the rehearsal phase, the lead designer and a facilitator meet up to create the exercises for the review. This phase is comprised of the following four steps:

1. **Identify the reviewers:** As ARID evaluates an intermediate architecture, the reviewers best suited are software engineers that are expected to apply the architecture. A group of around a dozen software engineers is selected.

2. **Prepare the design briefing:** In this step, the lead designer prepares the presentation and the facilitator reviews it. During the review, the facilitator asks questions for which the designer can prepare. The review improves the presentation and the designer practices to keep it within the given time frame. The presentation should not exceed two hours.

3. **Prepare the seed scenarios:** The lead designer and facilitator craft scenarios in which the design is applicable. These need not to be used for later evaluation but serve as a starting point and make the design more understandable.

4. **Prepare the materials:** Finally, the rehearsal concludes by preparing all the necessary materials for the review, such as copies of the presentation, scenarios, and agenda.

In the review phase, the stakeholders conduct the review by following these five steps:

5. **Present ARID:** The facilitator starts the review by presenting ARID and the steps involved.

6. **Present the design:** The lead designer presents the design and introduces the scenarios. Meanwhile, the audience can ask clarifying questions. Questions regarding the rationale or suggestions are not allowed. A minute taker notes down all questions that indicate a lack of clarity in the design, documentation, or presentation.

7. **Brainstorm and prioritize scenarios:** The group brainstorms scenarios that are suitable for the use of the design. Participants vote for scenarios that will be used for review.

8. **Apply the scenarios:** The group develops real or pseudo code using the design to tackle the problem presented in the scenarios. The group starts with the scenario that received the most votes. The designer is not allowed to help the group or give hints. If the group is stuck, the facilitator intervenes and the lead designer steers the group in the right direction. Every time this happens, an issue is recorded indicating a flaw in the design. The reviewers are also asked to bring up any discrepancies they reveal to be issues. This step continues until time is up or the top-rated scenarios are solved, or the group concludes that the design is suitable or unsuitable.

9. **Summarize:** Finally, the facilitator goes through all the issues identified, thanks the reviewers for participating, and asks for feedback regarding the review.

The result of ARID, SAAM, and ATAM is a list of issues. This list contains issues that the reviewers have uncovered in the architecture, for example that particular quality features cannot be met or that the design is not feasible. These issues help the designer to improve the architecture. APEP (cf. Chapter 5) applies the concept of the issue list.

2.3 Pattern-Based Development (PBD)

For the implementation of the applications examples, we used the Pattern-Based Development software lifecycle [SB18], which relies on the use of patterns as established software engineering knowledge throughout a software engineering lifecycle. From with the analysis phase, through system and object design to build- and release-management, PBD makes heavy use of patterns. The objective of PBD is the utilization and assignment of patterns starting with requirements elicitation. We define PBD as follows:

Pattern-Based Development is a model-based development approach that focuses on the extensive use of patterns throughout the software lifecycle.

In typical software lifecycles, patterns play an essential role during system and object design. PBD includes the application of patterns in all phases of a process, starting as early as the analysis phase. Figure 2.3 shows a software lifecycle in which patterns are available for use in each phase, including anti patterns that can emerge in all phases. While Gotel [GF94] focuses on the traceability of a requirement from its development and specification to its subsequent deployment and use, we focus on the traceability of model elements to patterns. We define Pattern Traceability as follows:

Every model element of a software design and every element in the code can be traced to a pattern.

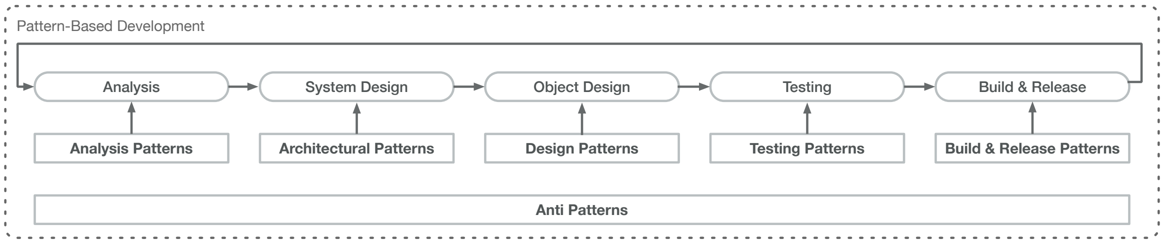

Figure 2.3: Pattern catalog for Pattern-Based Development

PBD builds on the concepts of Pattern-Oriented Analysis and Design (POAD) [YA03] and Pattern-Based Engineering (PBE) [AG10]. POAD represents a methodology for composing proven design patterns into reliable and robust large-scale software systems. It is used to create systems that are robust, scalable, and easy to maintain by utilizing UML class diagrams as building blocks. PBE is a systematic, disciplined, and quantifiable approach to software development that involves the use of pattern specifications and implementations throughout the software development and delivery process.

2.4 Fog and Edge Computing Paradigms

This section addresses the terms and definitions of Fog Computing, Edge Computing, and Cloudlets. Let's first take a look at the historical development [Sat17b]. In the late 1990s, Akamai introduced Content Delivery Networks (CDNs), geographically distributed networks of servers designed to provide availability and performance. Edge computing generalizes and extends CDNs through the use of Cloud Computing infrastructure. In 1997, Brian Noble and colleagues first demonstrated the potential value of Edge Computing for mobile computing by implementing speech recognition with acceptable performance on a resource-limited mobile device [NSN+97]. This idea was taken up by Flinn and Satyanarayanan in 1999 to increase battery runtime by outsourcing computing operations [FS99]. In 2001, Satyanarayanan introduced the term Cyber Foraging [Sat01] and laid the conceptual framework for Edge Computing [SBCD09] that emerged in 2009. Bonomi coined the term Fog Computing in 2012 [BMZA12].

The establishment of these concepts can be traced back to three factors. First, the number of connected devices is continuously increasing, with the result that the existing infrastructure does not meet the requirements. By 2025, 75 billion[3] networked devices are expected to generate a vast amount of data and network traffic. Second, existing software architectures and computing paradigms, notably Cloud Computing, lack location awareness. Furthermore, they do not provide the mobility support required by the agility of mobile devices and their users. Moreover, with the development of IoT in multiple domains, requirements for real-time access and low latency have risen. The objectives of these concepts are to utilize both the scaling and synchronization possibilities of the cloud and its virtualization techniques, and to decentralize the use of available resources, which are usually resource-limited devices in the field, to users on-site. The two worlds of Cloud Computing and IoT merge. Figure 2.4 presents a taxonomy of related terms and their relation.

The superclass *Fog and Edge Computing Paradigms* contains the similarities of all the mentioned concepts. The number of available publications on *Fog Computing*, *Edge Computing*, and *Cloudlets* reflects their importance. Besides the above-

[3]`https://statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/`

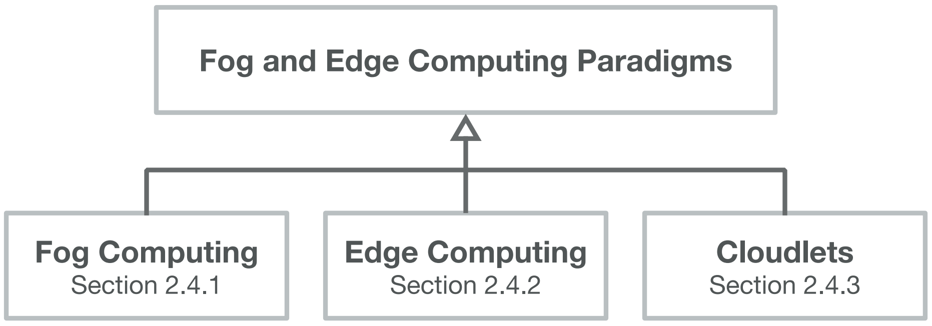

Figure 2.4: Fog and Edge Computing Paradigms taxonomy

mentioned concepts, there are also Dew Computing [Wan15], Mist Computing[4], Small Cell Cloud [OSB15], Femto Cloud [HAHZ15], and Follow Me Cloud [TK13] concepts. The characteristic feature of these concepts is that they address the requirements resulting from IoT. Centralized architectures can only partially meet these requirements, which is why a multitude of new architectural paradigms with different names but the same intention have developed. The establishment of an intermediate layer is a common property. The intermediate layer executes tasks for which the cloud is geographically too far away, or the computing power on an IoT device is insufficient. The concepts differ in the location of the intermediate layer or in the application scenarios for which they are intended. To date, no generic term has been established that unites the different concepts. [HESB18, YHQL15, PTW18] deal with the delimitation of the various terms. [VWB$^+$16, HNYL17, LDZQ18] use the terms Fog Computing and Edge Computing interchangeably, while [DB16] differentiates between them.

In this dissertation and our research, we use the term Fog Computing, but do not distinguish it from Edge Computing or Cloudlets. This is supported by a statement made by Satyanarayanan, who said that "fog computing [..] is consistent with the cloudlet concept" [Sat15].

2.4.1 Fog Computing

For Fog Computing, a variety of definitions have been published in the last few years. Originally, the term was invented by Bonomi, who describes it as a "highly virtualized platform that provides compute, storage, and networking services between end devices and traditional Cloud Computing data centers, typically, but not exclusively located at the edge of network" [BMZA12]. However, since Bonomi describes Fog Computing as a platform and we see it as an architecture (cf. Figure 2.5), we refer to the definition of [YHQL15]:

[4]`https://linkedin.com/pulse/cloud-computing-fog-now-mist-martin-ma-mba-med-gdm-scpm-pmp`

> "Fog computing is a geographically distributed computing architecture with a resource pool consists of one or more ubiquitously connected heterogeneous devices (including edge devices) at the edge of network and not exclusively seamlessly backed by cloud services, to collaboratively provide elastic computation, storage and communication (and many other new services and tasks) in isolated environments to a large scale of clients in proximity" [YHQL15].

The characteristics described provide a better understanding of Fog Computing: low latency and location awareness, wide-spread geographical distribution, mobility, a large number of nodes, predominant role of wireless access, strong presence of streaming, and real-time applications and heterogeneity [BMZA12]. These characteristics are the foundation of the forces leading to the Fogxy architectural style presented in Chapter 4. Comparable to Cloud Computing, which is not an innovation but the integration of past technologies [CLC11], Fog Computing is the combination of existing technologies to meet changing requirements. For example, Bonomi says that Fog Computing extends the Cloud Computing paradigm to the edge of the network, enabling new types of applications and services [BMZA12]. These new applications and services benefit from the use of Fog Computing in terms of real-time access, low and predictable latencies, reduced bandwidth usage, better privacy and security, and uninterrupted services in the presence of intermittent connectivity [PMV+18, Sat17b, CZS17].

According to [VRM14, CZS17, Con17], Fog Computing is about the effective distribution of components of a system between the cloud and the things (in the sense of IoT). The Fog Computing architecture builds on the layered architectural style in which each layer performs a specific role within an application. Fog Computing spreads across the layers *field* (or edge), *fog*, and *remote* (or cloud). The geographical distribution of layers—also called compute domains [MGSB17]—as well as the prevailing heterogeneity regarding hardware and platforms, poses a challenge for the realization of Fog Computing applications. Figure 2.5 shows the hierarchical order of the layers, the characteristics and roles of which we describe in the following.

Devices in the *field* layer generate large amounts of data and can be smartphones, smartwatches, AR glasses, but also PLCs. Due to the geographical proximity of the *field* layer to the end user, devices within that layer can perform real-time critical tasks. However, they are resource-limited, either in computation, memory or storage, but also in size and battery power. By using services from the higher layers, devices in the *field* layer overcome these limitations. The *field* layer connects to the *remote* and *fog* layer via a network.

The *fog* layer features fog nodes that are geographically close to the field, but usually

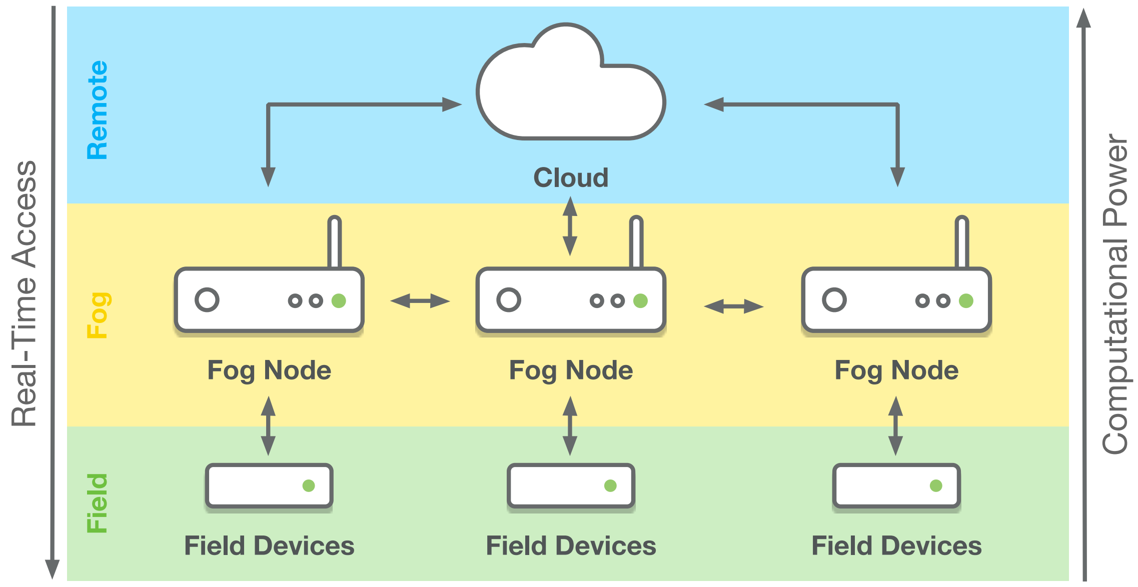

Figure 2.5: Hierarchical structure of Fog Computing

have more resources than devices in the *field*. There are no requirements for a device to be considered as fog node. Therefore, the *fog* layer contains different types of devices such as routers, desktop computers, and servers. The fog nodes perform resource-intensive tasks and facilitate coordination and communication among multiple devices in the field. The *fog* represents the gateway between the *field* and the *remote* layer.

The *remote* layer offers unlimited computational power and storage through efficient devices and scaling possibilities. Typically, Cloud Computing concepts apply within this layer. Especially resource-intensive tasks, such as machine learning or the persistence of large data sets, are carried out there. An advantage of the *remote* layer is the ability to connect to multiple fog nodes to synchronize data among them.

Each layer has its special capabilities—computational power in the *remote* and real-time capabilities in the *field* layer. The *fog* layer enables interaction and mediates between the two layers.

2.4.2 Cloudlets

The concept of Cloudlets is based on the idea of Cyber Foraging, which describes a technique to enable field devices to overcome their lack of energy, computational power, and storage by offloading computation and data to more powerful machines located in the cloud or at single-hop proximity [Sat01, LES$^+$14]. Cloudlets are mobility-enhanced small-scale cloud data centers at the edge of the Internet that overcome the limitations of Cloud Computing. The basic idea is to provide a data center closer to the field device; hence Cloudlets are called “data centers in a box” [SBCD09]. Delay-sensitive and bandwidth-limited applications use Cloudlets, which are comparable to fog nodes since devices can offload resource-intensive tasks to Cloudlets nearby. For latency insensitive requests, Cloudlets interact with the cloud [SCH$^+$14]. Cloudlets aim to provide high-performance computing resources to mobile devices to make them more responsive and extend their battery life, while Fog Computing focuses on the scalability of the IoT infrastructure [Sat17b].

2.4.3 Edge Computing

Edge or Mobile Edge Computing describes an approach to perform parts of data processing and analysis directly on devices at the edge of the network, while the cloud is used for coordination and data archiving [VS17]. Edge Computing describes devices that are deployed close to the end user and provide low latency and high bandwidth. It pushes applications, data, and services away from central servers to the edge of a network [SCZ$^+$16, Sat17b].

Chapter 3
Fog Meta Model

In this chapter, we introduce the Fog Meta Model, an extensible classification of Target Matters. The Fog Meta Model is a metamodel for Fog Computing that uses extensible UML profiles [Obj11] and serves as a starting point for the analysis of Fog Computing applications. It enables the identification of objects and packages that are critical for Fog Computing applications and their non-functional requirements. Section 3.1 describes the objectives and motivation and explains how the metamodel helps to realize Fog Computing applications. Section 3.2 defines the metamodel and describes it in detail. We introduce a new stereotype with three subclasses by extending the UML metamodel [Obj11]. Section 3.3 presents the two application examples FRODO and FARADAY, in which we have applied the Fog Meta Model.

3.1 Objectives and Design Goals

In the field of Fog Computing, the context of hardware nodes plays an important role. The Fog Meta Model allows the association of software requirements to hardware capabilities during the analysis phase. Figure 3.1 expresses this relationship in a high-level UML diagram. Software *Components* are mapped to physical hardware *Nodes* with different *Capabilities* to ensure the fulfillment of *Non-Functional Requirements*. Due to the geographical distribution and different performance *Capabilities*, entities with requirements regarding real-time access, synchronization, or availability must be detected and identified during the analysis phase. The Fog Meta Model supports three types of locality for hardware nodes: *Field*, *Fog*, and *Remote*.

An *Application* is a composite of several *Components* with *Component Artifacts* as leaf nodes. The *Architecture* constrained by the *Non-Functional Requirements* for the *Application* reflects the composition of the *Components* and their interaction. Additionally, *Functional Requirements* for individual components describe the functions of the *Application*. For example, to meet the non-functional requirement of real-time access, the *Component* must be geographically close to the data generating source. *Nodes* of type *Field* are placed directly on-site where the *Application* is used. *Fog*

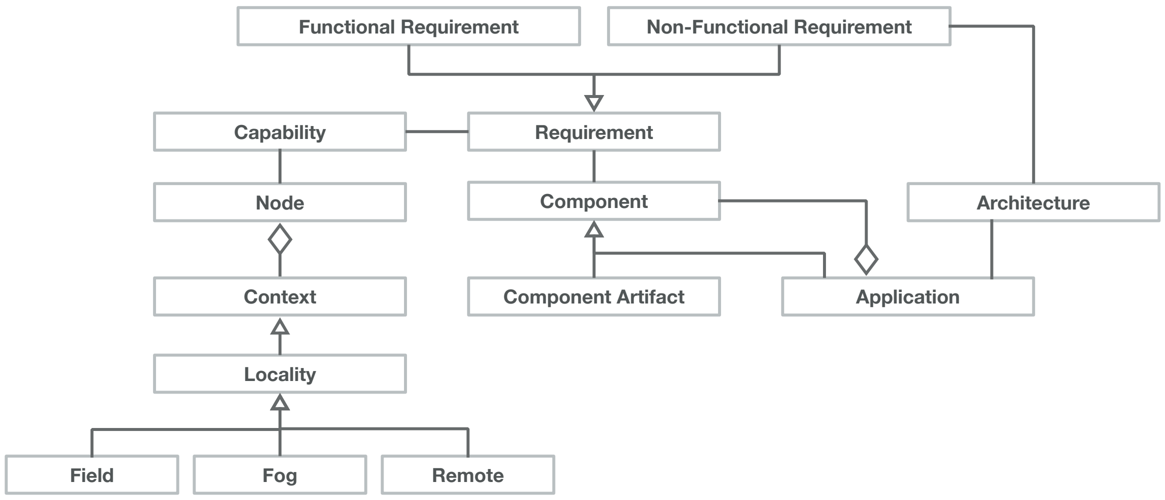

Figure 3.1: Towards the Fog Meta Model

represents the *Locality* between the edge of the network and the *Remote* location. The subclass *Remote* is typically a cloud component that offers synchronization, but is geographically too far away from the source of the data to realize real-time access.

Meta Object Facility (MOF), as defined by the Object Management Group (OMG), is an "extensible model driven integration framework for defining, manipulating and integrating metadata and data in a platform independent manner" [Tan09]. MOF uses object modeling techniques to describe any metadata in the form of metamodels, which is defined as "a model that consists of statements about models" [Jeu09]. The UML meta-metamodel is defined on level M3 of the MOF. The meta-metamodel is used to create the Fog Meta Model, which specifies the concepts of Fog Computing in the form of a UML class diagram. Although UML is the standard language for writing software blueprints, it "is not possible for one closed language to ever be sufficient to express all possible nuances of all models across all domains across all time" [BRJ99]. Therefore, we extend the Fog Meta Model on level M2 to meet the requirements and peculiarities of Fog Computing. UML is based on the Meta Object Facility (MOF) and provides three mechanisms for extensibility: stereotypes, tagged values, and constraints [BRJ99]. We apply the stereotype mechanism to shape and adapt UML to the requirements and needs of Fog Computing by introducing the new metaclass *Target Matter*. On level M1, we define the application design in the form of a style, the Fogxy architectural style, which we discuss in detail in Chapter 4. Fogxy is domain independent, abstract, and aims to be understandable for Fog Computing developers and stakeholders. The Fogxy architectural style is based on the Fog Meta Model and represents software engineering knowledge for instantiating real-world Fog Computing applications within level M0. The following design goals to realize the Fog Meta Model and its application are of importance:

(1) Usability: The expansion of the model should be lightweight and offer low entry barriers. Developers of Fog Computing applications can easily understand it and apply it immediately.

(2) Extensibility: We focus on the simultaneous fulfillment of non-functional requirements for real-time access and synchronization. However, the Fog Meta Model should be extensible concerning further requirement trade-offs and should be able to deal with these.

(3) Communication: Besides the extension of the UML metamodel for Fog Computing applications, the Fog Meta Model serves as communication tool for developers that facilitates communication between stakeholders.

(4) Interoperability: The Fog Meta Model must be applicable in different application domains and therefore should not specialize in the particularities of a domain.

3.2 Design

This section describes the structure and design of the Fog Meta Model. It abstracts core concepts found in different Fog Computing applications. Figure 3.2 introduces the metaclass *Target Matter* to describe the locality and characteristics of a node.

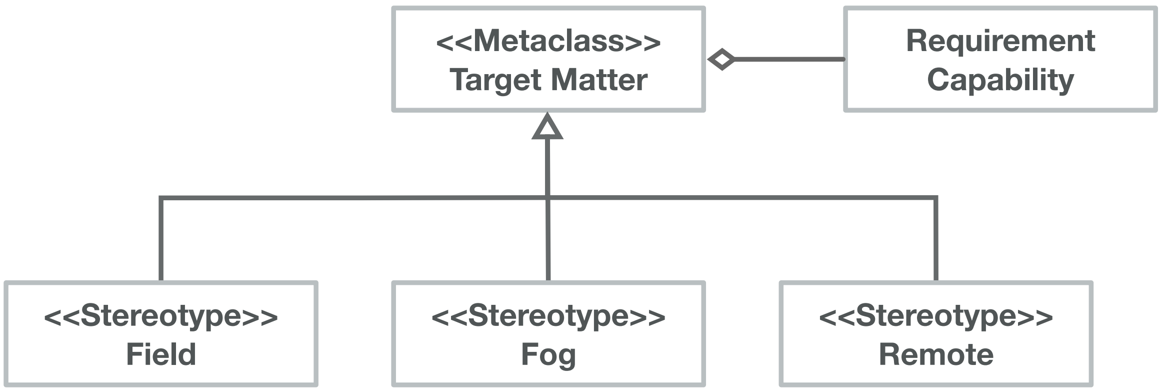

Figure 3.2: Target Matter taxonomy

A Target Matter has specific *Requirement Capabilities* that are determined by a node context, such as geographical proximity or computing power, leading to different features for the subclasses of *Target Matter*. Nodes of stereotype *Field*, which can be further subclassed into mobile and IoT devices, are lightweight and small. Capabilities such as processor speed, memory size, and storage capacity are of lower priority for them, which contradicts the capabilities of the stereotype *Remote*. The *Fog* stereotype represents the intersection between the *Field* and the *Remote*. Classes of stereotype *Field* are executed in-field, classes of stereotype *Fog* in an intermediate layer, between the remote classes of the stereotype *Remote*.

Target Matter derives from the UML object *Class*, which in turn is an instance of the MOF object Class [Obj16]. Figure 3.3 shows the classification of the metaclass Target Matter within the UML meta-metamodel.

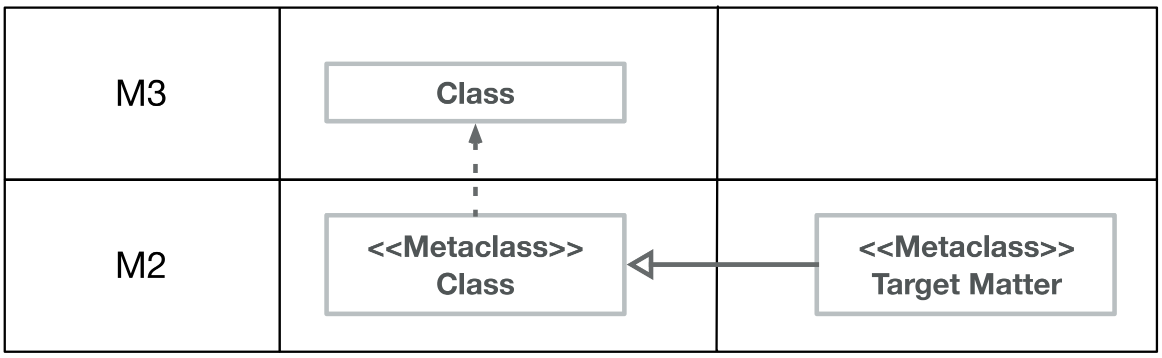

Figure 3.3: Fog Meta Model UML profile extension

These metamodel abstractions essentially allow the formulation of multi-colored graphs during analysis, allowing the analyst to formulate hints for the resulting hardware/software mapping to fulfill the non-functional requirements. We use three different colors: classes of stereotype *Field* are shown in green, classes of stereotype *Fog* are shown in yellow, and classes of stereotype *Remote* are shown in blue. To improve readability, we use the stereotype and color simultaneously throughout the UML models in this dissertation. Figure 3.4 defines the colors for the three stereotypes.

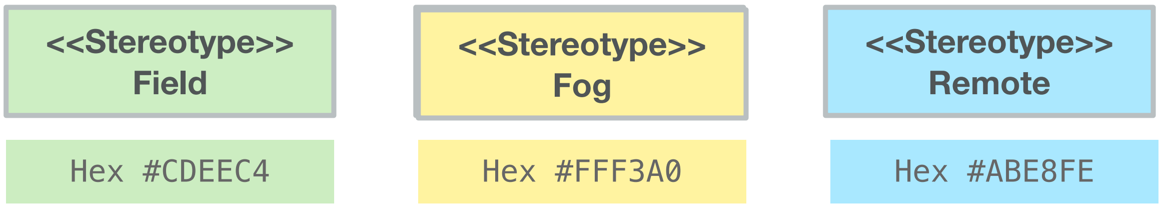

Figure 3.4: Fog Meta Model color scheme

3.3 Application Examples

We present two application examples in which we applied the Fog Meta Model. Section 3.3.1 presents FRODO, a system in the smart environment domain that enables occupants to express their preferences regarding thermal comfort in a decentralized manner. Section 3.3.2 demonstrates FARADAY, an application example from the domain of manufacturing that enables real-time analysis of sensor data for predictive maintenance scenarios.

3.3.1 FRODO

The integration of occupants into smart buildings raises challenges between meeting individual preferences and the generic rule set to optimize energy effectiveness. Merging the individual preferences of multiple occupants that share thermal zones compounds the challenge. To address related challenges, we developed FRODO (Fog Architecture for Decision Support in Organizations), a system designed to establish a location-aware environment for conflict negotiation and decision support. FRODO is based on MIBO, a framework that combines natural and intuitive user interfaces by focusing on multimodal user interaction technology [Pet16]. However, there are limitations posed by MIBO's architecture, such as the integration of the physical context of field devices. MIBO does not deal well with contradictions that arise due to conflicting generic definitions of smart buildings and occupant preferences. MIBO relies on a cloud-based architecture, with deficiencies related to real-time access and availability. The limitations of MIBO and the requirements imposed by individual human comfort require an adopted architecture, as does the goal of increasing global energy savings. FRODO transforms the centralized software architecture of MIBO into a decentralized architecture encompassing sensors, actuators, and the occupants of smart buildings. The transformation is facilitated by the reengineering technique of model refactoring and by the Fog Meta Model. Decentralized fog nodes allow occupants and organizations to express and discuss decision-making conflicts at their point of origin. FRODO captures the characteristics emphasized by Fog Computing, in particular real-time access, availability, security, increased quality of service, synchronization, and geographical distribution.

Figure 3.5 presents the analysis object model of FRODO. The *MIBO Real Subject* and *MIBO Fog Proxy*—children of the abstract class *MIBO Subject*—provide the functionality derived from the legacy system MIBO. The *MIBO Real Subject* represents the real subject class and the *MIBO Fog Proxy* represents the proxy class of the proxy pattern. The proxy pattern enables FRODO to operate independently of the availability of the *MIBO Real Subject* and reduces communication efforts. FRODO performs actions where they are triggered without needing to connect to the remote layer at any time (virtual proxy). The proxy pattern facilitates access restrictions to the remote component and therefore improves security aspects of the system (protection proxy). FRODO refers to the *MIBO Fog Proxy* as fog node and introduces

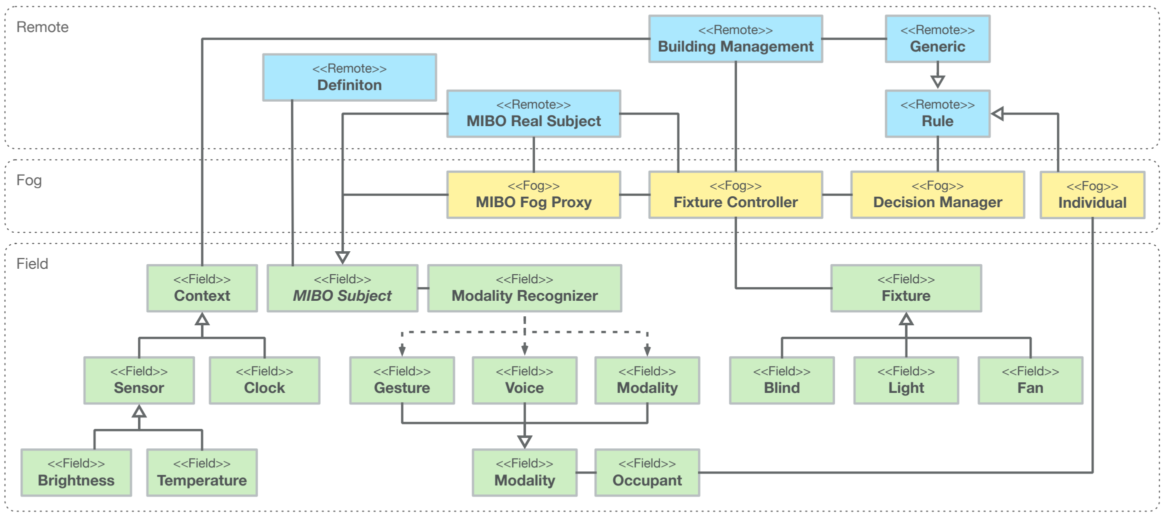

Figure 3.5: FRODO analysis object model

the *MIBO Real Subject* as a fog server [LJY+15]. FRODO relocates components in different environments and geographically distributed nodes, running the *MIBO Fog Proxy* in the fog layer close to the occupant's interactions, whereas the *MIBO Real Subject* remains remote. The *MIBO Fog Proxy* represents a tailored clone of *MIBO Real Subject* with restricted knowledge regarding *Rules* and *Definitions*. It handles *Rules* and *Definitions*, which are required for the specific location it is deployed in, such as a specific room or office space within a smart building. *Fixture Controller* is of stereotype fog and controls directly connected *Fixtures*. In a scenario in which an occupant performs a set of *Modalities* that are part of a definition stored at a *MIBO Fog Proxy*, these *Modalities* are translated to concrete actions within in the node. There is no need to forward the interaction, leading to a reduction in latency time between the performed *Modalities* of an *Occupant*, such as combined *Voice* and *Gesture* commands, and the outcome, such as turning on a *Light*. The *Modality Recognizer* is not aware whether it is connected to either a fog or remote component. We define a taxonomy for *Rules* in FRODO by introducing *Individual* and *Generic* rules. *Individual* rules represent preferences of *Occupants* that are performed depending on a specific *Context*. For example, *Occupants* define a *Rule* that daylight is more essential for them than the optimal illumination of their working place, which results in using less artificial light. *Generic* rules are based on a larger scale and are generally valid for every room in a building. Facility managers define them and they also depend on the *Context*. *Generic* and *Individual* rules might be contradictory. For example, a *Generic* rule defines to shut down blinds at a specific time of the day, whereas an *Occupant* prefers to work by natural light throughout the day. The discrepancy among conflicting *Rules* poses the need for a dedicated process to negotiate between them. FRODO enables negotiation and decentralized decision-making: a *Decision Manager* is part of each fog node and enables the negation between conflicting *Rules*. The *Decision Manager* of FRODO addresses the negotiation of finding the optimal decision for the needs and preferences of *Occupants* and the global goal of energy saving. *Rules* are not the single source for raising conflicts that need to be solved. *Definitions*, which are triggered by events, might interfere with defined *Rules*. For example, an *Occupant* wants to open the *Blinds* in their office. However, if a *Generic* rule stipulates that all *Blinds* remain closed at this point, a conflict situation arises. Therefore, the *Decision Manager* provides conflict resolution strategies to resolve these issues. [SJB$^+$17] discusses the unmentioned classes of the analysis object model.

We show the improvement of the existing MIBO system by applying the Fog Meta Model. Fog Computing allows the decentralized processing of data to interact with connected devices and enables smooth integration of cyber and physical components. Due to the geographical distribution of the fog nodes to locations such as office spaces, we enhance the quality of services with devices in close proximity for occupants. Regarding privacy, the architecture provides the possibility for occupants to identify where their individual preferences are specified—on a remote repository or within their nearby fog node. Fog Computing encourages the seamless integration of heterogeneous smart objects and edge devices, including sensors and actuators. Processing, storing, and communication concepts are realigned with their cloud components, which raises challenges regarding construction and maintenance of the underlying infrastructure. Introducing a decision manager does not solve the problem of securing the requirement of not dissatisfying any objective in general. Instead, it sets the battleground for decision-making processes and requires the specification of solution strategies.

With FRODO, we present a system that handles interest conflicts and supports decision-making processes. FRODO does not solve the conflicts itself but instead establishes multiple decentralized points of interaction to negotiate, discuss, and decide on actions that should be performed to satisfy the individual preferences.

3.3.2 FARADAY

This section presents the FARADAY (Fog Architecture for Real-time and Adaptable Data Analytics) application example. We present the problem and show the analysis object model of FARADAY and validate the application of the Fog Meta Model in the manufacturing domain.

Industrial Internet and the digitalization of manufacturing operations enable the collection and analysis of production related information. The data-driven decision-making and process control help to optimize workflows and reduce downtime, thereby making the manufacturing process more efficient and effective [LKY14]. Cloud platforms offer the prerequisites for analytic approaches as they offer scalable, on-demand access to configurable computing and storage services [DMR16]. However, the cloud concept is ill-equipped to meet the requirements of industrial applications regarding bandwidth limitations, low latency, resilience, and data security [SCM18]. The challenge is not the analysis of the data, but the transmission, processing, and accessing of data in the industrial environment, which is called shop floor or field.

To create a system that can both take advantage of the cloud and meet real-time requirements for data analysis, we analyze the view from the user domain and apply the Fog Meta Model.

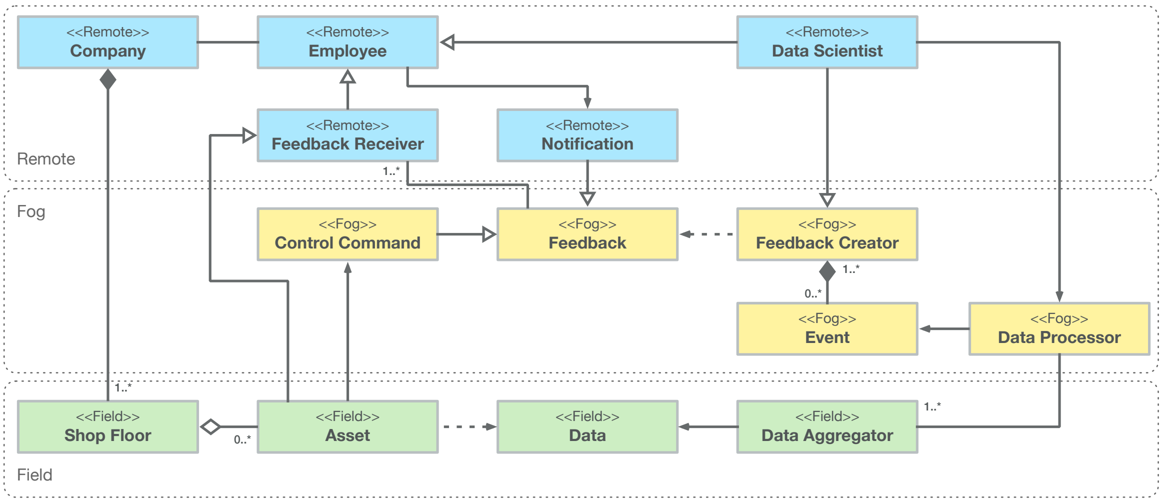

Figure 3.6: FARADAY analysis object model (adapted from [Buc17])

The analysis object model in Figure 3.6 gives an overview of FARADAY and illustrates its main concepts, properties, and relationships visible to the user in the application domain. Classes of the field stereotype represent entities that have to run on-site in the factory to fulfill real-time requirements. Classes of remote stereotype must be available in a remote environment and the fog stereotype types classes that are located between the *Shop Floor* and the remote components.

The *Data Processor* and the *Feedback Creator* are the key abstractions of FARADAY. A *Company* has an economical interest in continuously monitoring, controlling, and optimizing production processes and *Assets* to increase the efficiency and effectiveness of the manufacturing workflow. Measures to achieve this goal are derived from *Data* gathered on the *Shop Floor*. The *Company* employs *Data Scientists* who extract the information from raw *Data*. FARADAY's core functionality is to enable the *Data Scientists* to acquire and process the *Data* and provide *Feedback* on detected *Events*. *Assets* instantiate *Data* collected from sensors. The *Data Aggregator* ingests the raw *Data* and makes it consumable and processable by the subsequent components. The *Data Scientist* configures the properties of the *Data Processor* according to the desired functionality. The *Data Processor* checks the incoming *Data* for anomalies such as limit value violations or peaks. It transforms the input data, for example by calculating a rolling average or batch multiple samples before forwarding them. First, the *Data Processor* triggers an *Event*, once a predefined anomaly occurs. In the latter case, the *Data Processor* provides the transformed data as output. It is also possible to combine both behaviors and to check the calculated mean against a threshold. *Events* are typed and assigned to specific anomalies. Each event contains a policy that defines what *Feedback* is created and published to *Feedback Receivers*. The *Feedback Creator* manages all possible *Events*. If an anomaly is detected, the *Data Processor* triggers an *Event*, whereupon the *Feedback Creator* instantiates and publishes the *Feedback* according to the policy. There are two types of *Feedback*. *Notifications* are displayed in the cloud user interface or sent as an e-mail to company *Employees*. FARADAY provides *Control Commands* for *Assets* providing a suitable interface. The *Data Processor* and the *Feedback Creator* can be located on the *Shop Floor* or in the remote environment and thus provide their functionality both locally and remotely. The demand for short system response time requires the immediate deduction and return of *Control Commands*, enabled by on-site analysis and feedback generation. Results are additionally sent to the remote environment for monitoring and archiving. High available bandwidth or uncritical latency requirements, however, allow the transmission of all or most of the data to the cloud. There, it is further processed and stored for future algorithm training or machine learning. With the capability to run the analysis both locally and remotely, FARADAY provides the flexibility to adapt the network load and the latency to the specific circumstances and requirements of different cases.

Chapter 4
Fogxy - An Architectural Style for Fog Computing

This chapter describes the Fogxy architectural style[1], a style for the realization of Fog Computing applications, using the schema of [BMR+96]. Section 4.1 describes the applied pattern mining approach. Section 4.2 motivates the context in which the architectural style can be applied. Section 4.3 describes the problem the architectural style addresses. Section 4.4 presents the forces that led to Fogxy. Section 4.5 is the center of the chapter focusing on the solution and discussing smart objects in detail. Section 4.6 shows the consequences with benefits and liabilities. Section 4.7 and Section 4.8 show related patterns and list the known uses. Section 4.9 illustrates the application of the architectural style in two application examples.

4.1 Approach

We describe the mining approach for writing and developing Fogxy. Figure 4.1 shows the three activities that were carried out before the initial formulation of Fogxy. We investigated literature dealing with Fog Computing architectures. Many of the presented systems are research projects, and access to documentation and accurate descriptions has been difficult. The presented approaches are usually described in scientific papers, and therefore more profound insights regarding the architecture are not possible. However, we were able to identify similarities between the different architectures that influenced the design of Fogxy. The design of Fogxy was influenced by the goal to simultaneously address non-functional requirements that used to be mutually exclusive.

We proceeded iteratively and used Fogxy in several application examples from different domains. After the realization of each application example, we evaluated the Fogxy architectural style. We considered the lessons learned and used the takeaways to enrich Fogxy. The learnings were compared and matched with the requirements. If characteristics concerning the formulation and description of the design were identified, Fogxy was adapted accordingly.

[1]The architectural style was accepted at the 2018 EuroPloP conference [STB18b].

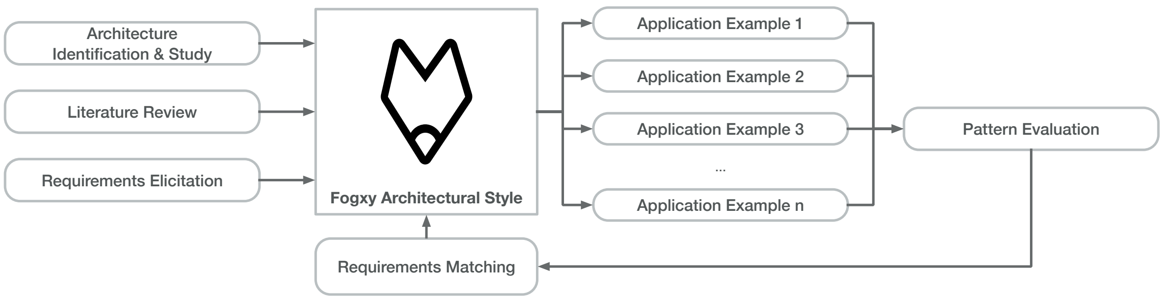

Figure 4.1: Pattern mining approach

4.2 Context

IoT applications have become popular in recent years and are used in industry and private sectors. Sensors and actuators are used to sense environments and perform actions based on data. IoT applications are not limited to one domain but find their way into sectors from health to industrial production and autonomous vehicles. Applications in these areas impose requirements that cannot, or can only partially, be solved by existing cloud technologies. Therefore, Fog Computing has emerged as an extension of the Cloud Computing paradigm to the edge of the network to enable new types of applications and services [BMZA12]. The Fogxy architectural style can be applied to applications in different domains and presents an architectural solution that meets real-time, synchronization, or availability requirements that cannot be fulfilled by cloud-only architectures.

4.3 Problem

The simultaneous realization of non-functional requirements such as synchronization and real-time access represents a challenge for existing software architectures. Existing architectural styles address these individually, but do not allow the easy formulation of trade-offs between them. Fogxy deals with this problem:

> *How can we solve the limitations of Cloud Computing solutions, such as high latency and missing real-time support, for IoT scenarios?*

The cloud is essential for IoT scenarios [GBMP13], but has drawbacks: real-time applications suffer from the high latency imposed by the physical distance of devices to the cloud. Since the devices depend on the availability of the cloud, their functionality is impaired if the communication is disrupted. Nevertheless, Cloud Computing provides various advantages regarding scalability, synchronization, computing power, and portability. The aim is to combine IoT and Cloud Computing while retaining positive aspects and eliminating negative ones.

4.4 Forces

The forces leading to the use of the Fogxy architectural style are based on the characteristics of Fog Computing [BMZA12]:

Low Latency and Real-Time Interaction. IoT applications have real-time requirements and demand low latency. Centralized solutions are not suitable because of their geographical distribution and the resulting communication overhead.

Heterogeneity. IoT applications use specific hardware and software. Fog Computing is not yet standardized and ranges from embedded devices without an operating system to virtualized cloud solutions. Many devices and components are part of the system, are distributed, and must collaborate across networks to provide the desired functionality. Given the heterogeneity of the different nodes, interoperability must be established and ensured. Both hardware and software components are heterogeneous, as is data that is analyzed and processed.

Resource Limitation. Devices used in field and fog layer have special purposes and are limited in terms of memory, computing power, and battery. Depending on the application, dedicated hardware is used that, for economic reasons, should be inexpensive to purchase and operate. In IoT scenarios, it is particularly important to achieve the longest possible battery life for mobile devices or to keep operating costs as low as possible. It is, therefore, necessary to use resources as efficiently as possible.

Geographical Distribution. The cloud is located in central data centers, while fog nodes can be distributed everywhere. Applications are possible in remote locations without an Internet connection. Intelligent orchestration of the distributed components is required to enable effective collaboration.

Inadequate Infrastructure. The requirements regarding network speed and availability are continually increasing. Increasing data collection, data processing, and the rising number of requests pose a challenge to the existing infrastructure. Increased network load and inadequate or unavailable infrastructure limit existing cloud solutions. However, as infrastructure expansion is slower than IoT application development, a solution is needed.

Interplay with the Cloud. Embedded devices and cloud solutions have both advantages and disadvantages. For example, cloud components are dynamically scalable, while embedded devices can meet real-time requirements. The combination of both technologies enables the elimination of their disadvantages. Cloud components perform computationally or memory intensive tasks, while embedded devices perform real-time critical tasks.

Access Control. Given the large number of Smart Objects and the amount of data, access control to cloud components represents a challenge. Fog Computing requires access restrictions for services and data. Authentication and authorization solutions regulate the access of Smart Objects to cloud components.

Location Awareness. The multitude of devices and their mobility pose a challenge. To ensure satisfying service delivery, environments must be aware of the devices currently contained. An environment traces the devices entering and leaving it to carry out orchestration, task distribution, and service provisioning.

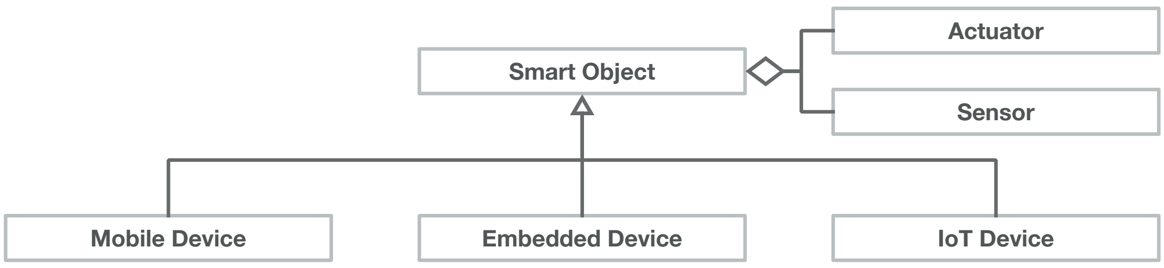

Figure 4.2: Smart Object taxonomy

4.5 Solution

Fogxy addresses real-time and synchronization by decoupling heterogeneous *Smart Objects* from centralized components. By introducing different layers, system components can be distributed along a remote-field continuum. Services are geographically distributed and offered where they are needed. The decentralized deployment of *Smart Objects* and its services enable real-time critical applications. We apply Kruchten's 4+1 view model for software architectures to describe the architectural style in the following [Kru95]. Figure 4.3 shows the logical view, Figure 4.4 the development view, and Figure 4.5 the physical view. We use a scenario in Section 4.5 to present the dynamic aspect of Fogxy.

Structure

In the context of IoT, *Mobile Devices*, *Embedded Devices*, and *IoT Devices* are equipped with *Sensors* and *Actuators*. Sensors are used to measure and monitor the environment, while actuators control it. *Smart Object* in Figure 4.2 generalizes mobile devices, embedded devices, and IoT devices [Sch17].

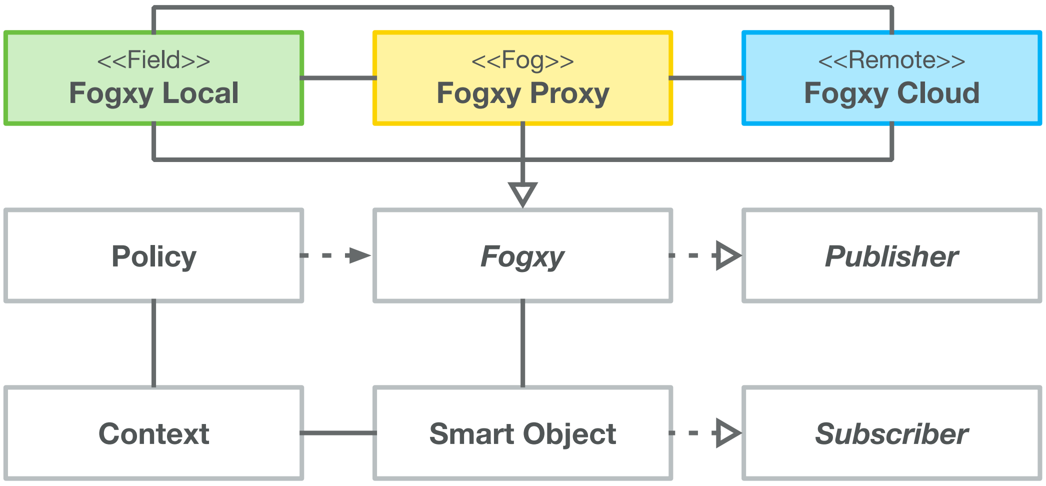

Figure 4.3: Fogxy architectural style overview

Figure 4.3 shows an overview of Fogxy. The *Context* class describes the context of *Smart Objects*, such as battery power, network connectivity or computational power. The *Policy* considers this information along with real-time and availability requirements to select a *Fogxy* strategy for reaching *Fogxy Local*, *Fogxy Proxy*, or *Fogxy Cloud*, as described in the strategy pattern [GHJV95, BD09]. The object interacts with Fogxy based on the selected strategy. Fogxy uses the proxy pattern, in which the abstract *Fogxy* represents the subject, the *Fogxy Proxy* the proxy, and the *Fogxy Cloud* the real subject. The proxy pattern is essential to Fogxy, because it enables availability regardless of cloud availability. Furthermore, the network load is reduced as requests merely need to reach the *Fogxy Proxy* and not the cloud. This reduces latency and enables real-time applications. The observer (publish/subscribe) pattern facilitates backward communication between *Fogxy* and the *Smart Object*, enabled by the two interfaces *Publisher* and *Subscriber* [BHS07]. The *Smart Object* subscribes to updates, while the Fogxy instance publishes information to all subscribed *Smart Objects* via broadcast or multicast. Similar to the observer pattern, *Smart Objects* register themselves as subscribers to the publisher *Fogxy*. Therefore, *Fogxy* is aware of currently subscribed *Smart Objects* and traces entry and exit of *Smart Objects*.

Component View

Following the idea of a layered architecture, Fogxy distributes components on three different layers: *Field*, *Fog*, and *Remote* (cf. Figure 4.4).

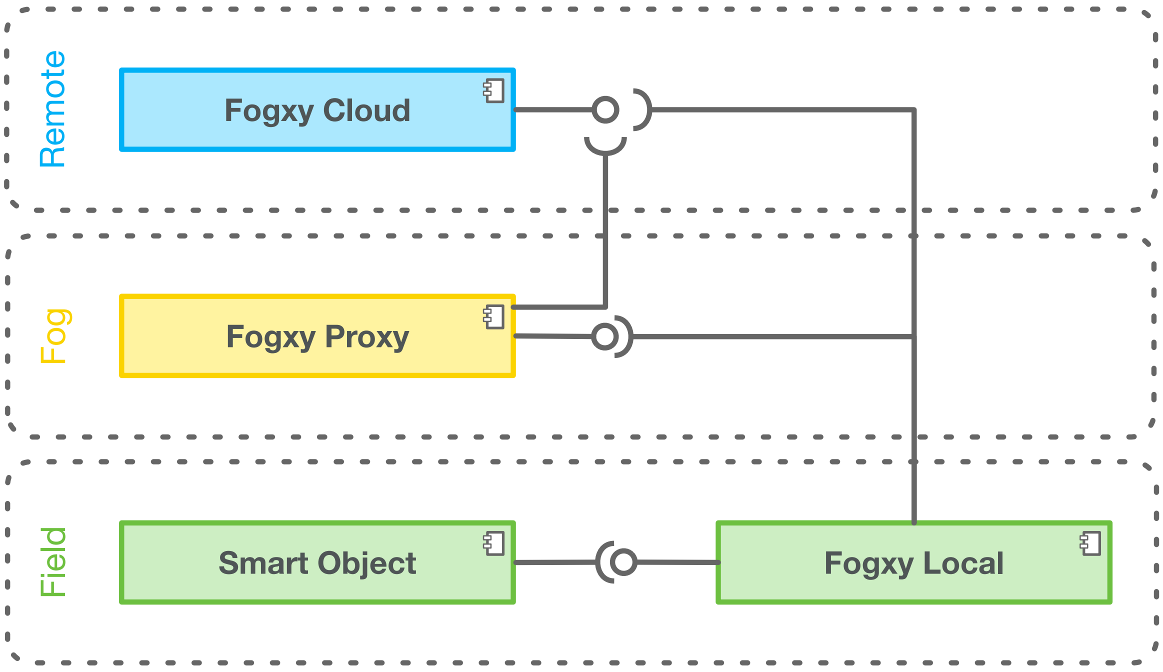

Figure 4.4: Fogxy architectural style component view

Field components (lower layer; green) are located near the end-user and comprise a *Smart Object* component and a *Fogxy Local* component. The *Fogxy* component offers an interface for the *Smart Object* to interact. *Fogxy Proxy* itself acts as a proxy (cf. proxy pattern [GHJV95]), allowing the *Smart Object* to communicate with local, the fog, or the remote. The fog (middle layer; yellow) is in close vicinity to the field and offers computing resources. The remote (upper layer; blue) is a centralized, location independent computing and storage provider. The layers are arranged hierarchically and enable the fulfillment of the various non-functional requirements.

Fogxy Local can request services of *Fogxy Cloud* via *Fogxy Proxy* or directly. In situations where a *Fogxy Proxy* may not be available, the *Fogxy Cloud* serves the requests of *Fogxy Local*. Fogxy differs from the proxy pattern because it does not assume that a proxy object is always available. In Section 4.5, we discuss different strategies and the routing between the layers.

Deployment View

The deployment view shows the mapping of software components to hardware nodes. Figure 4.5 depicts the *Client Node* containing both the *Smart Object* and the *Fogxy Local* component. Individual deployment units depend on the context of the project, ranging from personal computers to embedded devices. Multiple nodes can connect to either *Fogxy Local*, *Fogxy Proxy*, or *Fogxy Cloud*. *Fogxy Local* runs on the hardware component that also uses the *Smart Object*. The *Fogxy Proxy* is deployed on a *Fog Node*. Each of the fog nodes and the client nodes can connect to a *Cloud Server*. If the *Fogxy Proxy* cannot handle requests, or they are not real-time critical, the *Cloud Server* serves as a fallback. Additionally, the cloud offers storage space and computing resources for more performance draining requests.

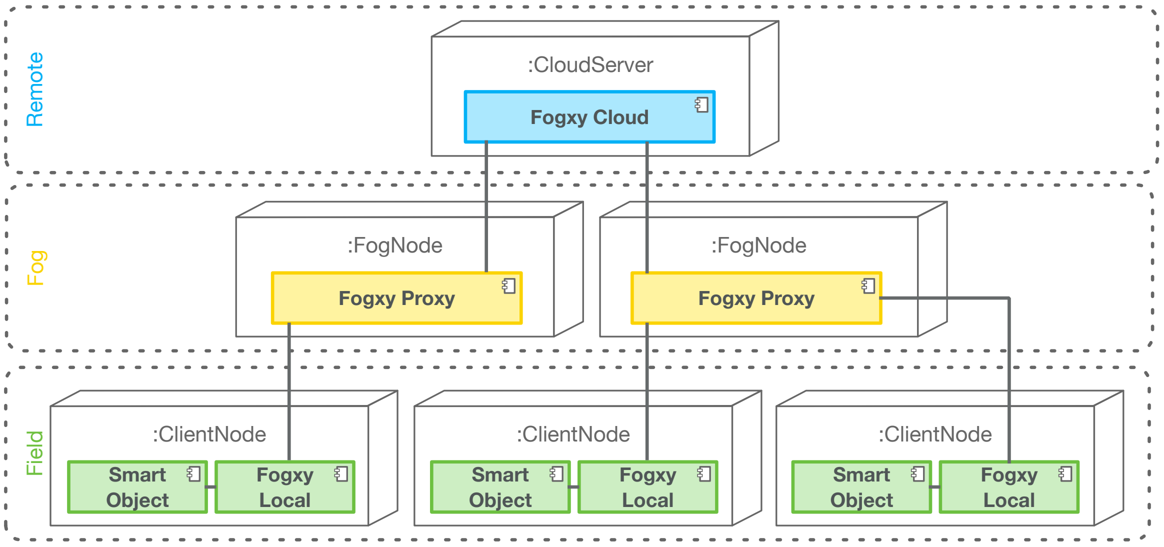

Figure 4.5: Fogxy architectural style deployment view

Dynamics and Routing

The communication diagram in Figure 4.6 shows the component interaction within Fogxy to serve a request. The *Context* class considers different parameters that determine which component processes a request. Possible influences are the availability, the existing network connection, the respective computing power, battery consumption, or real-time requirements. The goal is to identify an optimum of the various influences to be able to process requests as effectively and efficiently as possible. The definition of the optimum is application-dependent and specific to the problem domain. The *Policy* defines which instance of Fogxy handles a request. The following example shows the dynamic behavior and routing for latency-sensitive and latency-insensitive requests. For a request with low latency, the *Smart Object* sets the *Context* as latency sensitive. The *Policy* observes the *Context* and selects the fog communication strategy. The *Sensor* senses an event and forwards it to *Fogxy*. As the *Policy* has chosen a fog communication strategy, Fogxy performs the request on the *Fogxy Proxy* (cf.

step 5b in Figure 4.6). The *Fogxy Proxy* handles the request and an *Actuator* reacts to the response. The observer pattern realizes the communication and synchronization of the *Actuators* [GHJV95]. The *Fogxy Local* component may be available but cannot service the request because of the limitations of processing power or battery. In contrast, requests that do require high computational power are serviced by the *Fogxy Cloud*. The *Policy* decides for a cloud communication strategy and thus the *Fogxy Cloud* performs the requests (cf. step 5c in Figure 4.6). Depending on the type and requirements of the required service the specific instance of *Fogxy* to handle the request is decided at runtime.

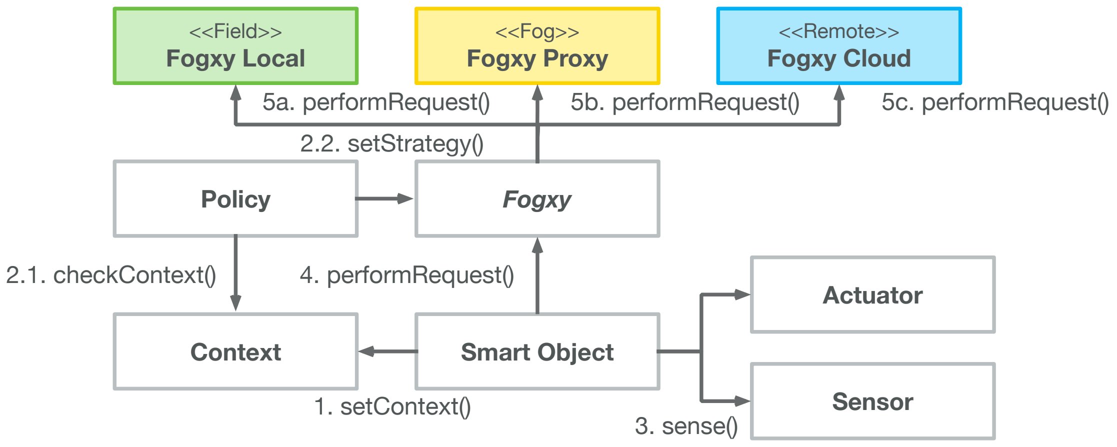

Figure 4.6: Fogxy architectural style communication dynamics

Variants

According to the proxy design pattern, there are also three variants of the Fogxy architectural style. These variants are based on the variants of Buschmann and Gamma [BMR+96, GHJV95].

Caching Fogxy. *Fogxy Proxy* can act as a cache in both directions. Data collected by sensors can be processed and temporarily stored and forwarded to the cloud as required or available. In the same way, results already provided by the cloud can be made available in the fog.

Substitute Fogxy. *Fogxy Cloud* or *Fogxy Proxy* may not be available. We assume that *Fogxy Local* is always available, but cannot meet application requirements and therefore require fog or cloud services. Due to the mobility of *Smart Objects*, a fog node with the corresponding service might not be nearby, available or accessible. The child classes of *Fogxy* act as a stand-in for each other. They are not identical, but offer corresponding services according to the possibilities available. There may be requests that can be handled quickly by the *Fogxy Proxy*, but may not deliver as accurate results as with execution on a cloud component.

Access Control Fogxy. Another variant of the Fogxy architectural style is that *Fogxy Proxy* acts as an access control or security gateway. Thus, the proxy can determine which request and data from the multitude of possible *Smart Objects* are forwarded to *Fogxy Cloud*.

4.6 Consequences

The following benefits can be achieved by using the Fogxy architectural style:

Real-Time Access. The *Fogxy Proxy* offers services for *Smart Objects* in close proximity. If the *Proxy* and *Smart Objects* are on the same network, latency is reduced and IoT requirements are met. Instead of processing requests in the distant cloud, a fog node allows requests to be handled locally. Local processing reduces latency and enables the realization of real-time critical applications. Answers to requests may already exist in the *Fogxy Proxy* and requests to the cloud can be avoided. The architectural style serves as a Caching and Substitute Fogxy.

Interoperability. *Smart Objects*, *Fogxy Local*, *Fogxy Proxy*, and *Fogxy Cloud* are self-contained components that collaborate to meet non-functional requirements. The Fogxy architectural style enables interoperability of different hardware nodes and software components through clearly defined interfaces and task assignments. Fogxy allows existing infrastructure to be reused, thereby allowing already existing computing resources in vicinity to act as a fog node. This is an economic benefit as fog nodes are already in place and do not have to be deployed.

Overcome Resource Limitation. Resource-limited devices can offload tasks and thus overcome their limitations. By offloading to devices in the layers above, resources are utilized efficiently and requests are processed where they can be executed best.

Geographical Distribution. Cloud components are centrally located and fog nodes are decentralized. The advantages of both approaches can thus be combined. The availability of systems can be increased because either the cloud is available if there is no fog node nearby or the other way around. If the cloud is not available or accessible, a fog node with the corresponding services may be available (cf. *Substitute Fogxy*).

Reduced Network Load. The amount of data transferred to the cloud can be minimized by processing requests in the *Fogxy Local* or *Fogxy Proxy*. Application components can filter data and requests, reducing the network load between fog and remote. It restricts the transmission of unnecessary data and prevents inefficient bandwidth consumption. At best, the data is processed where it is needed (cf. *Caching Fogxy*), leading to reduced network traffic and available bandwidth [GGdFP+16].

Interplay with the Cloud. Fogxy as an architectural style for Fog Computing does not replace Cloud Computing, but enables the effective interaction of IoT components with Cloud Computing. Computational tasks that are not time-critical or require central data storage are linked to services that have real-time requirements. Fogxy does not specify where requests must be executed. As described in Section 4.5, Fogxy provides the foundation for performing requests on different layers in the cloud-thing continuum.

Access Control. The *Access Control Fogxy* variant enables the authentication and authorization of *Smart Objects* to fog and cloud components. *Smart Objects* have different access rights to the components.

Location Awareness. *Fogxy Proxy* is aware of its *Smart Objects* and thus enables location awareness. *Smart Objects* nearby interact with the *Fogxy Proxy* and access offered services. Services can thus be tailored to the needs of users and devices.

The application of the Fogxy architectural style entails the following liabilities:

Dealing with Complexity. The complexity of applications utilizing Fogxy increases due to the multitude of different components and their interaction. Responsibilities and interfaces must be defined appropriately.

Testability. Due to the different application contexts and the availability of cloud and fog components, testability is a challenge.

Integration and Deployment. The variety of components, their heterogeneity, and their distribution raise challenges for the integration and deployment of applications. The integration of as many different components as possible into fog applications offers advantages, but also entails increased effort for the provision of corresponding integration and deployment solutions.

4.7 Related Patterns

Latency problems have occurred several times in software architecture history. For example, the systems mentioned in the introduction were solved in mainframes by *Memory Cache Hierarchies* or on the Internet using the *Proxy* pattern. We summarize these technologies under the generic term reduced latency architectures and introduce a taxonomy for it (cf. Figure 4.7). A memory cache hierarchy increases the computing performance of mainframes and personal computers. The communication delay caused by the distribution of client and server was reduced by introducing proxies that cache content. These patterns serve as an inspiration and valuable resource for Fogxy.

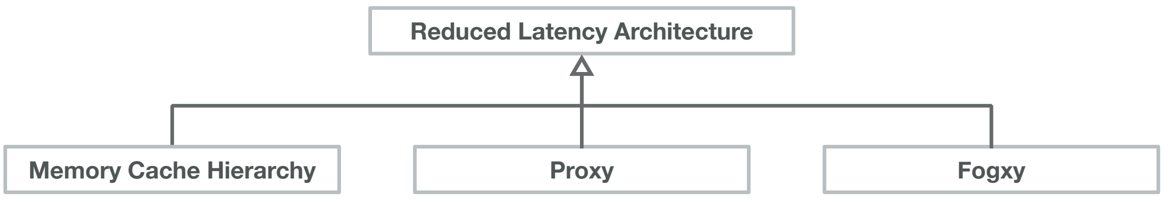

Figure 4.7: Reduced Latency Architectures taxonomy

REST is as an architectural style for network-based software and is based on the proxy pattern [Fie00]. Both REST and Fogxy aim to improve network efficiency. REST constraints require requests to be explicitly or implicitly flagged as cacheable or not cacheable. This enables interactions to be partially or completely eliminated and improves the efficiency, scalability, and user-perceived performance. *Fogxy* achieves identical goals by using the proxy pattern, but it is more general than REST, because REST can be used within *Fogxy* to enable communication between the different components.

Syed and his colleagues formulated another Fog Computing pattern. Their pattern defines Fog Computing as a “virtualized platform that stands between Cloud Computing systems and Internet devices” [SFI16]. This platform provides computing, storage and networking services between cloud vendors and edge devices. In contrast to Syed’s approach, Fogxy is not a platform, but an architectural style focusing on the context of different components.

The Open Fog Consortium provides a generic reference architecture without platform dependencies from a high granularity point of view [Con17]. Both architectures feature the intermediate layer (fog) between the edge (field) and the cloud. However, Fogxy provides a lower granularity and presents a detailed solution guide to implement Fog Computing applications.

4.8 Known Uses

We applied Fogxy for applications in different domains:

- **FRODO** [SJB+17]: A system for decentralized decision-making in smart home applications. Decisions about temperature or brightness are no longer made centrally, but decentralized with the help of fog nodes. Discussed in Section 3.3.1.
- **FARADAY and FEAt** [SBB18]: Two industrial application scenarios in which Fogxy is used to increase the availability of production facilities on the one hand and to enable the analysis of sensor data in real-time on the other hand. Discussed in Section 4.9.1 and Section 4.9.2.
- **AIIoT** [SHS+18]: AIIoT links Fogxy with augmented reality and machine learning technologies. Industrial *Smart Objects* are detected, identified, and the attached sensor data is visualized in real-time using emojis. The use of Fogxy allows the combination of technologies and the interpretation of sensor data in real-time. Discussed in Section 7.1.
- **IIoT Bazaar** [SHM+18]: The IIoT Bazaar is the continuation of the AIIoT project. It is an app store for edge applications that can be provided to factory workers on edge devices using drag & drop in place. The IIoT Bazaar shows that Fogxy also works in conjunction with Blockchain technology. Discussed in Section 7.2.
- **IPRA** [HSHB18]: IPRA uses Fogxy to perform the computationally intensive operation of person identification using smart glasses on a nearby fog node. The goal of IPRA is the simultaneous fulfillment of real-time requirements and the runtime extension of wearables through computational offloading.

4.9 Application Examples

This section demonstrates the applicability of the Fogxy architectural style in two examples from the manufacturing domain. In Section 4.9.1, FARADAY presents the integration of Fogxy into an existing manufacturing system for production processes. Sensor data is collected for analysis, processed in real-time, and synchronized. In addition to the requirements of real-time access and synchronization, Fogxy reduces network traffic. Section 4.9.2 presents FEAt, which simultaneously increases its availability and offers real-time access for the analysis of sensor data. Both application examples show that Fogxy is applicable in the industrial domain and helps to overcome problems associated with cloud deployments.

4.9.1 FARADAY

With FARADAY, we evaluate Fogxy regarding traffic reduction in an enterprise network and the requirements for low latency that cannot be met with cloud-only approaches. FARADAY must adapt to available bandwidths in varying networking infrastructures for industrial use. This implies reducing the transmitted data as far as possible and increasing the information forwarded to the cloud to enable further processing, model training, and storage.

Design

Figure 4.8 shows the packaged analysis object model of FARADAY. It refers to the analysis object model described as an application example in Section 3.3.2. We group related classes in packages and show how Fogxy actualizes the system architecture. To perform an initial division and facilitate further system decomposition, we structure the analysis object model according to the *Manufacturing*, *DataHandling*, and *Feedback* packages.

The *Manufacturing* package contains all entities involved in the company's manufacturing process. The *Assets* on the *Shop Floor* provide *Data* and obtain *Control Commands* that are propagated to control the manufacturing process. The *Data Processor* is the central component of the *DataHandling* package. It runs locally or remotely and processes the data received from the *Data Aggregator*. If an anomaly occurs, it triggers an *Event*, which defines the policy for the *Feedback* creation. The *Feedback* package includes the *Feedback Creator*, which generates concrete *Feedback* instances, such as *Notifications* or *ControlCommands*. The *Data Scientist* defines these policies and thus determines which *Feedback* is published.

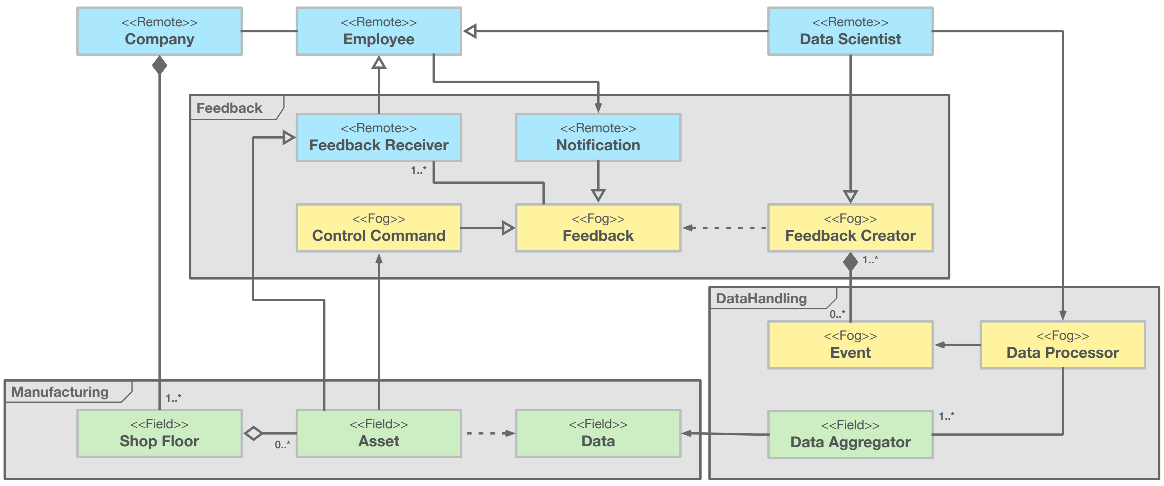

Figure 4.8: FARADAY packaged analysis object model

The architecture of FARADAY is based on Fogxy and enables data processing in the field and fog to reduce network traffic and system response times. FARADAY consists of a lightweight core that is dynamically extensible by components with further functionality. The intra-node communication between the components on the same node is based on a publish-subscribe messaging service. Inter-node communication uses the MQTT[2] protocol. FARADAY comprises the following components: a communicator component mediates between the intra-node and the inter-node communication and provides the routing capability necessary to forward messages to the destination node. The *Data Aggregator* interfaces assets and machinery on the *Shop Floor* to collect data. This data is analyzed by a *Data Processor* that triggers events in case of anomalies. The *Feedback Creator* transforms these events into feedback messages, which can either be human-readable notifications or control commands for *Assets*. In the latter case, the *Asset Controller* executes the connection and control of each *Asset*. Depending on the component distribution across *Field-*, *Fog-*, and *RemoteEnvironment*, FARADAY can be optimized for latency, bandwidth, and reliability requirements of specific use cases.

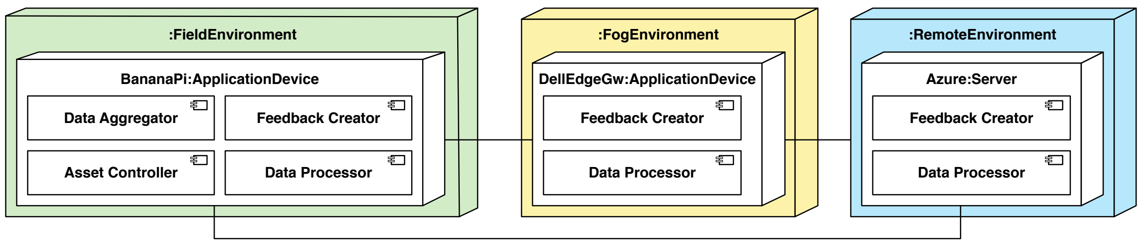

Figure 4.9: FARADAY deployment diagram

[2] `http://mqtt.org/`

To assess FARADAY's capabilities regarding real-time access and network traffic reduction, we evaluate the effectiveness of techniques such as local data analysis or data preprocessing and compression on field or fog nodes with the setup shown in Figure 4.9. We adopt an industrial predictive maintenance application for our evaluation scenario in which we monitor vibrations and temperatures of electric motors to detect attrition and imminent failures. Figure 4.10 visualizes the test setup. A Banana Pi Single Board Computer (1) equipped with an acceleration sensor (2) and a temperature sensor (3) serves as a field device. The DellEdgeGateway (4) serves as the fog node and is wired via ethernet. The components in the remote layers were deployed on Microsoft Azure. While the data collection and processing is performed on hardware, we simulated a control loop by logging the received feedback messages and the respective latency on the field device.

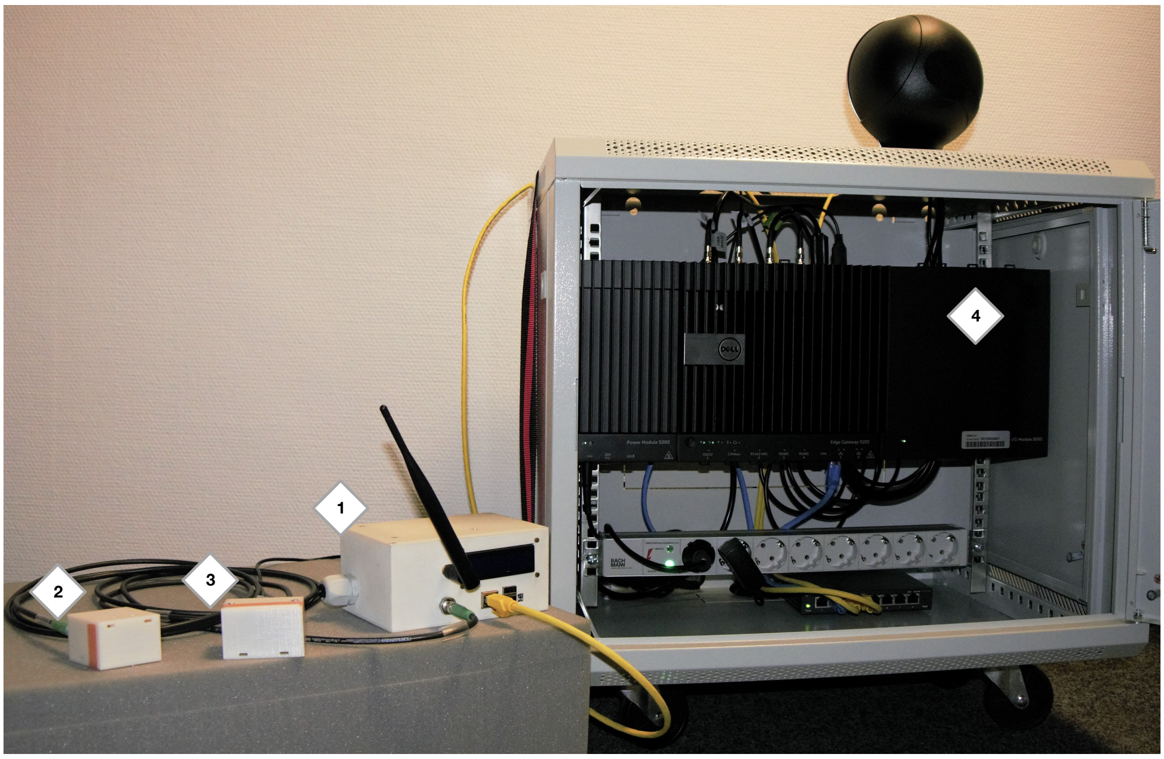

Figure 4.10: FARADAY test setup

Results

Table 4.1 shows the duration between the data sample recording and the reception of a feedback message when performing the analysis in the field, fog, or remote layer. Wireshark captured the average feedback latency between the involved devices.

Field-only processing yields the fastest feedback with an average latency of 6.53 ms. Data processing on the fog node increases latency to 11.95 ms. Therefore, 5.52 ms

	Field-Only	**Field-Fog**	**Field-Remote**
Remote	–	–	1.26 ms
Fog	–	5.72 ms	–
Network	–	0.71 ms	37.82 ms
Field	6.35 ms	5.52 ms	5.83 ms

Table 4.1: FARADAY processing time

(46.19 %) are attributable to the field node, 5.72 ms (47.87 %) on the fog node, and the remaining 0.71 ms (5.94 %) represent the network transmission time. The field-remote solution achieves a response time of 44.91 ms. The more powerful cloud hardware decreases the time for feedback creation to 1.26 ms, an improvement by a factor of 4.54 compared to the fog node. Due to the larger spatial distance, the transmission time increases to 37.82 ms (84.21 % of total latency). These results confirm the assumption that a fog approach offers real-time benefits compared to cloud-only deployments. The techniques presented for traffic reduction on a field or fog node reduce the outbound data rate and thus decrease the load on the network. Data compression especially appeared to be a powerful approach. Figure 4.11 visualizes the bandwidth savings achieved by compressing sensor readings or batches of acceleration data sampled at 1000 Hz. The compression of aggregated data batches increases effectiveness, but also introduces additional delay.

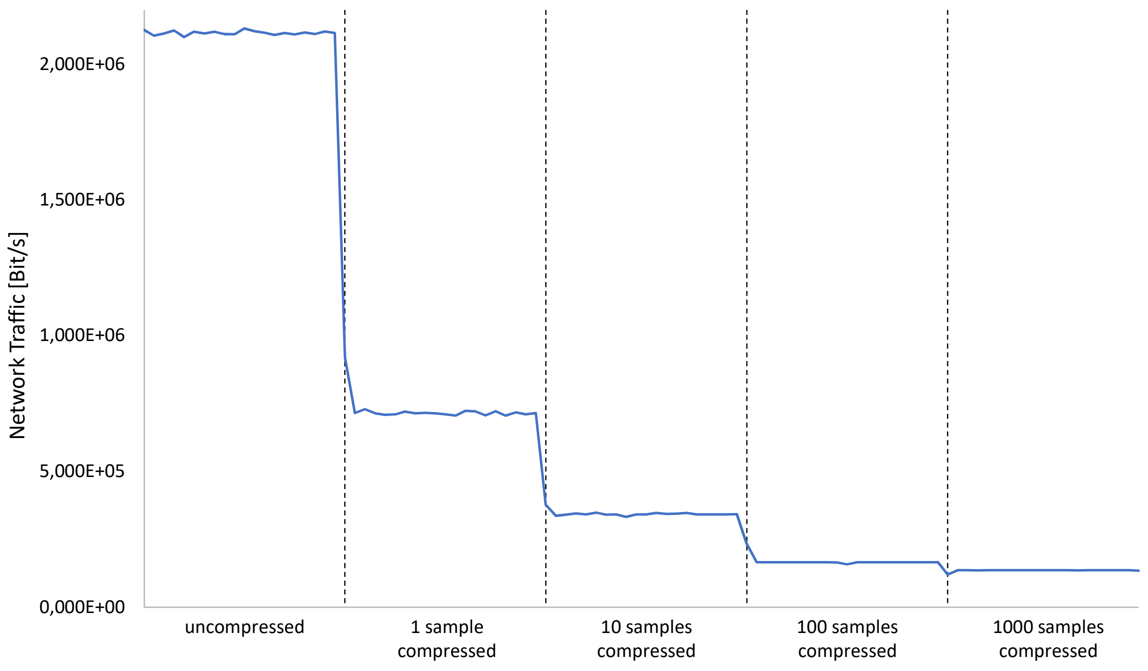

Figure 4.11: FARADAY bandwidth savings

Besides the trade-off of traffic reduction and latency, further parameters exist that influence the Quality of Service (QoS) provided by the system. The weighting of the entries of the QoS-vector and the resulting system behavior are dependent on the application and requirements. The capability to adapt the QoS-vector to meet the respective circumstances is essential to targeting a broad range of uses. FARADAY offers this adaptability due to its fog-based and modularized architecture. Its extensibility enables the integration of external components to be upgraded and tuned for further applications. The increased internal messaging effort and the need for general interfaces induced by the modularity and flexibility impair system performance compared to specialized solutions. Shifting performance critical tasks to dedicated systems integrated within FARADAY addresses this drawback.

4.9.2 FEAt

FEAt (Fog-Based Resilient Edge Application) addresses a factory located in a rural area with a poor Internet connection. A simplified industrial manufacturing factory served as an example. The factory uses the following steps to process goods: incoming material is checked for completeness and intactness in the incoming material inspection and is placed in the warehouse. Material required to manufacture an order is transported from the warehouse to the manufacturing cells. An overhead crane reaches all manufacturing stations, including the warehouse and quality assurance.

Figure 4.12 shows the Lego Mindstorms crane that simulates the factory and provides sensor values. Three manufacturing cells execute different processing steps: milling, welding, and drilling. The crane transports products in a cell and continues after a defined period of time. After production, the finished products are quality assurance tested and dispatched from the warehouse. Fogxy enables FEAt despite a poor Internet connection and enables the real-time processing and synchronization of sensor data.

Design

The business and operations related systems run centrally in a remote data center which controls the operations of the factory: production plan, execution instructions for manufacturing cells, acquisition of operating data from machines and environmental sensors, and the creation of work orders. These operations rely on a working Internet connection.

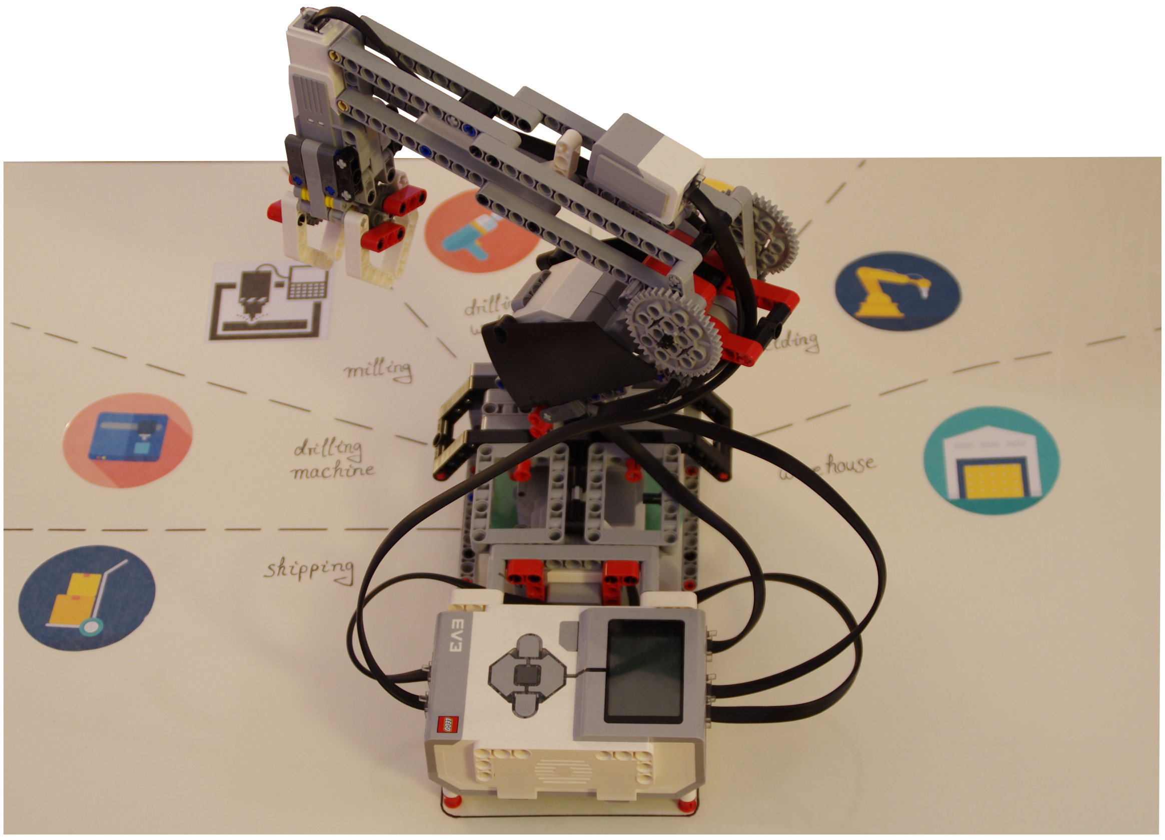

Figure 4.12: FEAt test setup

Results

Applying Fogxy to FEAt increases service availability and minimizes bandwidth limitations while enabling real-time access for sensor value data. To validate the assumptions, we deploy FEAt with different configurations and measure the round-trip time (RTT) and bandwidth.

The components of the *RemoteEnvironment* are placed on geographically distributed servers provided by AWS[3] (EU-Central, EU-West, and US-West). Table 4.2 shows that, compared to *RemoteEnvironments*, the latency from the *Field-* to the *FogEnvironment* is smaller and the available bandwidth higher. Considering the low standard deviation of *FogEnvironment's* latencies, the response times are more predictable. A small number of potential outliers can disrupt the application. An explanation for the larger standard deviation is the higher number of hops from the *Field-* to the *RemoteEnvironment*, compared to the *FogEnvironment*, which is typically one or two hops away. As shown in Table 4.2, response times for services deployed in the *RemoteEnvironment* compared to the *FogEnvironment* are 2.17 times slower than AWS

[3]`https://aws.amazon.com/`

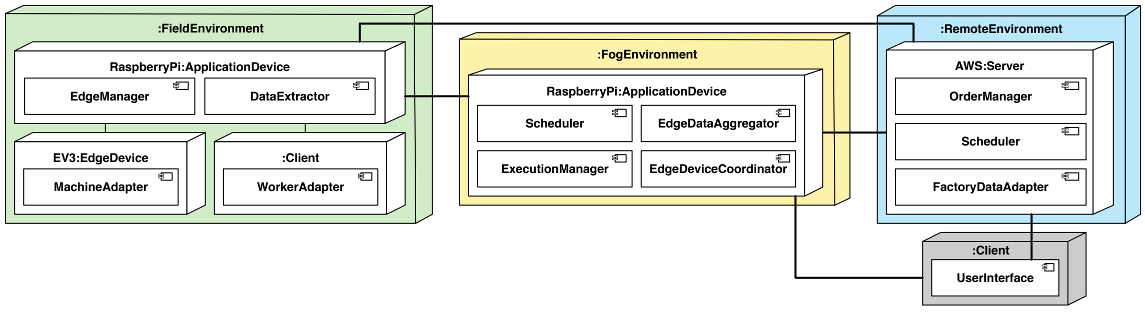

Figure 4.13: FEAt deployment diagram

EU-West deployments and 16.77 times worse than AWS US-West deployments. The measurements were conducted from Munich, Germany, and led to weaker results for AWS US-West-1a deployments because the data center is geographically further away. When considering the total loss of connection to the cloud, fog-based solutions can still provide services, which is not the case for remote ones.

	RTT mean	**RTT std. deviation**	**Downlink**	**Uplink**
Fog	11.06 ms	0.50 ms	94.1 MBps	94.2 MBps
EU-Central	24.08 ms	7.87 ms	46.5 MBps	9.47 MBps
EU-West	46.02 ms	8.89 ms	39.5 MBps	8.46 MBps
US-West	185.00 ms	5.12 ms	24.1 MBps	7.66 MBps

Table 4.2: FEAt test measurements

Chapter 5
APEP and RIAP

While writing and formulating Fogxy, we faced several challenges. Writing an architectural pattern is an iterative process and requires several iterations [WF12]. The first design attempts were vague and not clearly formulated and did not sufficiently describe the pattern. Other software engineers did not understand the pattern well enough and could not apply it to realize Fog Computing applications. Section 5.1 introduces APEP (Architectural Pattern Evaluation Process). It allows for an architectural pattern draft to be evaluated based on adaptations of sophisticated methods for architectural evaluation. Section 5.2 presents the RIAP (Review of Intermediate Architectural Patterns) method. To prove the feasibility and applicability of APEP and RIAP, Section 5.3 describes an application example where Fogxy has been iteratively evaluated.

5.1 Architectural Pattern Evaluation Process (APEP)

The goal of APEP is to support a pattern designer in evaluating and iteratively improving an architectural pattern and its formulation. Existing software evaluation methods are too sophisticated to evaluate a pattern, which is why we designed the lightweight and easy to use APEP process. APEP can be applied during the entire pattern writing process and supports different evaluation methods. APEP raises issues towards the design and formulation of the pattern. These issues facilitate the improvement of the pattern. Figure 5.1 shows the APEP process, which begins with choosing an evaluation method, then carries it out, and finally addresses the resulting issues.

5.1.1 Design

APEP requires two roles: designer and reviewer. The designer provides and introduces the architectural pattern. The designer is also responsible for answering questions that arise during the evaluation. APEP can be used for the early evaluation of patterns—called Intermediate Architectural Patterns—and the evaluation of mature patterns.

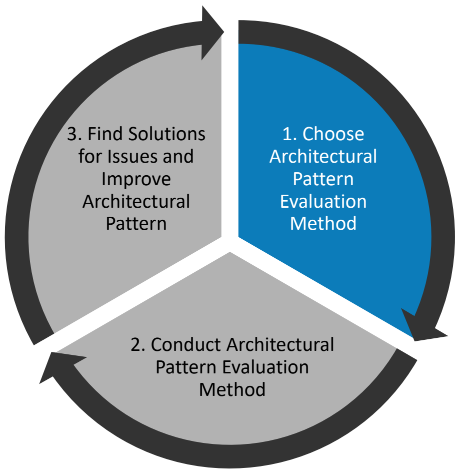
1. Choose Architectural Pattern Evaluation Method
2. Conduct Architectural Pattern Evaluation Method
3. Find Solutions for Issues and Improve Architectural Pattern

Figure 5.1: APEP process overview

Figure 5.2 presents an overview of the different steps and artifacts of APEP. The designer initiates the process by choosing an architectural pattern evaluation method. The reviewers perform the architectural pattern evaluation method and bring up issues. These issues are gathered in an issue list and described in more detail in Section 5.1.3. Once issues are raised, the designer can develop solutions and improve the pattern. If an evaluation method does not open up further issues, another method can be chosen and carried out.

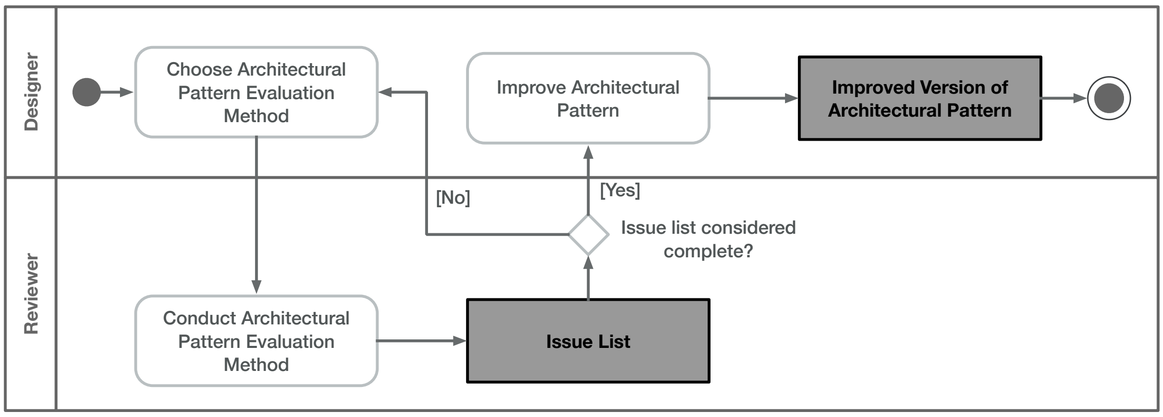
Designer
Reviewer
Choose Architectural Pattern Evaluation Method
Improve Architectural Pattern
Improved Version of Architectural Pattern
[No]
[Yes]
Issue list considered complete?
Conduct Architectural Pattern Evaluation Method
Issue List

Figure 5.2: APEP activity overview

5.1.2 Architectural Pattern Evaluation Methods

We took the methods for architectural evaluation presented in Section 2.2.3 and adapted them for architectural pattern evaluation. We present an adaption of ARID, namely RIAP, a Review for Intermediate Architectural Patterns. A proof of concept shows both the feasibility of an architecture and an architectural pattern through the implementation of a specific scenario. The architectural pattern evaluation method enables the reviewers to instantiate the pattern, elicit quality requirements, and assess the resulting architecture against those requirements. Figure 5.3 illustrates the sequence of activities for conducting the architectural pattern evaluation method.

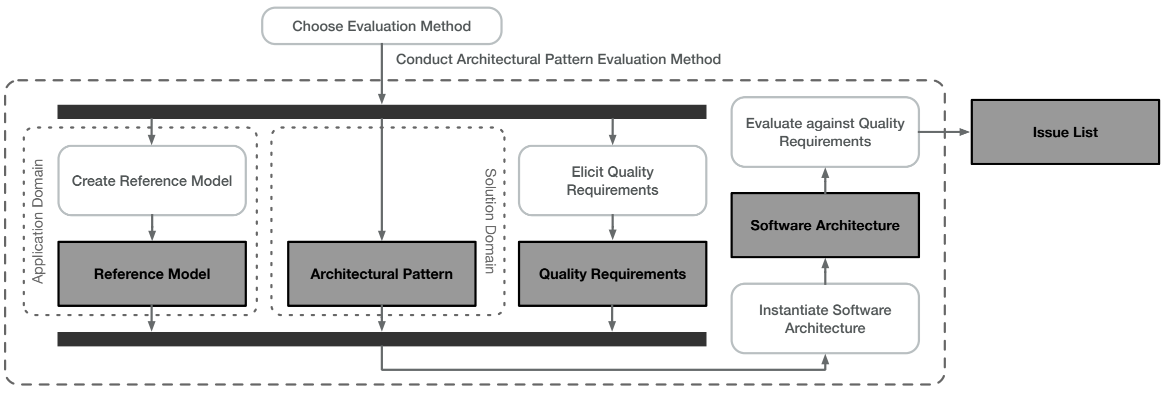

Figure 5.3: APEP conduction activity diagram

As illustrated in Figure 2.2, a software architecture is the result of a reference model and architectural patterns. The evaluation method therefore requires the reviewers to create a reference model. The designer provides the architectural pattern. In addition, quality requirements must be defined to assess the suitability of the architecture. Reviewers instantiate a software architecture using the artifacts, reference model, architectural pattern, and quality requirements. The evaluation of the architecture in terms of the quality requirements identified results in a list of issues containing all issues uncovered during the process and the evaluation.

5.1.3 Issues

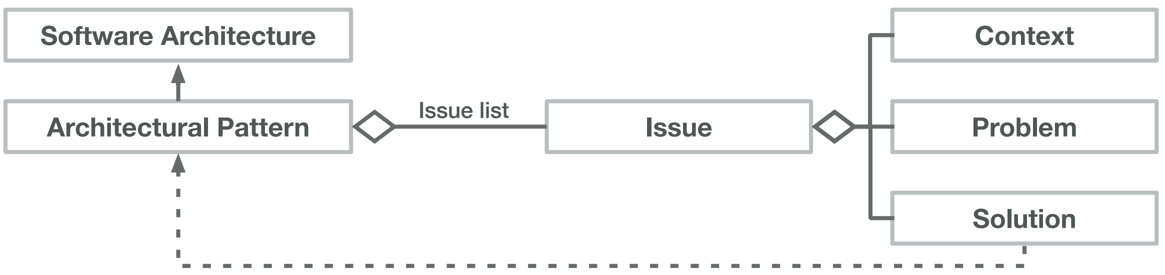

Figure 5.4: Issue list structure

All architectural pattern evaluation methods described reveal *Issues*. These issues arise in a *Context*, for example during the communication between two components of the architectural pattern. They feature a *Problematic Solution* and give rise to a *Refactored Solution*. The improved solution is applied to enhance the architectural pattern. Figure 5.4 visualizes the structure. This tripartite is used in the style of an anti pattern (cf. Figure 5.5) [BMMM98]. The *Context* is adopted directly. We merge *Problem* and *Problematic Solution* into the problem of the *Issue*, and the *Refactored Solution* transitions to our *Solution*.

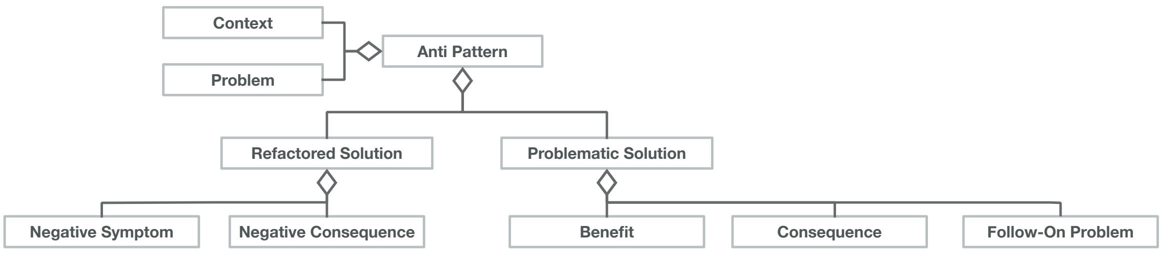

Figure 5.5: Anti pattern structure

5.2 Review for Intermediate Architectural Patterns (RIAP)

We present RIAP as an architectural pattern evaluation method for APEP. RIAP is an adaption of ARID (cf. Section 2.2.3) for architectural patterns. Like ARID, RIAP is capable of early evaluations of intermediate architectural patterns.

5.2.1 Design Goals

The goal of RIAP is to strengthen the evaluated pattern. RIAP is part of the APEP process and is used for the early evaluation of intermediate architectural patterns.

5.2.2 Characteristics

During the design of RIAP, we considered its applicability in practice. In terms of requirements, we distinguish between functional and non-functional characteristics: features that RIAP must fulfill (functional characteristics - CR) or a restriction it must obey (non-functional characteristics - NCR) to be able to carry out the specified objective of strengthening the pattern [BD09]. In consultation with other pattern authors and inspired by other methods of architecture evaluation methods, we identified the following functional and non-functional characteristics for RIAP:

(CR1) Identify Issues: RIAP must identify issues of the architectural pattern.
(CR2) Improve Pattern: The identified issues must be usable to improve the pattern regarding context, problem, or solution description. There may also be negative consequences that need to be considered.
(CR3) Instantiate Architecture: According to APEP, an evaluation method must

instantiate a concrete architecture. RIAP must instantiate an architecture that includes the architectural pattern that is being evaluated.
(CR4) Provide Guidance: In contrast to the underlying ARID method, RIAP targets intermediate architectural patterns instead of intermediate designs. RIAP must provide guidance during the review.
(NCR1) Time: RIAP must be able to be carried out in a reasonable time frame - we define reasonable in this context as less than four hours. In contrast to architecture evaluation methods, reviewers do not associate themselves with the pattern as much as with an architecture they would be using in a project.
(NCR2) Simplicity: Since RIAP targets intermediate architectural patterns, it must be feasible for undocumented patterns as well.
(NCR3) Heterogeneity of Participants: RIAP must be executable with a heterogeneous group of software engineers. Heterogeneous in this context means that they may not have experiences in the domain in which the pattern is applicable, and that the group may have different levels of experience.

5.2.3 Implementation

ARID serves as the basis for RIAP as it is lightweight, can be completed in a reasonable time frame and can also be used for intermediate architectural patterns. RIAP needs a facilitator, a designer, and reviewers. The designer provides the architectural pattern, introduces it, and answers questions that arise during the review. The role of the facilitator is to guide the process and capture issues. The reviewers' task is to test and review the architectural pattern. They do not need to have experience in the pattern's application domain. RIAP is applicable for a heterogeneous group of software engineers even though it expects general knowledge of software architecture and software engineering. The duration of RIAP is set at three hours to stay within a reasonable time frame. The process is divided into two phases: the rehearsal and the review. In the rehearsal phase, the designer and facilitator prepare the actual review, which takes place in the review phase.

Rehearsal

The rehearsal consists of four different steps. Figure 5.6 depicts those steps as a UML activity diagram. The facilitator and the designer meet and select the reviewers first. The designer prepares a pattern briefing that explains the architectural pattern and chooses a problem domain in which to evaluate the pattern. Since an architectural pattern is used in a broader context than a design, we assume that the scenarios that the evaluators would develop were too different to evaluate. We restrict all scenarios to a specified problem domain. The facilitator prepares the RIAP presentation, which includes the design briefing and the description of the problem domain. We recommend printing both the design briefing and the problem domain on a handout for the reviewers to use throughout the review.

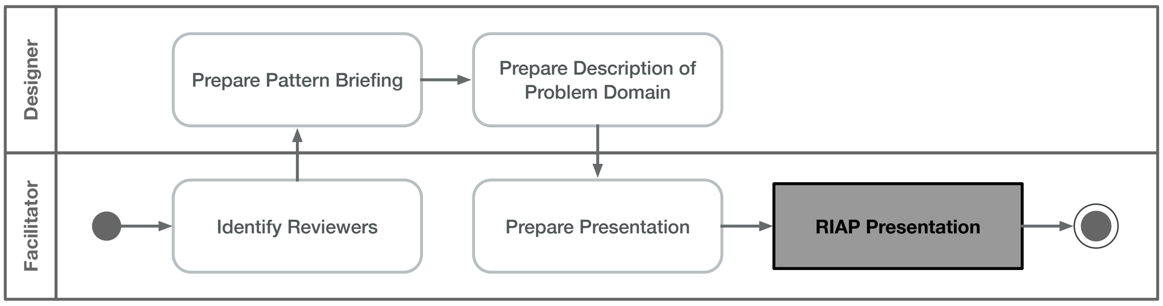

Figure 5.6: RIAP rehearsal phase activities

Review

While ARID targets a design, RIAP targets an architectural pattern: evaluating this seems to be too daunting of a task for the reviewers to tackle on their own. RIAP provides strict guidance through the process. In the review phase (cf. Figure 5.7), the facilitator introduces RIAP to the reviewers by describing all the steps. The designer presents the architectural pattern. During this time, reviewers can ask questions about the understanding of the architectural pattern. The facilitator records these questions as issues. Afterwards, the designer presents the problem domain to the reviewers.

To guide the reviewers in the following more active part, we set a total of seven tasks. The facilitator introduces each task. The reviewers complete the task, while the designer can answer questions. These questions target the understanding of the architectural pattern. The designer may not give their opinion on how to implement certain aspects of the system, as this would influence the reviewers' design decisions. The questions and any problems that arise during the completion of the tasks, such as how to map components of the pattern to the scenario, are recorded by the facilitator. The seven tasks are described in the following:

(T1) Brainstorming Scenarios: The goal is to generate scenarios in which the architectural pattern is applicable and which fit the proposed problem domain. We suggest using brainstorming techniques such as the gallery method[1] or brain sketching[2]. The reviewers write the scenario title on a card and give a brief description.

[1]https://www.mycoted.com/Gallery_method
[2]https://www.mycoted.com/BrainSketching

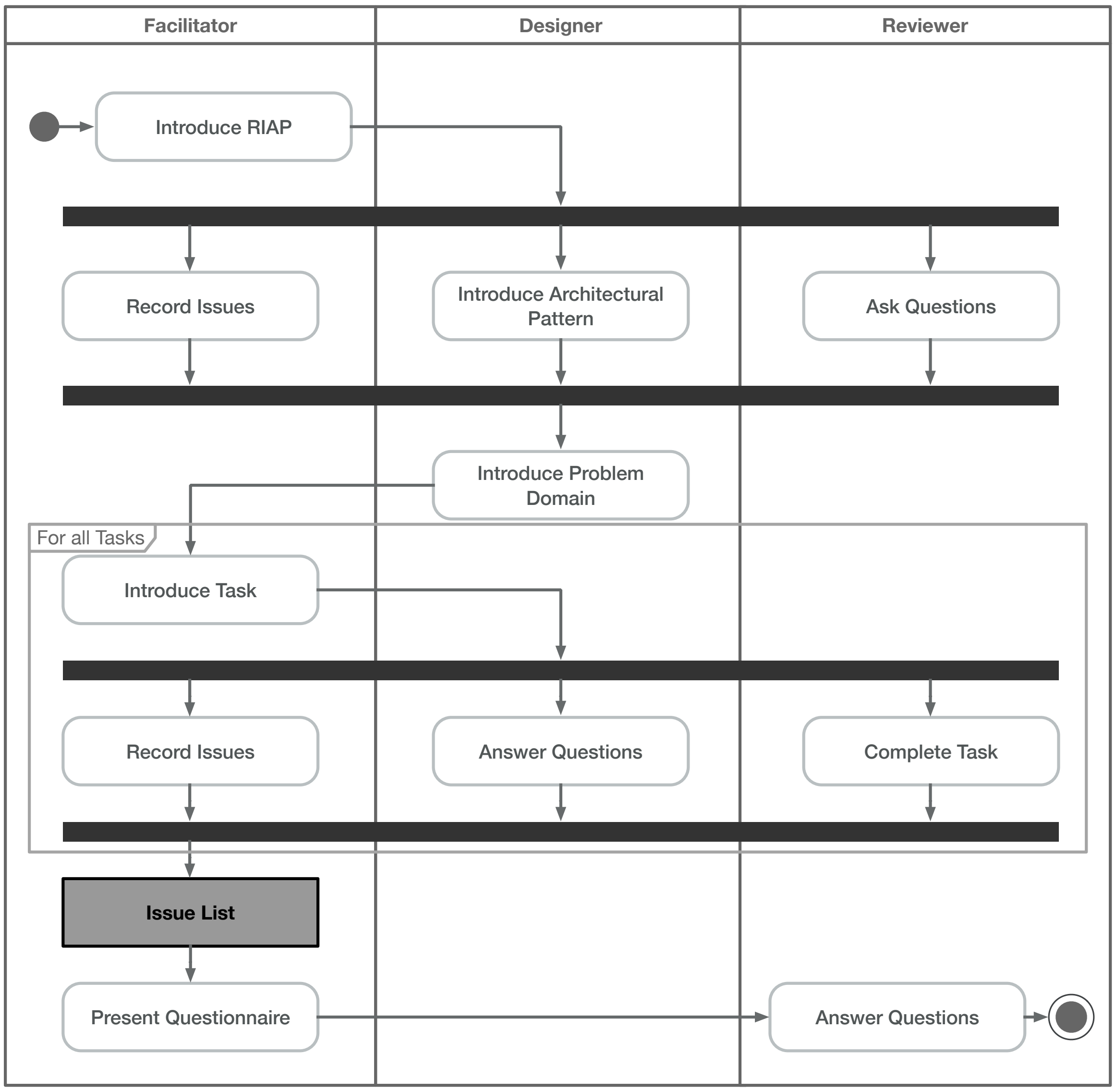

Figure 5.7: RIAP review phase activities

The cards are pinned to a white board and displayed visibly for all reviewers. The reviewers can ask questions if a scenario is unclear. The process of writing cards and looking at the results can be repeated until no new ideas emerge.

(T2) Prioritize Scenarios: Due to time constraints, it is not possible to implement every suggested scenario, meaning that they must be prioritized. Each reviewer has a total of three votes which they can assign to the scenarios. The participants attach stickers to the cards on the white board, as described in the sticking dots method[3]. The top-rated scenario is used for all subsequent tasks.

(T3) Formulate Demo Scenario: The scenario has a title and a brief description. To implement it, the reviewers formulate a detailed description of the flow of events. The implementation must be able to carry out all these steps.

[3]`https://www.mycoted.com/Sticking_Dots`

(T4) Elicit Non-Functional Requirements and Quality Requirements: The group elicits non-functional requirements that are important for the scenario and must be considered in the design to meet the quality requirements of the architecture.
(T5) Create Subsystem Decomposition: To design the system, the reviewers create a subsystem decomposition. All parts of the pattern must be included.
(T6) Establish Hardware/Software Mapping: To demonstrate that the subsystems provided can be mapped to hardware, the group creates a hardware/software mapping that shows the hardware and protocols used for communication between the different subsystems.
(T7) Apply the Demo Scenario: The group is supposed to show how the designed system carries out all individual steps of the demo scenario. For example, they can show the dynamic behavior of all participating subsystems. During this task, designer and facilitator can ask questions regarding the feasibility of the proposed solution. For example, they can examine how the system meets the non-functional requirements.

After all tasks have been completed, the facilitator creates a list containing all issues from the tasks and the initial presentation of the architectural pattern. To further encourage reviewers to provide feedback, RIAP concludes with a questionnaire. All questions are written in the style of Active Design Review questions and aim to get reviewers to question the architectural pattern in this scenario. The facilitator presents the questions about the architectural pattern, such as whether the pattern could meet all quality requirements or, more generally, what the purpose of the pattern is, and the reviewers answer them.

5.3 Application Example

To evaluate RIAP, we conducted two case studies in July 2017 based on the architectural pattern Fogxy. The results of RIAP were used to improve Fogxy. A total of eight reviewers participated in the first case study and six participated in the second case study. The first case study took three and a half hours while the second case study was reduced to three hours. The pattern had not yet been published at that time and was still in a raw version. The results of RIAP were used to improve Fogxy.

5.3.1 Materials

To prepare for the case study, the designer and the facilitator met to craft a presentation and a handout[4]. The presentation contained the following slides: introduction

[4]The presentation and handout of the case study are available at `https://github.com/andreasseitz/apep`

to RIAP and the steps involved in the process, design briefing for the architectural pattern Fogxy, problem domain, task description, and questionnaire. For the design briefing, we used an intermediate version of Fogxy that included quality requirements, object model, dynamic view of object interaction, and component diagram.

5.3.2 Reviewers and Environment

We distinguish two groups of reviewers: software engineers familiar with the concepts of Fog Computing and people with less knowledge of Fog Computing. The first group consisted of computer science students doing research in the field of Fog Computing. The second group consisted of software engineering doctoral students. Figure 5.8 shows the allocation of the 14 reviewers to the two groups.

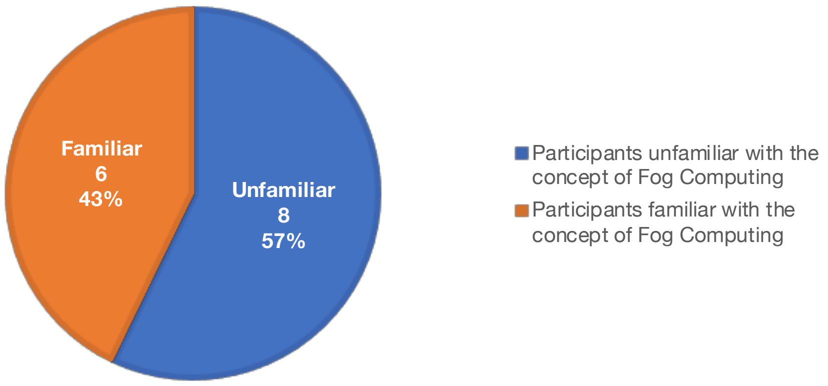

Figure 5.8: Prior knowledge distribution of the RIAP application example

The review took place in a meeting room, which offered magnetic boards for sketching and hanging cards. Figure 5.9 gives an impression of the working environment and the results.

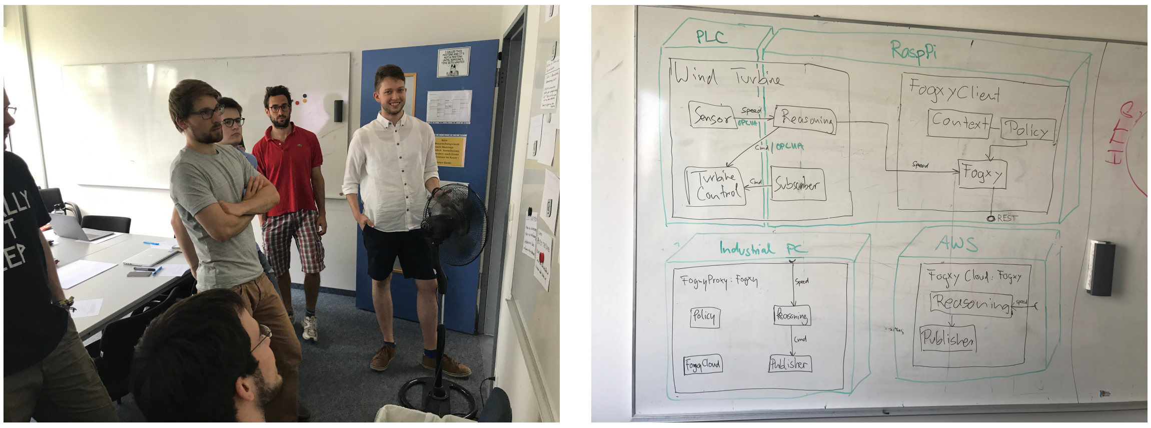

Figure 5.9: RIAP working environment and results

Problem Domain and Questionnaire

We used different problem domains for each case study. The first case study focused on the domain of renewable energies. A wind turbine must continuously adapt to the weather conditions to deliver as much electricity as possible and to protect it from damage. The adjustment is done by an actuator which either tilts the blades into the wind to increase the speed of the turbine or tilts them out of the wind to decrease it. A sensor is attached to the turbine to constantly measures the wind speed. The cloud provides a weather forecast which, in conjunction with the current speed, allows both a proactive and reactive response to wind speed changes.

The second case study is from the field of autonomous driving. A car drives autonomously due to attached sensors that evaluate the surrounding environment of the vehicle. The cloud provides additional information and acts as a central coordinator to reduce traffic congestion.

The questionnaire contained the following Active Design Review questions:

- What is the idea and purpose of Fogxy?
- What are the shortcomings of Fogxy in this scenario?
- Where did you have problems implementing Fogxy?
- Which non-functional requirements could not be fulfilled?

These questions allow participants to revise the architectural pattern and provide the opportunity to address further issues.

5.3.3 Evolution of the RIAP Method

In the first case study, it became clear that our decision to select a problem domain without scenarios was inconvenient. The tasks of brainstorming and prioritizing scenarios and determining non-functional requirements were not well received by the participants. These tasks were a means to an end and therefore took too long. Additionally, it was unclear to the reviewers how they were supposed to formulate scenarios independently of the architectural pattern being evaluated. We therefore decided to predetermine the concrete scenario and its non-functional requirements. We ensured that the architectural pattern was appropriate for the scenario and that there was no argument about it.

For the second case study, we described the following scenario: A car crashes and publishes this information. The system receives the information and coordinates the following vehicles to either brake or avoid a collision by steering away. The system must be available 99.9999 % of the time and the decision whether to brake or pass must be transferred to the following vehicles in less than 10 ms.

5.3.4 Findings

Table 5.1 gives an overview and compares the two case studies conducted. Based on the changes introduced to RIAP, we were able to reduce the time of the evaluation to three hours. In both case studies, two groups implemented one scenario. Each group came up with a different design. The first case study revealed seven problems, which were also confirmed in the second case study. The questionnaire did not reveal new insights in either session.

	Case Study 1	Case Study 2
Time	3h 30min	3h
# reviewers	8	6
# created scenarios	8	1
# implemented scenarios	1	1
# different designs	2	2
# issues	7	6 (all duplicates)

Table 5.1: RIAP application example metrics

5.3.5 Issue List

During the review phase of RIAP, the facilitator started to explain RIAP and gave a brief introduction to Fog Computing, and the designer introduced Fogxy. We encountered the first set of issues: a participant posed the question whether a smart textile would be an actuator or a sensor. Other participants added a light switch, which was not covered by Fogxy's object model. This was recorded as issue I1:

ID	**I1**
Name	**Client Taxonomy**
Context	Understanding Fogxy's object model
Problem	The taxonomy of the client is not clear regarding the classification of devices such as smart textiles or light switches.

For the participants, the policy object in the Fogxy object model was unclear. Its functionality and the relationship to the Fogxy object were not clearly described. We delineated this as issue I2:

ID	**I2**
Name	**Policy Object**
Context	Understanding Fogxy's object model
Problem	The purpose of the policy object and its relationship to the superclass Fogxy are unclear.

The communication between the client and the Fogxy node was unclear, as the opposite direction was covered through the publish/subscribe mechanism, as described with Issue I3:

ID	**I3**
Name	**Publish / Subscribe**
Context	Understanding Fogxy's object model
Problem	It is unclear how the client communicates with the Fogxy if it cannot be a subscriber itself.

Through discussion, it became clear that in some cases it is possible or necessary for the computation to be carried out by the client itself. This was reflected in Fogxy's object model, which led to the recording of issue I4:

ID	**I4**
Name	**Client-Computation**
Context	Understanding Fogxy's object model
Problem	The architectural pattern does not support the client carrying out simple computations by itself.

The discovery mechanism was the final issue with the object model. It was unclear to the participants how the client locates Fogxy nodes regardless of their deployment. This was formulated as issue I5:

ID	**I5**
Name	**Fogxy Discovery**
Context	Understanding Fogxy's object model
Problem	The discovery mechanism for Fogxy is not clear.

The Fogxy client includes the Fogxy object, the publisher interface, and the policy object, but they were perceived as abstract. The Fogxy object was a generalization of two remote systems and thus the reviewers were confused about why it is mapped to a client component. Issue I6 records this confusion:

ID	**I6**
Name	**Components**
Context	Understanding Fogxy's component model
Problem	The distinction between the client and the Fogxy client is unclear.

This confusion led to further ambiguities about the locality of the policy object, as described in issue I7:

ID	**I7**
Name	**Policy-Location**
Context	Understanding Fogxy's component model
Problem	The location of the policy object is unclear.

The groups then performed the RIAP tasks and created subsystem decompositions and hardware/software mappings for the different scenarios. Figure 5.9 shows the resulting hardware/software mapping of the first reviewer group.

Chapter 6
Seamless Computing

The Fogxy architectural style described in Chapter 4 is the prerequisite for Seamless Computing. Seamless Computing is a build- and release-management concept that enables the homogeneous distribution of Fogxy components to heterogeneous hardware nodes. Section 6.1 defines the terminology used for Seamless Computing. We define requirements in Section 6.2 and the reference model for Seamless Computing in Section 6.3. Section 6.4 investigates existing technologies from the field of Cloud Computing to determine whether they are suitable for the designed reference model. Section 6.5 present Fogernetes and DYSCO as concept implementations for Seamless Computing.

6.1 Terminology

Fog Computing applications consist of components distributed across different domains that have specific characteristics and provide particular software environments to manage the software lifecycle, such as methodologies, tools, and processes used to design, develop, build, test, deploy, run, and manage software. Figure 6.1 shows the domains and devices included. We present the properties and characteristics of domains and devices:

Field Domain. Traditionally, computing in the field domain is based on dedicated embedded hardware with limited resources regarding computing power and storage capabilities. In-field edge devices have direct access to physical sensors and actuators. They often perform under real-time constraints and are part of mission-critical processes or infrastructure, such as energy systems or industrial processes. Software on field devices uses low-level programming languages, without operating systems or with specific real-time operating systems [BSS+18, CSB17]. The deployment and update of software require service technicians to go on-site, leading to long, inefficient and error-prone software lifecycles. With increasing computing and storage capabilities, it is possible to establish field devices with standard computer environments, such as

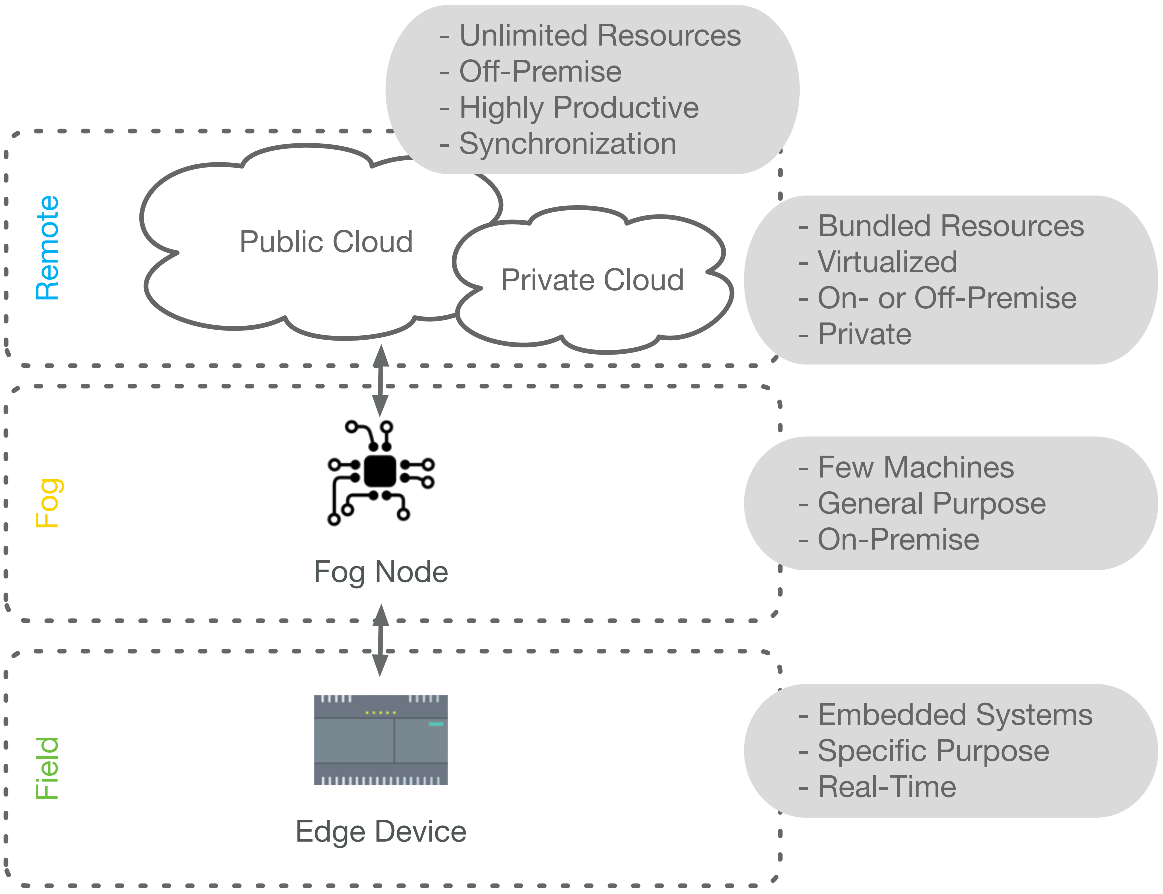

Figure 6.1: Seamless Computing domains of industrial systems (adapted from [MGSB17])

operating systems and higher programming languages.

Fog Domain. Computing devices in the fog domain typically consist of general-purpose hardware and standard operating systems. Fog nodes for example are located at production lines of an industrial plant, or in a substation of an electrical grid. The fog domain within a site consists of a single computer up to a smaller number of nodes, ranging from industry PCs to workstations or servers. Due to its logical and geographical proximity to physical machines, the fog domain is characterized by high-security requirements and low latency times for interactions with the field devices.

Remote Domain. The remote domain is the most abstract and standardized computing environment. Cloud providers offer computation and storage capacities on-demand, enabling elasticity and scalability of applications deployed in that domain. Cloud Computing has led to the emergence of service models such as Platform as a Service (PaaS), cloud orchestration, and continuous delivery, which accelerate the software lifecycle, transforming deployment frequency from every few months to several times a day [Hum17]. Public cloud services are centralized and offered by a few large providers such as Amazon, Microsoft, and Google. The geographical distance between field devices and the central remote components results in high latency and limits real-time applications. General purpose servers are used in the remote domain and for industrial scenarios. It is common for medium-sized or large companies to run IT applications in such data centers. While they were initially based on physical servers, virtualization techniques enable application requirements to be flexibly mapped to existing server capacities.

The different software environments of the compute domains lead to a static assignment of system components and limit the distribution across layers. Seamless Computing addresses these issues and implements similar, ideally identical software environments across compute domains. It provides consistent tools and technologies for designing, developing, testing, deploying, and running software to support multi-domain applications. While the application of this homogeneous environment to field devices with microcontrollers and non-standardized operating systems, or no operating systems at all, is initially not feasible, field devices are expected to be penetrated by generic hardware and operating systems. Seamless Computing can also be used for mobile devices. However, the prerequisites for mobile devices are different, as they usually possess standard hardware, operating systems, and varying network connectivity. As a basis for the relocation of components, we assume modular applications consisting of workloads, based on the architectural model of micro-services [New15]. We define *Workload Mobility* as the ability to move application components within and across compute domains, using the same code, application lifecycle tools, and deployment artifacts. Figure 6.2 shows two scenarios of an application deployment configuration. Scenario 1 visualizes the distribution of application components across the three compute domains *Field*, *Fog*, and *Remote*. Scenario 2 shows the allocation of the same components in the two lower domains. Both scenarios present the deployment of the same application in different configurations. Workload mobility has two different characteristics: *Static* and *Dynamic Workload Mobility*. *Static Workload Mobility* allocates components at deploy time and allows component requirements to be mapped to characteristics of the compute domains. *Dynamic Workload Mobility* allows the relocation of components at run time and enables to react to the dynamic behavior of the compute domains and their application components. Dynamic reconfiguration based on utilization or erroneous states leads to improvements, such as higher availability or better performance.

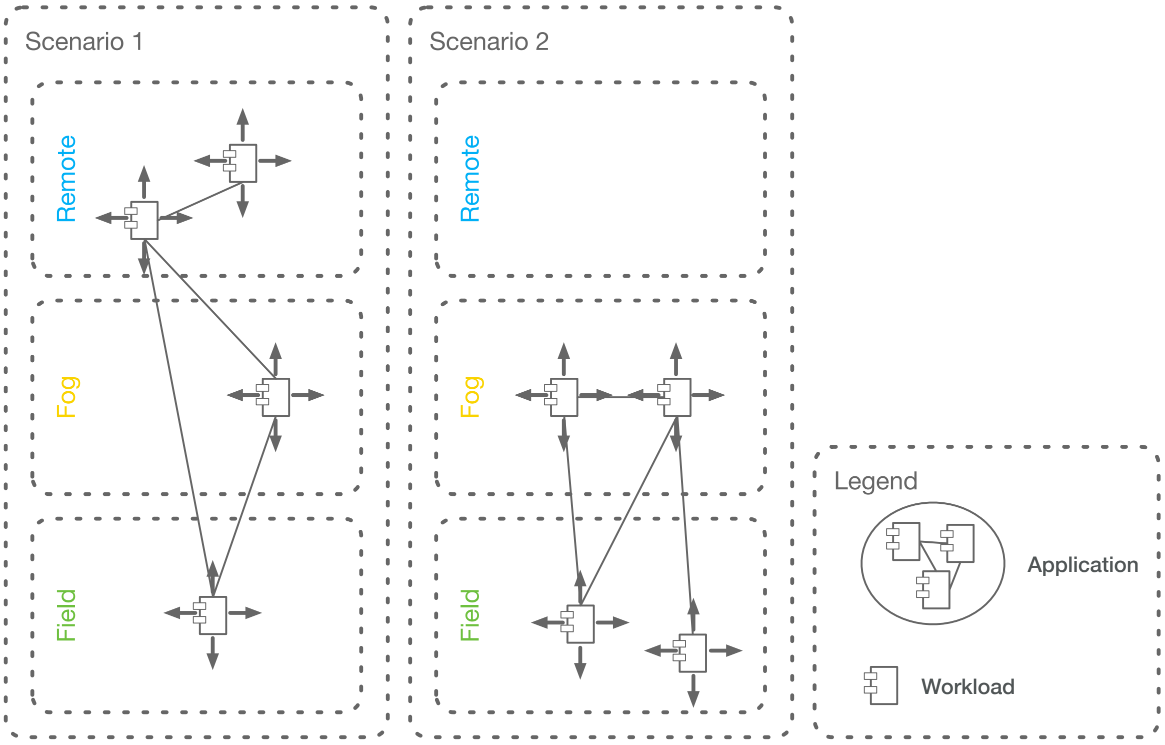

Figure 6.2: Workload Mobility in Seamless Computing

6.2 Requirements

In cooperation with partners from the manufacturing industry, we elicited the following requirements for Seamless Computing:

(1) Provide a homogeneous software environment: The same methodologies and tools for implementing, testing, deploying, running, and managing software need to be available across compute domains.
(2) Static workload mobility: Provide the ability to allocate application components to node instances within and across the compute domains. This process must be automated.
(3) Dynamic workload mobility: Schedule application components at runtime, according to the status of the nodes in the system, and according to defined rules or, ideally, optimization criteria. This includes rescheduling in the event of performance degradation or other system changes.
(4) Stay close to standards: The aim is to use standard and widespread technologies, many of which are available or emerging in the field of Cloud Computing.
(5) Support physical device connectivity: The system must be able to deal with connecting devices, such as sensors and actuators, often using legacy interfaces and protocols. It should be possible to attach new devices to the platform without the need for configuration by a human operator.

(6) Support real-time applications: Provide the capability to describe and map real-time characteristics of applications and system components. This must be supported in the orchestration and scheduling mechanisms and in the runtime environment.
(7) Small footprint: The Seamless Computing environment must run on nodes with few resources.
(8) Security: Ensure data privacy and provide mechanisms to secure deployment, operation, and management of the application components running in the Seamless Computing environment against vulnerabilities.

6.3 Reference Model

The reference model supporting Fog Computing applications enables different functions of the platform to be identified and establishes relationships among them. Figure 6.3 shows the reference model in an architectural stack. The functions derive from the requirements and consider existing software stacks typically used in remote domain platforms or orchestration tools. To apply Seamless Computing across all domains, the functions in the reference model must be present in all domains, which requires a concept for functional distribution. In the following, we describe the different building blocks of the reference model:

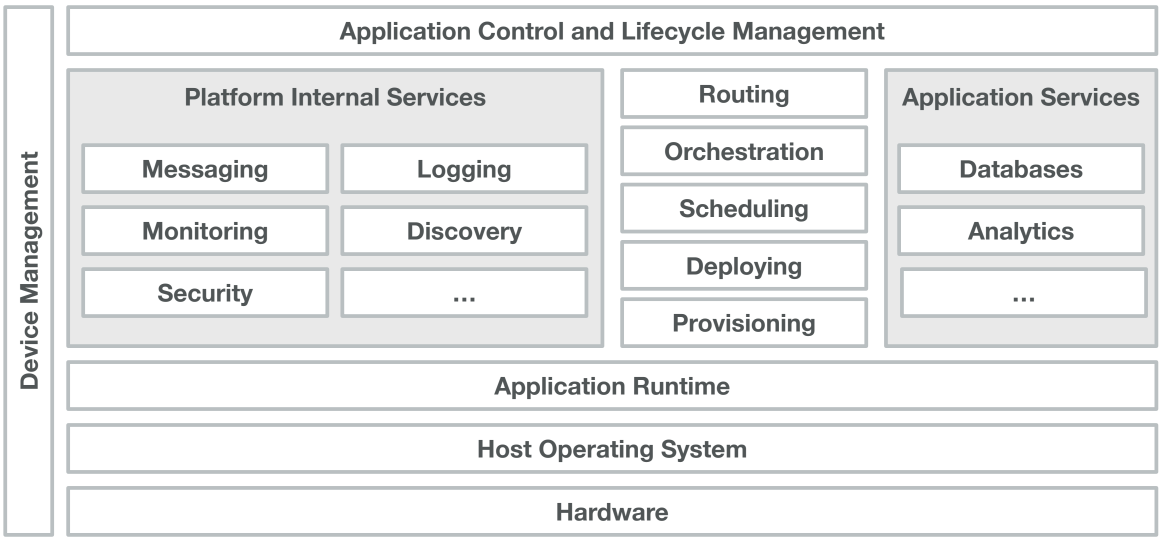

Figure 6.3: Seamless Computing reference model

Hardware denotes the physical infrastructure that hosts the applications and ranges from enterprise-grade servers to field devices supporting different hardware architectures. The *Host Operating System* bridges the hardware and application runtime. Its purpose is to host the environment on which the applications are running. A minimal, single-purpose OS is preferred. The host OS needs to support the underlying hardware (e.g., x86 and ARM-based architectures). The *Application Runtime* executes applications in an isolated environment, including libraries and dependencies. Most popular solutions are based on virtualization or container technologies. *Provisioning* creates the virtual environment necessary to host the applications. This includes the setup of virtual machines, volumes, networks, or firewalls. *Deploying* and bootstrapping manages the download of the application to the virtual environment. This includes the resolution of dependencies and makes them available for the application. With *Scheduling*, Seamless Computing enables the efficient allocation of available resources to realize real-time requirements. *Orchestration* enables automated management and coordination of applications based on a predefined workflow. It includes autonomic aspects and thus enables self-managed systems. *Routing* forwards incoming requests to the correct recipient, i.e., a specific instance of an application component. Additional features, such as load-balancing or service discovery, offer scalability, resiliency, and decoupling. *Platform Internal Services* represent the services used to manage applications. These include, but are not limited to, *Messaging*, *Monitoring*, *Security*, *Logging*, and *Discovery*. *Application Services* are services that are hosted and managed by the platform and provided to the applications. They comprise functions that are shared by applications such as *Databases* and *Analytics*. *Device Management* encompasses several functionalities, the most important ones being device registry and discovery, device access management, device configuration capabilities, device updates (firmware and applications), and device monitoring. *Device Management* allows hardware to be added in any layer of the system and made immediately available to the applications. This functionality must be supported at different levels of the functional model and is therefore presented as a cross-section task. The *Application Control and Lifecycle Management* integrates the application development process with the production and release processes, including control aspects in production, such as restarting the application and scaling it.

6.4 Gap Fit Analysis

The gap fit analysis allows us to examine existing technologies from the area of Cloud Computing and IoT and compare them to the Seamless Computing requirements described in Section 6.2 along with the reference model described in the previous section.

The goal is to use existing technology to establish the Seamless Computing platform and thus enable the management of multi-domain applications. Table 6.1 summarizes the results of the analysis. Cloud Foundry[1] fulfills most of the requirements. However, its large footprint makes it unsuitable for the fog or field domain. We combine lightweight technologies such as containers (Docker) and container orchestration (Kubernetes) to realize Seamless Computing. The Fogernetes and DYSCO application examples consider these findings and represent implementations for Static and Dynamic Workload Mobility in the following section. In [MGSB17], we discuss in detail different technologies and map them to the reference model.

	Host Operating System	Application Runtime	Provisioning	Deploying	Scheduling	Orchestration	Routing	App. Control and Lifecycle Management	Device Management	Standards	Support Heterogeneity	Physical Device Connectivity	Real-Time Capability	Small Footprint
Unikernel	-	+	+	-	-	-	-	-	-	o	o	o	-	+
Docker	-	+	+	-	-	-	-	-	-	+	o	+	-	+
OpenStack	+	o	+	o	+	o	o	-	-	+	+	-	+	-
CloudFoundry	+	+	+	+	+	+	+	+	-	+	o	-	-	-
Brooklyn	-	-	+	+	-	+	-	o	-	+	+	-	-	o
Kubernetes	-	-	-	+	+	+	+	-	-	+	+	-	-	+
Mesos	-	-	-	-	+	-	-	-	-	+	+	-	-	-
Marathon	-	-	-	+	+	+	+	-	-	+	+	-	-	-
hawkBit	-	-	-	-	-	-	-	-	+	o	+	+	-	+

Table 6.1: Seamless Computing technology gap fit analysis

6.5 Application Examples

This section presents two concept implementations of the Seamless Computing reference model. Fogernetes in Section 6.5.1 enables Static Workload Mobility, and DYSCO in Section 6.5.2 investigates dynamic scheduling to realize Dynamic Workload Mobility.

[1]`https://www.cloudfoundry.org/`

6.5.1 Fogernetes

Fogernetes is a platform to manage and deploy Fog Computing applications. It allows component requirements to be mapped to heterogeneous device capabilities by establishing a labeling system. The application Fodeo serves as a test application that uses Fogernetes to present its practical applicability for the deployment and management of Fog Computing applications. Fodeo analyzes video streams from multiple cameras and optimizes the video delivery. Solutions for deploying and managing Fog Computing applications are still in their infancy. While there are technologies tailored to orchestrate Cloud Computing applications, Fog Computing lacks such technologies. The characteristics of Fog Computing, such as distribution, availability, heterogeneity, and real-time pose challenges for application deployment and management platforms. A sophisticated approach is required to deploy and orchestrate fog applications. Fogernetes overcomes these challenges by comparing and mapping requirements of application components to available nodes and ensuring optimal deployment to meet non-functional requirements.

Analysis

Fogernetes is based on Kubernetes[2]. Its configuration files and scripts are open-source and available on GitHub[3]. Fog nodes are hosts in a Fog Computing environment. These hosts interact and communicate with each other in a distributed manner for optimized performance. Components are distributed on different nodes in the environment. The assignment of components to nodes is configured individually depending on the use case. A node runs either in the remote, fog, or field domain. Field devices, fog nodes, and dedicated cloud servers are classified as nodes. We define deployment mapping as a process in which these requirements are mapped to existing capabilities and a deployment is subsequently performed. To specify this mapping process, Fogernetes addresses the requirements we describe in the following. In contrast to cloud applications, fog applications run on heterogeneous nodes, such as microcontrollers, smartphones, or servers. Fogernetes must be able to support these different types of devices (FR1). An advantage of Fog Computing is the increased availability of systems. This is achieved by redundant deployment of components. Fogernetes must also manage system failures through node redundancy (FR2). To be able to create a match between component requirements and node capabilities, both the requirements and the capabilities must be labeled (FR3). A suitable deployment can be guaranteed if the capabilities match the requirements. Furthermore, Fogernetes must

[2]`https://kubernetes.io`
[3]`https://github.com/ls1intum/fogernetes`

be able to handle different computing layers, namely remote, fog, and field (FR4). Deployment to different layers is essential to meet the NFRs of the applications to be deployed [BBL01]. Additionally, deployment settings, required capabilities, and setup configurations must be traceable. Applications can be rolled back to previous versions in case of a failure (FR5). For successful management of Fog Computing applications, all available nodes in the network must be traceable and monitored (FR6). Fogernetes must provide and take into account this information, thereby enabling the performance evaluation and optimization of the deployment mapping of fog applications and nodes [YHQL15]. In addition to the functional requirements, Fogernetes must also meet non-functional requirements. Fog Computing applications must be easily deployable. New application releases can be deployed with a minimum number of clicks in a short time (NFR1). Nodes are chosen based on the application's requirements (NFR2). Developers should not interfere in this process, as it ensures the best possible deployment mapping. Developers choose the technology stack of the Fog Computing applications because application development should happen without deployment constraints. The deployment and management tooling should not limit developers in their choice of tools or programming languages (NFR3). The platform needs to support at least 1000 nodes (NFR4) [BMZA12].

Requirements and Capability Mapping

In contrast to Cloud Computing, the capabilities of servers and hardware nodes cannot be dynamically scaled and adjusted in Fog Computing. Traditionally, a server is selected in a data center that meets the requirements. Within Fog Computing, existing hardware is used and can rarely be changed. An application component may require more memory to operate properly, and another component may need to be in a specific location to provide a low latency service. The challenge is to map requirements to the heterogeneous node capabilities. Fogernetes uses a labeling system to realize the mapping process. Applications and nodes must be described according to the capabilities that they require or offer. Labels describe the requirements of components and characteristics of available hardware nodes. Labels are assigned either automatically, based on hardware capabilities or manually. Labels are the foundation for the deployment mapping of Fogernetes. During application deployment, Fogernetes verifies the application's requirement labels and compares them to the capability labels of available nodes. This allows Fogernetes to distribute application components across the environments and meet the requirements. Fogernetes defines the following four label categories (C) to establish a mapping between requirements and capabilities:

Location (C1). Fog components must be geographically distributed. Location labels are particularly important for the field layer and enable the placement of fog components on dedicated fog nodes.
Device Extension (C2). Fog nodes could have custom extensions, such as sensors to record measurements.
Performance (C3). We define performance labels to describe the usage of resources. This ensures that components can be used according to their performance requirements and that the necessary computing resources are in place.
Connectivity (C4). Connectivity is important for Fog Computing applications. For sensors with low data rates, a dial-up connection is sufficient; for larger monitoring systems, it must be ensured that sufficient bandwidth is available.

```
fogernetes/<category>.<selector>: <value>
```

Listing 6.1: Fogernetes labeling system specification

Listing 6.1 shows the naming scheme for Fogernetes that enables the mapping between requirements and capabilities. A fog node at the Technical University of Munich, for example, has the Fogernetes label `fogernetes/location: germany.munich.tum` to encode location information.

Realization

This section presents Fodeo, a custom fog application we created for testing and evaluating Fogernetes. We deployed and managed Fodeo using the platform and compare the results to the requirements to verify whether Fogernetes is capable of deploying and managing Fog Computing applications. Fogernetes extends Kubernetes to meet the requirements for Fog Computing applications by the following three extensions:

Application Registry. A private Docker image registry[4] is used as part of the Kubernetes ecosystem. This Kubernetes add-on offers the possibility of storing private Docker images. Each application component is packaged in a Docker image that can be used on different hardware if supported by the container. An additional advantage is that dependencies are always available in the correct version. The Docker tagging function enables the deployment of older application versions or the rollback of changes.

Kubernetes Deployment. Two objects in the Kubernetes ecosystem are relevant, the Pod and the Deployment Controller. "A Pod is the basic building block

[4] `https://github.com/kubernetes/kubernetes/tree/master/cluster/addons/registry`

of Kubernetes—the smallest and simplest unit in the Kubernetes object model that you create or deploy. A Pod represents a running process on your cluster"[5]. The Deployment Controller organizes the Pods and Replica Sets[6]. It automatically adjusts the number of Pods in a controlled manner to ensure availability and performance. The deployment definition is stored in a file. Based on the formalized approach, we use the `nodeSelector` configuration from the Kubernetes Pod documentation. A Kubernetes cluster uses DNS as a service registry and enables distributed components over different layers to discover each other. A new service automatically creates an entry in the DNS server. For example, the DNS record *nginx.fog* describes the service *nginx* in the namespace *fog*. This allows Pods that are part of the namespace *fog* to identify the cluster IPs of the Pods running this service.

Management. To add a new node, it is necessary to install Kubernetes on the node and add it to the existing cluster. Fogernetes uses Heapster as a monitoring tool, together with InfluxDB and Grafana[7], enabling the creation of custom dashboards. The dashboard provides a user interface for administrators to identify problems.

For the realization and testing of the three extensions, we used the Fog Computing application Fodeo. Fodeo is a surveillance application that delivers and analyzes videos from multiple cameras. The video signal is recorded and analyzed in a remote component. Fodeo applies Fogxy to improve video delivery performance and saves bandwidth by filtering incoming data and sending compressed video data to the cloud. Figure 6.4 shows the four Fodeo components and their relations.

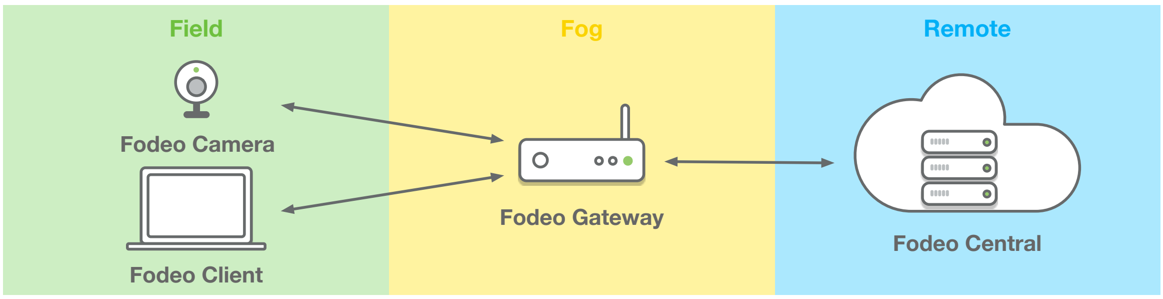

Figure 6.4: Fodeo top-level design

The *Fodeo Camera* component continuously takes pictures of its surroundings and sends these to the nearest *Fodeo Gateway* component. The *Fodeo Camera* component is placed in the field layer. It aggregates the received images from camera components

[5]https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/
[6]https://kubernetes.io/docs/concepts/workloads/controllers/deployment/
[7]https://grafana.com/

and converts them to optimized video files. It is placed in the fog layer since it accumulates multiple data streams from the field layer and partially forwards them to the *Fodeo Central* component to save bandwidth. The *Fodeo Central* component is placed in the remote layer to make video files accessible at a central location. The *Fodeo Client* component is placed in the field layer. It accesses the raw videos on a *Fodeo Gateway* component to ensure that the transfers are working correctly.

Fodeo Deployment

To deploy Fodeo, the following steps are required: (1) setup of the Fogernetes platform, (2) description of the Fodeo components including the requirements necessary for deployment mapping, and (3) deployment of the components. (4) Application components automatically discover each other's interfaces and (5) the infrastructure has to be managed and monitored. We explain the steps in detail and show the technical implementation:

(1) Fogernetes Platform Setup. Three different components are deployed on the Fogernetes platform. The *Fodeo Client* runs locally on a development laptop. It enables access to data in the fog layer and serves as a test device, such as a laptop, connected to the internal network. To implement Fogernetes, we create a Kubernetes cluster including a master node using the *kubeadm init* command. The corresponding *kubeadm join* command is executed on each of the three nodes. Two nodes are servers which host the remote and fog application components. One node is a Raspberry Pi running the *Fodeo Camera* component. The labeling system defines the node capabilities. Table 6.2 shows the different nodes and their assigned capabilities. Generic classifiers such as small, medium, or large define capabilities to match components in broad categories.

(2) Artifact Creation. A Dockerfile defines the required libraries and component requirements. After creating the artifact, we can push it to the application registry. Therefore, the component artifacts are available to the entire cluster and are ready for deployment.

(3) Component and Service Definition. Fogernetes deploys the three components *Camera*, *Gateway*, and *Central.* The component definition declares how many component replicas run in the fog layer, which artifacts to use from the application registry, and how to name the different components. The service definition declares

Node	Capabilities
Remote Node	• fogernetes/layer: remote • fogernetes/performance.storage: large • fogernetes/performance.cpu: medium • fogernetes/performance.memory: medium • fogernetes/connectivity: fiber **Hardware**: ESXi VM Server
Fog Node	• fogernetes/layer: fog • fogernetes/performance.storage: small • fogernetes/performance.cpu: large • fogernetes/performance.memory: medium • fogernetes/connectivity: dsl **Hardware**: ESXi VM Server
Field Node	• fogernetes/layer: field • fogernetes/extension.media.camera: true • fogernetes/performance.storage: small • fogernetes/performance.cpu: small • fogernetes/performance.memory: small • fogernetes/connectivity: wifi **Hardware**: Raspberry Pi

Table 6.2: Fogernetes labels describing node capabilities

open ports and remote access. For testing purposes, we use the *NodePort* type[8] that allows us to access the service via a direct IP address and a specific port. Based on the labeling system, we define requirements for these components. Table 6.3 shows the requirement descriptions of the three different components.

(4) Deployment. After creating the artifacts, defining the components, establishing services, and adapting the application to discover available services, we can deploy Fodeo. Components and services can be deployed with the Fogernetes platform using two commands. The deployment mapping is performed automatically. The components and their corresponding services are provided on the cluster. Fodeo's requirements are mapped to the capabilities associated with each node. Figure 6.5 shows the mapping and gives a top-level view of the deployed application. Using the Kubernetes labels and selectors, we assign labels to both the nodes and the application components.[9] Therefore, the mapping is performed automatically by finding exact matches between the label and the selectors.

[8]`https://kubernetes.io/docs/concepts/services-networking/service/`
[9]`https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/`

Component	Requirements
Fodeo Central	• fogernetes/layer: remote • fogernetes/connectivity: fiber • fogernetes/performance.storage: large
Fodeo Gateway	• fogernetes/layer: fog • fogernetes/connectivity: dsl • fogernetes/performance.storage: large
Fodeo Camera	• fogernetes/layer: field • fogernetes/connectivity: wifi • fogernetes/extension.media.camera: true

Table 6.3: Fogernetes labels describing component requirements

(5) Platform Management and Monitoring. The Kubernetes Dashboard allows us to manage deployments. We can create new deployments or delete existing ones. In addition, existing deployments can be scaled by increasing the number of Pods running on the fog nodes. The Grafana dashboards enable the visualization of Fodeo's performance.

Results and Discussion

We discuss the results of the Fodeo deployment using Fogernetes and the contribution of features.

Fogernetes supports artifact creation, component definition, service discovery, deployment, management, and monitoring. We were able to deploy Fodeo's components on different nodes. The *Fodeo Camera* component was deployed onto a Raspberry Pi in the field layer. Both the *Fodeo Gateway* and the *Fodeo Central* components were deployed onto ESXi virtual machines. These nodes were placed in different locations and environments. Fogernetes and its labeling system allow node capabilities to be matched to component requirements, thereby creating suitable Fog Computing deployments. Kubernetes, with its support for many nodes (cf. NFR4), represents a viable candidate for the deployment and management of Fog Computing applications. Due to its extensibility, missing features can be added, and additional devices are supported (cf. FR1). The mapping between requirements of a Fog Computing application and capabilities of fog nodes is a major requirement (cf. FR3). Fogernetes meets this requirement with its labeling system and enables the creation of mappings (cf. NFR2). Besides this, Kubernetes takes care of node failures and ensures a working deployment (cf. FR 2). Kubernetes is intended for applications that are comprised of different components or split into different services. It is easier to scale components

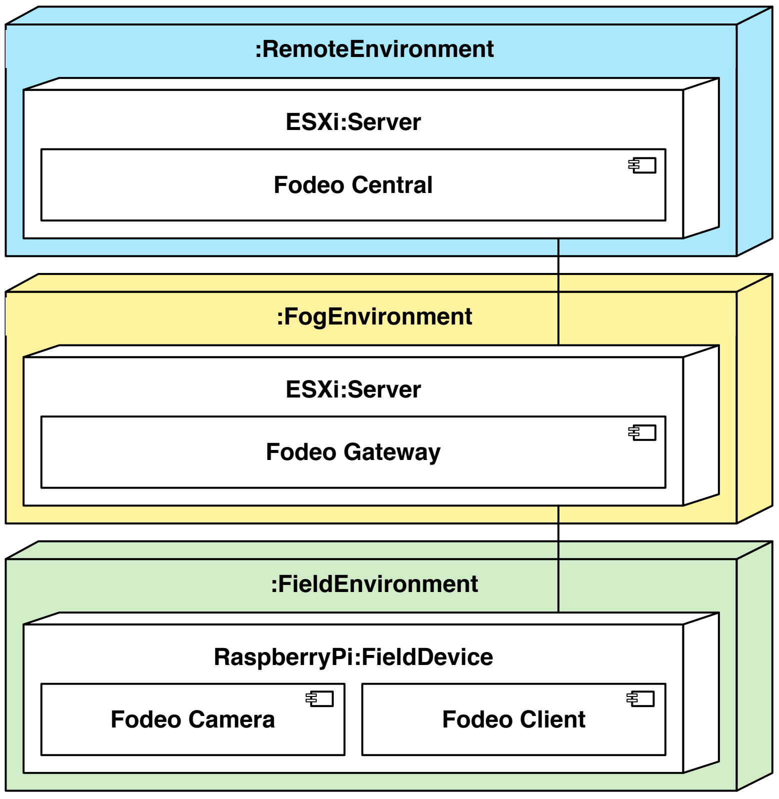

Figure 6.5: Fodeo deployment diagram

without affecting the rest of the system. This is suitable for the distributed nature of Fog Computing and its different layers (cf. FR4). The micro-service architecture enables scalability and facilitates deployment changes. Therefore, Fogernetes automatically adapts to failing nodes and ensures that fog components are recreated and deployed on suitable nodes.

Deployment is possible with a single command (cf. NFR1) and components run independently with Docker containers. Fogernetes enables developers to deploy Fog Computing applications without complex procedures. The integrated versioning of the application registry allows the roll back of changes or the simultaneous execution of different application versions (cf. FR5). The different dashboards integrated in Fogernetes support the node management (cf. FR6). While Fogernetes meets the requirements of a Fog Computing deployment and management platform, it has limitations. Fogernetes was executed inside an isolated network and interference or weak network bandwidth might affect the performance and functionality.

6.5.2 DYSCO

Production systems require availability, fault tolerance, and extensibility. This section presents DYSCO (Dynamic Scheduling for Seamless Computing), a system for the dynamic rescheduling of software components in a distributed, heterogeneous Fog Computing cluster at runtime. DYSCO extends Kubernetes through the addition of a monitoring tool and enhances the scheduler to enable dynamic component rescheduling. We verify the requirements of DYSCO using test cases for an industry-specific scenario in a Fog Computing cluster. It operates a safety-critical application that must immediately react to machine failures (emergency) and an application that processes employee data for analytics (timestamp). DYSCO reschedules software components at runtime while ensuring technology independence, availability, fault tolerance, and usability. DYSCO represents a realization of Seamless Computing and deals with the mobility of IoT devices by scheduling applications between different nodes at runtime.

Requirements

The requirements of DYSCO partially overlap with the requirements of Fogernetes as they both represent implementations of Seamless Computing. For DYSCO, the requirements for scheduling at runtime (FR9) and the optimal resource utilization (NFR4) are of particular interest to realize Dynamic Workload Mobility.

DYSCO has to meet the following functional requirements: (FR1) DYSCO must support heterogeneous nodes with different capabilities such as CPU architecture, memory, and network connectivity. (FR2) The deployment and scheduling of applications are compute domain independent. (FR3) DYSCO must be able to utilize remote nodes from multiple cloud service providers. (FR4) The components communicate over defined interfaces and can be distributed across compute domains. (FR5) DYSCO enables the specification of capabilities for heterogeneous nodes (FR6) and the specification of component requirements. (FR7) DYSCO supports the scheduling of stateless applications (FR8) and stateful applications. (FR9) Using the node capabilities and component requirements, DYSCO schedules applications dynamically at both deploy- and runtime. (FR10) DYSCO monitors the active nodes and applications in the cluster to detect changes within the cluster, such as node failures, topology changes, connectivity issues, or resource scarcity.

In addition to the functional requirements, DYSCO must meet the following non-functional requirements: (NFR1) Developers can add application components with a single deployment script and a single command. Developers can create applications with any technology. (NFR2) They may freely choose the programming languages, frameworks, and tools. (NFR3) DYSCO automatically restores a working state in

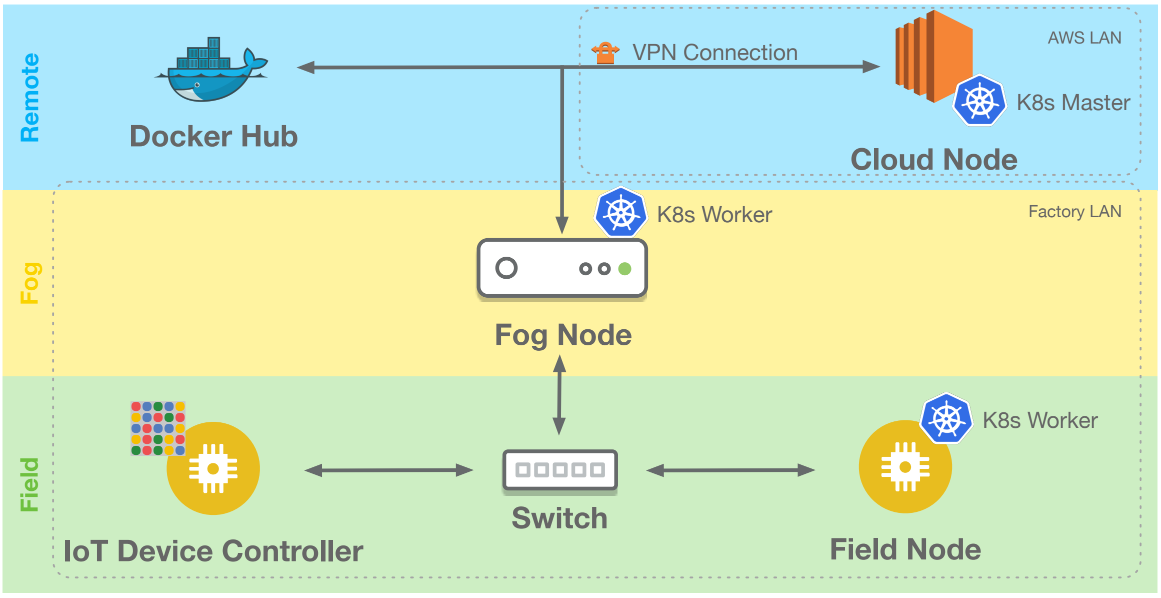

Figure 6.6: DYSCO test case design

the event of a node or network failure. DYSCO continuously optimizes the mapping between component requirements and node capabilities and tries to best match requirements to capabilities. (NFR4) DYSCO reschedules components repeatedly to find the optimum and to ensure QoS. (NFR5) DYSCO handles the failure of individual components. Failed components have no effect on the availability of the other system components. (NFR6) There is no downtime when rescheduling application components to different nodes in the cluster.

DYSCO defines the following contextual elements to describe component requirements and node capabilities: name, compute domain, location, computing power, processor architecture, ephemeral memory, persistent memory, and network connectivity. The definition of Fog Computing lists location awareness as a contextual element to describe hardware nodes. DYSCO extends this approach and considers various contextual elements to achieve the best possible mapping between requirements and capabilities.

Realization

The following paragraph describes the realization of DYSCO. Four test cases validate whether DYSCO meets the requirements. We describe the setup to validate DYSCO in the following.

Figure 6.6 shows the design for conducting the test cases. DYSCO uses a Kubernetes cluster with three nodes: a powerful *Cloud Node*, a less powerful *Fog Node*, and a limited *Field Node*. The *Cloud Node* is an AWS EC2 t.large instance. The *Fog Node*

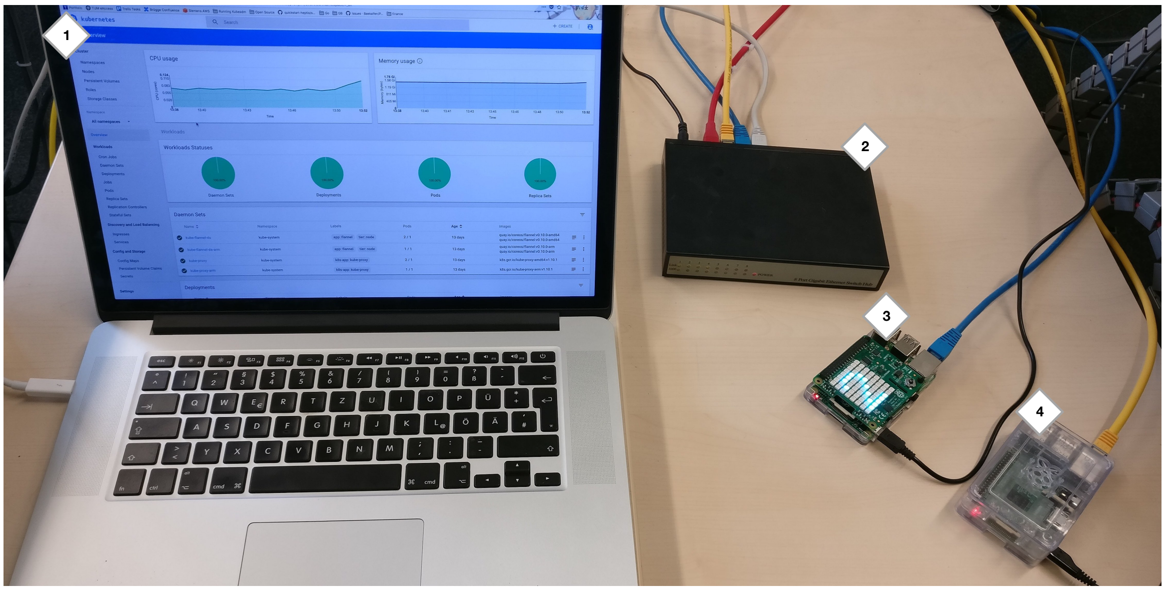

Figure 6.7: DYSCO test case setup

is an Ubuntu virtual machine running on a laptop. The *Field Node* is a Raspberry Pi 2. The *Fog* and *Field Node* are in the same factory Local Area Network (LAN), connected with a *Switch* while the *Cloud Node* in the AWS data center is in a different LAN. Kubernetes expects all nodes to be in the same LAN and therefore all nodes are connected via a *VPN Connection*. The Kubernetes cluster includes the *Cloud Node* as the Kubernetes master and the *Fog* and *Field Node* as Kubernetes workers. DYSCO uses Docker for the *Cloud* and *Field Node*, and containerd[10] for the *Fog Node*. The *Docker Hub* stores the required container images. A second Raspberry Pi represents the *IoT Device Controller*, including a small LED screen as actuator, a gyroscope sensor, and an accelerometer sensor for motion detection. The *IoT Device Controller* is not integrated in the Kubernetes cluster and simulates the production line of a factory.

Figure 6.7 shows the setup for the evaluation.

Test Scenarios. DYSCO schedules two different applications in the cluster: the timestamp and the emergency application, each consisting of two components. The timestamp (TM) application enables employees in a factory to register and store the beginning and end of their work. It consists of two components: the storage component stores the timestamp data and provides an interface with an overview of all registered timestamps. The register component provides an interface for employees to register their timestamps. The emergency (EM) application stops the assembly line of the factory in the event of an emergency. The stop component is responsible for stopping the assembly line, which is simulated by an exclamation mark on the Sense

[10] `https://containerd.io/`

HAT screen. The alarm component notifies interested parties about stop events, such as an emergency button being pushed or a malfunction in the machinery. In addition to the application components in the cluster, DYSCO provides a monitor component that supervises events of Kubernetes nodes. DYSCO uses kube-scheduler[11] as a monitoring extension that enables dynamic scheduling. The monitor component watches for status changes in nodes, including topology changes, node failures, or network failures. If such a case occurs, the kube-scheduler triggers a redeployment, which reorders and restarts the components considering the node capabilities and component requirements. Kubernetes allows status updates of workers nodes to be configured to the master and interprets missing updates as failures. DYSCO uses this mechanism to trigger redeployments. However, there is a trade-off between availability and network overhead. For testing purposes, we configured the update interval to 5 s. Kubernetes interprets missing updates as a failure if the node has not notified the master within 15 s. The DYSCO monitor requests node statuses every 5 s.

Deployment Mapping. DYSCO uses node labels[12] to describe node capabilities. The capabilities computing power, ephemeral memory, and persistent memory are described through resource requests and limits. To describe persistent memory capabilities, DYSCO uses persistent volumes. In contrast to Fogernetes, DYSCO uses the `nodeAffinity` constraint as it offers a more flexible description language than the `nodeSelector`. DYSCO differentiates between core requirements (CR) that must match a node capability and optional requirements (OR). For the test cases, we define the following values for the emergency and alarm application components:

- Timestamp Register: location = factory (OR), compute domain = fog (OR)
- Timestamp Storage: location = AWS (CR), compute domain = remote (CR)
- Emergency Stop: location = Factory (CR), compute domain = field (OR)
- Emergency Alarm: location = Factory (CR), compute domain = field (OR)
- DYSCO Monitor: location =AWS (CR), compute domain = remote (CR)

Test Cases. The following test cases validate the non-functional requirements of DYSCO. We executed the test cases in consecutive order for six iterations. The implementation test case 1 includes the setup of the cluster and the implementation of the components for the timestamp and emergency applications. The deployment test case 2 describes the deployment of application components in deployment files. DYSCO enables the deployment of components with a single deployment file and a

[11]`https://kubernetes.io/docs/concepts/overview/components`
[12]`https://kubernetes.io/docs/concepts/configuration/assign-pod-node`

Iteration	#1	#2	#3	#4	#5	#6
Before Test Case 3	🟦 DYSCO Monitor 🟦 TS Storage 🟨 EM Alarm 🟨 TS Register 🟩 EM Stop	🟦 DYSCO Monitor 🟦 TS Storage 🟩 EM Alarm 🟩 TS Register 🟩 EM Stop	🟦 DYSCO Monitor 🟦 TS Storage 🟩 EM Alarm 🟩 TS Register 🟩 EM Stop	🟦 DYSCO Monitor 🟦 TS Storage 🟨 TS Register 🟩 EM Alarm 🟩 EM Stop	🟦 DYSCO Monitor 🟦 TS Storage 🟨 EM Alarm 🟩 TS Register 🟩 EM Stop	🟦 DYSCO Monitor 🟦 TS Storage 🟨 EM Alarm 🟨 TS Register 🟩 EM Stop
After Test Case 4	🟦 DYSCO Monitor 🟦 TS Storage 🟩 EM Alarm 🟩 TS Register 🟩 EM Stop	🟦 DYSCO Monitor 🟦 TS Storage 🟩 EM Alarm 🟩 TS Register 🟩 EM Stop	🟦 DYSCO Monitor 🟦 TS Storage 🟨 TS Register 🟩 EM Alarm 🟩 EM Stop	🟦 DYSCO Monitor 🟦 TS Storage 🟨 EM Alarm 🟩 TS Register 🟩 EM Stop	🟦 DYSCO Monitor 🟦 TS Storage 🟨 EM Alarm 🟨 TS Register 🟩 EM Stop	🟦 DYSCO Monitor 🟦 TS Storage 🟨 EM Alarm 🟩 TS Register 🟩 EM Stop

Table 6.4: DYSCO deployment mapping

single command. The node failure test case 3 foresees the scheduling of application components according to their core and optional requirements. We simulate a failure by disconnecting the power supply of the *Field Node*. DYSCO reschedules the timestamp and emergency components to resume work. During rescheduling DYSCO provides 100 % availability. For the topology change test case 4, we reconnect the power supply. DYSCO reschedules the components according to their deployment mapping without downtime.

Results. DYSCO performs as expected according to test cases 1 through 3 and partially correctly according to test case 4. For test cases 3 and 4, we measured the time that the DYSCO monitor takes to trigger a rescheduling and the time it took to complete it. Table 6.4 visualizes the deployment mapping before test case 3 and after test case 4 for each iteration.

The emergency and alarm application were implemented in Python and Flask and show that DYSCO supports arbitrary programming languages or frameworks enabled by container technology. Each component of the timestamp and emergency applications was deployed with a single command and specified in a single file. DYSCO performs correctly according to test cases 1 and 2. On average, the DYSCO monitor triggered the failure recovery and rescheduling 20.7 s after disconnecting the power supply of the field node. The scheduler required an average of 6.5 s to reschedule the components of a failed node. After a node failure, DYSCO restored it to a working state in each of the six test case iterations with four different deployment mappings. During rescheduling, there was no downtime, thus DYSCO performed according to test case 3. DYSCO experienced varying delays in triggering the rescheduling of components after reconnecting the power supply. The delay consists of the time it takes the field node to start up and reconnect to the Kubernetes cluster, and for the DYSCO monitor to detect the new node. The time varied between 38 s in the fifth iteration and 84 s in the third iteration. Four different deployment mappings resulted from rescheduling in the six iterations. The scheduler respected all core requirements and only considered the optional requirements in the fifth iteration.

DYSCO implements parts of the Seamless Computing reference model and enables rescheduling of application components at runtime without downtime.

Chapter 7

Case Studies

In the previous chapters, we used application examples to show the applicability of the Fog Meta Model, Fogxy architectural style, and Seamless Computing formalizations. Figure 7.1 outlines all applied examples from the Manufacturing, Digital Health, and Smart Environments domains that utilized these formalizations. The application examples are assigned to the formalizations. In this chapter, we discuss two case studies in detail. Section 7.1 describes AIIoT, which uses emojis to visualize the state of IIoT devices in real-time. Section 7.2 describes IIoT Bazaar - a marketplace for industrial edge applications that applies Seamless Computing. Both projects allowed us to investigate the applicability of the Fog Meta Model and Fogxy in an industrial context.

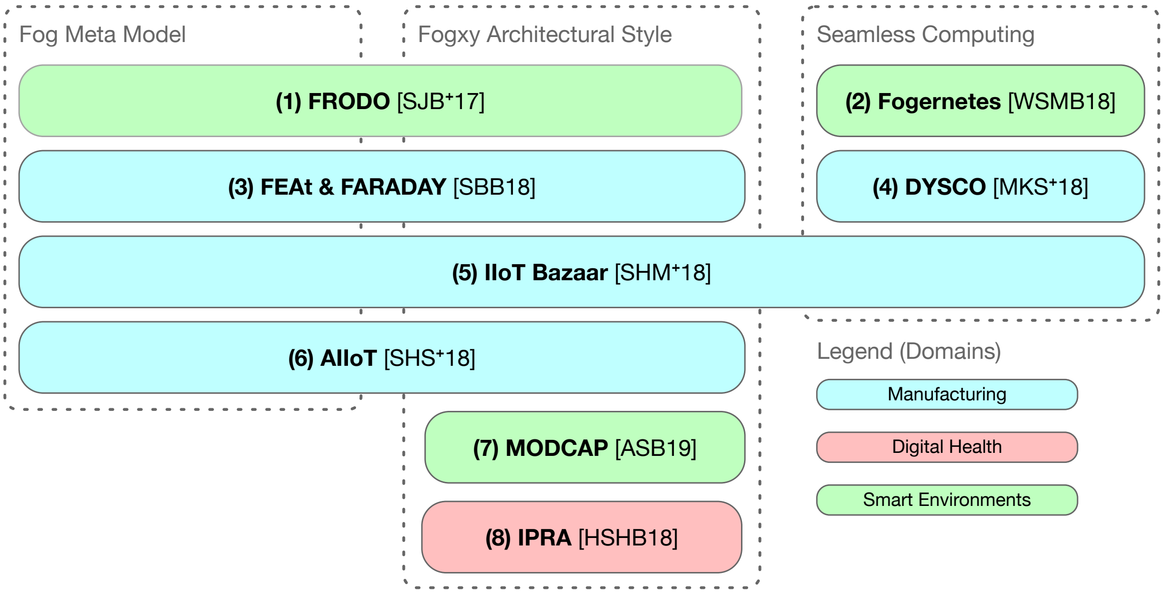

Figure 7.1: Overview of application examples and related publications

7.1 AIIoT

The vision of AIIoT (Augmenting Industrial Internet of Things) is to simplify the increasing complexity in industrial production plants and to demonstrate the combination of Fogxy with an innovative user interface using Augmented Reality (AR) and emojis. We applied Fogxy to address the real-time analysis and synchronization of sensor values. The increasing amount of data in industrial environments raises challenges for the infrastructure and workers. Furthermore, the increasing interconnectivity and digitization of the Industry 4.0 initiative lead to increasing complexity. AIIoT simplifies the decision-making process for employees.

When Scott Fahlman, a research professor of computer science at Carnegie Mellon University in Pittsburgh, invented the smiley (emoji) in September 1982, no one yet realized how this would affect the way we communicate and express our feelings today. Emojis have not yet penetrated the industrial domain but offer advantages. For instance, the status of machines can be depicted using emojis. Emojis help to get a quick impression of the condition of a machine. This information can be delivered in real-time and is provided in a way already familiar to workers. Systems become more user-centric and less machine-centric. Supported by augmented reality and emojis, an employee with contextual guidance and the information needed for successful decision-making can complete tasks faster and with fewer errors. AIIoT enables employees to obtain a quick overview and access to the processes in intelligent and networked products. Pure data collection is not sufficient; it must be transformed into information and made available.

AIIoT combines three technologies and concepts: IIoT, Fog Computing, and emojis in augmented reality. IIoT enables the collection of data and actuation of machines. With Fog Computing, data can be transported and processed effectively in real-time. The visualization of the data utilizing emojis leads to new interaction scenarios in the industrial environment.

Visionary Scenario

The following visionary scenario presents the functionality of AIIoT: Martin, an employee of an industrial manufacturing company, is responsible for a production line. He is in charge of ensuring that this production line is running correctly, without any downtime. Martin can walk around the production facility and use his smartphone to obtain information on the status of individual components. Martin uses the smartphone camera to detect and identify the device for which he wants to retrieve information. The app on his smartphone recognizes the type of the device, identifies it and automatically obtains all available data for this device. Since Martin has a large number of different devices in his production line, he does not know each one in detail. He would like to have a brief, concise summary of the device's state. The summary of the condition is provided by in place analytics and represented by emojis. If a device shows abnormal behavior, Martin can visualize the data using different graphs.

To make this scenario a reality, the following requirements must be met:

(1) Device Detection: The device of interest must be detected without markers or labels. The three-dimensional outline of the device must be sufficient for detection.

(2) Device Identification: After successful detection of a device, it must be identified. A unique identifier must indicate the individual device. Several devices of one type may be used in one production plant. The identifier is required for interaction with the data and the device.

(3) Data Gathering: The unique identifier is used to establish the relationship between the device and its characteristics. The available characteristics of the device must be gathered.

(4) Data Interpretation: Once the data for the device is available, it must be interpreted. The state of a device can only be displayed when the data is interpreted. Valuable information is then available for processing.

(5) Device Characteristics Visualization (Emoji or Graph): The available device characteristics must be visualized in a way appealing to employees. In the first step, the condition is mapped by emojis, which provide a quick overview. Graphs are used for further analysis.

The analysis object model in Figure 7.2 describes the structure of AIIoT from the problem domain. By using the Fog Metal Model, relevant classes for the remote, fog, and field layers can already be identified in the analysis phase.

The *Field Devices* for which interesting information is available must be detected. A *Production Line* consists of several installed devices. The respective device must be identified with a *Camera* to be able to access the data. The application on the mobile device receives the available *Real-Time Sensor Data. Real-Time Sensor Data* and *Historical Sensor Data* are children of *Sensor Data. Historical Sensor Data* is used to generate *Knowledge* by analysis. The *Interpreter* interprets available *Sensor Data* and gives them a meaning. For example, if the value of a sensor measurement exceeds a particular threshold value, this must be rendered visually for the employee as a *Visualization.* The increasing complexity of industrial systems and a large amount of sensor data present challenges for their evaluation and analysis. AIIoT supports employees and helps them to make decisions through *Visualizations.* There are two types of *Visualizations*: *Emoji* and *Graph.* The condition of the *Field Device* or *Sensor* can

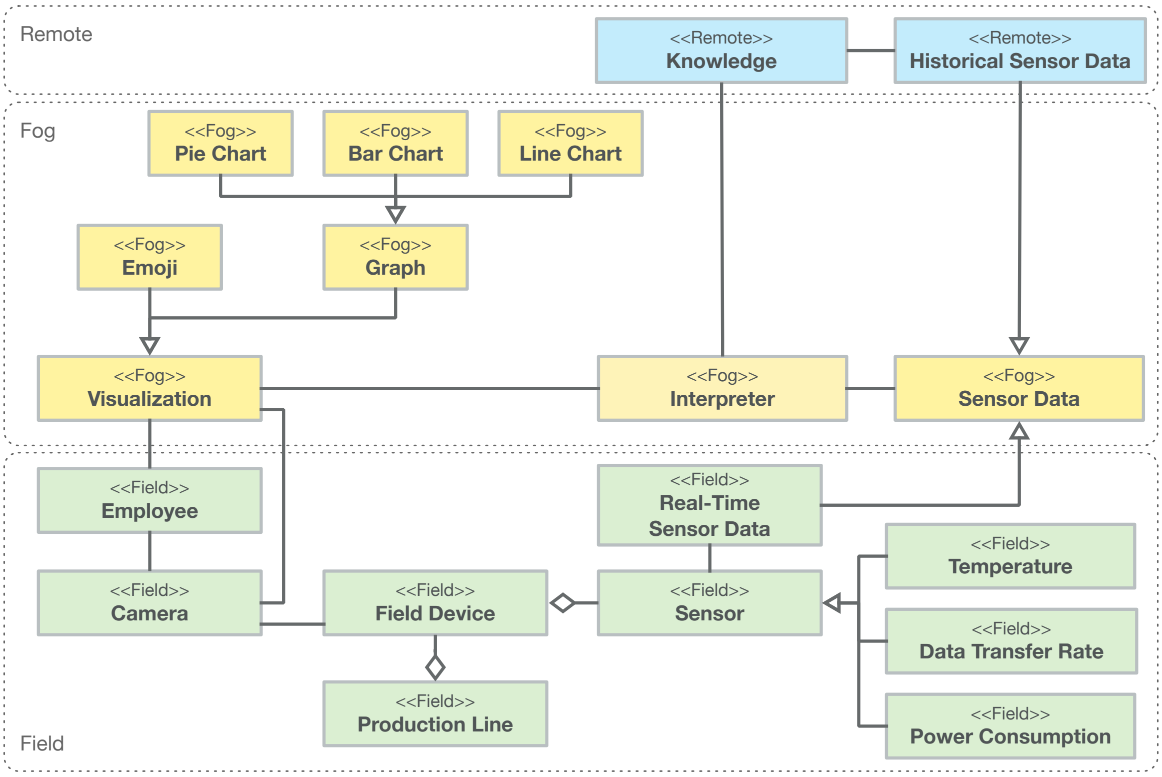

Figure 7.2: AIIoT analysis object model

be represented by a corresponding *Emoji* to offer the user available information at a glance. If the *Employee* wants to get detailed information or interpret the correlation between sensor data, the application offers a *Graph* interface. The application offers different emojis for different states as well as different types of graphs for data inspection (cf. *Graph* taxonomy in Figure 7.2). The sensor, visualization and graph taxonomies allow the extensibility of AIIoT.

Design

Figure 7.3 shows the top-level design of AIIoT. *Field Devices* and *Sensors* are located in the field layer - the source of the data. *Sensors* are attached to a *Field Device* that reads the sensor values and forwards it to the corresponding *Fog Node*. Examples of *Field Devices* are Siemens IOT2040[1], Dell Edge Gateway[2], and Krones ReadyKit[3]. For safety and architectural reasons, it is usually not possible to use the values of already installed sensors in existing production lines. To solve this problem, additional sensors are attached to collect the data for data analysis. Production lines with a planned duration of more than 20 years can be upgraded for the use of

[1]http://w3.siemens.com/mcms/pc-based-automation/en/industrial-iot/
[2]http://www.dell.com/us/business/p/edge-gateway
[3]https://www.krones.com/de/readykit.php

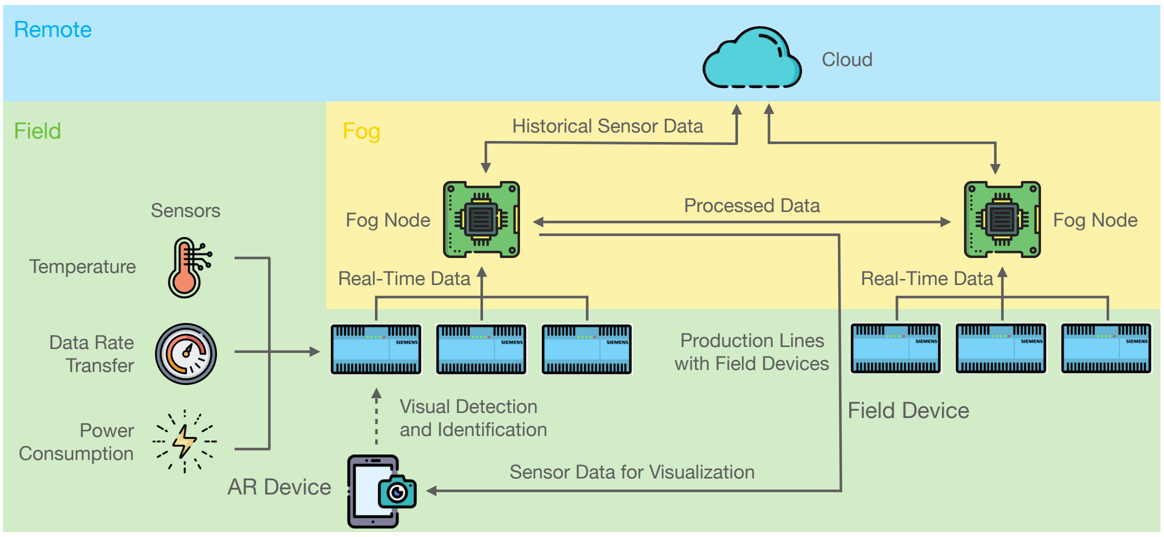

Figure 7.3: AIIoT top-level design

these technologies. An *AR Device* is the interface between field and fog. *Devices* are detected and identified through vision, e.g. the camera. The *AR Device* detects and identifies the *Field Devices* and receives data from the corresponding *Fog Node*.

Fog Nodes are responsible for data aggregation, interpretation, and provision. *Fog Nodes* are located in the fog layer near the data sources (*Field Devices*). *Devices* in the field tend not to know anything about their neighbors, but *Fog Nodes* do. Therefore, they have the possibility of exchanging data and models for interpretation. Multiple *Field Devices* are assigned to one *Fog Node*. This facilitates the combination of values from different sensors and the detection of dependencies. The result of the interpretation is transmitted and visualized on the *AR Device*. The *Fog Nodes* forward the aggregated data to the *Cloud*. A *Fog Node* serves as a proxy. It decides what happens to the data, which data is passed on and which visualizations have to be carried out based on the interpretation.

The *Cloud* serves as a brain that receives sensor data from *Fog Nodes* and metadata from other systems such as CRMs and ERPs. Data aggregation across multiple production lines is possible. The *Cloud* can analyze data and generate new knowledge with the help of machine learning. This newly acquired knowledge is shared in the form of models with the *Fog Nodes*. Models enable *Fog Nodes* to interpret data and derive visualizations.

Since AIIoT is based on Fogxy, the real-time access and synchronization requirements can be simultaneously achieved. On the one hand, the sensor data can be analyzed in real-time and visually depicted in the form of graphs or emojis on an *AR Device*. On the other hand, historical sensor data is stored in the remote layer. Based on the data, new insights can be gained and applied in practice.

Realization

Together with an industrial partner, we implemented AIIoT in fall 2017. Since the implementation involving industrial factories is challenging, we used a mobile demonstration board representing a production line (see Figure 7.4). The demonstration board consists of three IOT2040 devices, a router for local WiFi and an Intel Nuc as a fog node. In the remainder of this section, we describe the technical details, structure, and implementation of the prototype.

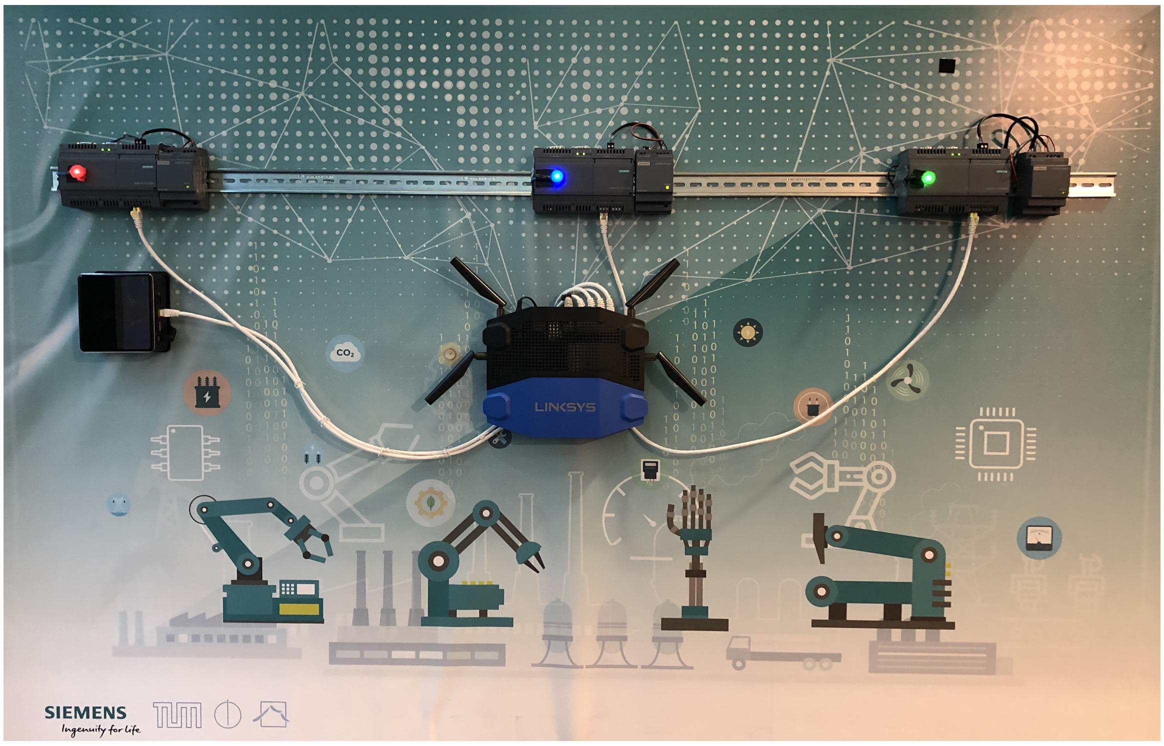

Figure 7.4: AIIoT demonstration board

(1) Device Detection: The IOT2040 device detection is based on the SSD MobileNet machine learning model using supervised learning. The model uses the TensorFlow Object Detection API[4]. The TensorFlow to CoreML Converter[5] converts the training model into a Core ML[6] compliant model, which is transferred in the iOS application. 800 pictures of devices were used to train the model. The model achieved an accuracy of 95 % to detect IOT2040 devices in previously unknown pictures.

(2) Device Identification: IOT2040 devices are equipped with QR codes to allow the unique identification of each device. However, since the approach is extensible

[4]`https://github.com/tensorflow/models/blob/master/research/object_detection/`
[5]`https://github.com/tf-coreml/tf-coreml`
[6]`https://developer.apple.com/documentation/coreml`

and not all devices can be equipped with QR codes, devices can also be identified using colored LEDs.

(3) Data Gathering: The IOT2040 devices are wired to the router. Since the field devices are close to the fog node, real-time interpretation of data is possible. The fog node uses mDNS to promote the offered services in the local network and a MQTT broker for communication. The IOT2040 devices act as publishers and transmit their sensor values. Interested parties can subscribe to broker topics and access those sensor values. Besides the broker functionality, the fog node analyzes and aggregates the data. The cloud component serves as a storage repository for historical data.

(4) Data Interpretation: Due to the lack of computing power and to save energy, the interpretation of the data is not performed on the field nodes (IoT device or smartphone), but on the fog node. We used a simplified threshold model, i.e., sensor values must be within a certain range, otherwise erroneous behavior is assumed. Based on the state of the device, the fog node decides which emoji has to be displayed.

	High	**Low**	**Zero**
Temperature	🔥	❄️	n.a.
Power Consumption	⚡	💡	🔌❗
Resource Consumption	⏲️	⏲️	n.a.
Data Transfer Rate	🚀	🐌	🔗
Idle Time	💤	n.a.	n.a.

Table 7.1: AIIoT emoji to sensor mapping

(5) Device Characteristics Visualization: Table 7.1 shows the mapping between sensor states and emojis to visualize the condition of devices. For example, a fire emoji indicates that the temperature is above a certain threshold. A snowflake indicates that the temperature is below a certain threshold . For characteristics such as power consumption or data transfer rates, it is relevant if the value of the sensor is zero. Smileys as shown in Figure 7.5 represent the general state of a device and its sensors. A happy smiley indicates that everything is within the given range and working as expected. The unhappier the smiley looks, the more critical the situation is.

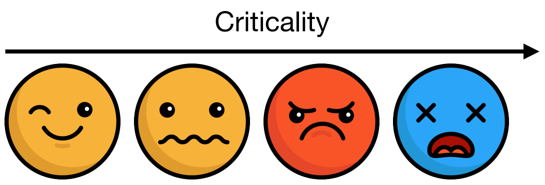

Figure 7.5: AIIoT smiley visualization for the state of a device

To get an in-depth look at the data, the application offers a graph view. A time axis visualizes available characteristics in real-time. The application dynamically adds and removes available sensors to the graph to detect and analyze correlations between different characteristics. Employees can share interesting graph progressions with others by pinning graphs into the augmented reality view.

Findings

Figure 7.6 shows a screenshot where the temperature of the *Flux Capacitor* device is too high. A fire emoji is displayed above the IOT2040 device using AR. The position of the emoji is fixed and therefore always remains in the same position. Since the emoji is a 2D image, it is always positioned correctly for the user's perspective. Colored LEDs enable device identification.

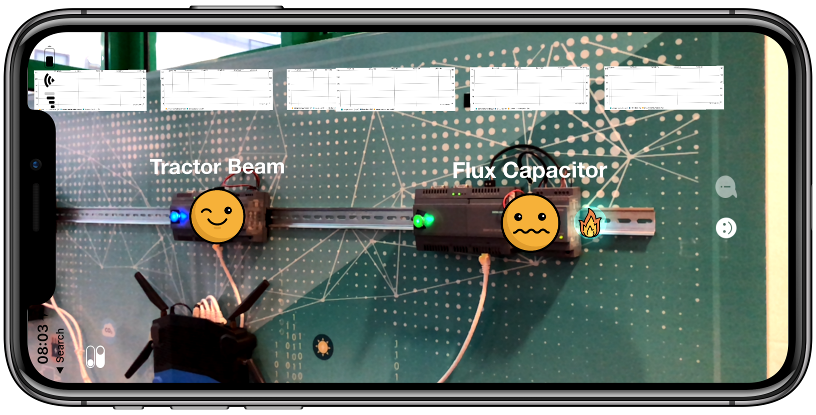

Figure 7.6: AIIoT AR user interface

AIIoT combines Fogxy with emojis in AR for IIoT applications. Emojis are easy to understand and reflect the condition of a machine at a glance. AIIoT supports the use of other AR devices besides the smartphone. Glasses or wearables offer the advantage that users have their hands free to interact with the machines. The AIIoT proof of concept implementation with partners from industry achieved the five requirements from the visionary scenario: (1) Device Detection, (2) Device Identification, (3) Data Gathering, (4) Data Interpretation, and (5) Device Characteristics Visualization. The combination of augmented reality, Fog Computing, and emojis to augment IIoT devices is a promising approach supported by industry.

7.2 IIoT Bazaar

The IIoT Bazaar case study extends the AIIoT system and combines it with a Blockchain component to create a marketplace for industrial edge applications. Fog Computing enables on-site data analysis, and Blockchain as a decentralized framework establishes security and trust. The combination of these technologies in an industrial environment opens up new possibilities and opportunities. The IIoT Bazaar is a decentralized marketplace for industrial edge applications[7] that relies on Blockchain to create transparency for all stakeholders involved and enable the traceability of app installations on field devices. Fogxy enables the integration of resource-limited field devices into the IIoT Bazaar ecosystem. Fog nodes provide applications on field devices and ensure integration into a decentralized Blockchain network. Augmented reality serves as an interface between the users and the machines, allowing people to interact intuitively with the field devices. We demonstrate the design and prototypical implementation of IIoT Bazaar and its applications. In this case study, we use the Fog Meta Model in the analysis phase to classify the field, fog, and remote layers. The architecture of the IIoT Bazaar is based on Fogxy around the decentralized field devices that connect with remote components via the components in the fog. Seamless Computing enables the deployment and administration of edge applications of the IIoT Bazaar. The case study enables the investigation of the integration and interaction of the formalizations Fog Meta Model, Fogxy architectural style, and Seamless Computing with this case study.

Problem and Motivation

Devices and software in industrial environments are distributed and heterogeneous and therefore difficult to manage since they are often used in many places for different purposes. Moreover, devices and their software are often custom tailored solutions, which makes them difficult to develop and maintain. In addition to the maintenance, the distribution and promotion of software and accompanying updates is tedious. Marketplaces for apps have emerged from the consumer world, where they are already well-known and used for smartphones. Examples of consumer marketplaces are the Apple App Store[8] and Google Play Store[9]. Users are accustomed to extending the functionality and possibilities of their smartphones with various apps. These processes are designed to be particularly user-friendly; after authentication and payment, the app is installed on the device and is immediately executable. Marketplaces bring

[7]We refer to industrial edge applications as applications running on field devices.
[8]`https://www.apple.com/ios/app-store/`
[9]`https://play.google.com/store`

advantages for users and for the developers or providers of apps. If an app reaches a critical mass, it is successfully distributed through a central point of contact. Contrary to app stores for smartphones, marketplaces in the industrial environment face various challenges. IIoT solutions are often tailor-made individual solutions in terms of both hardware and software. The reuse of applications is not possible due to the lack of distribution platforms. From the developer's point of view, the appropriate delivery channels are missing. Current solutions also do not offer flexible payment models such as pay per use.

From the perspective of field device developers and technicians, working with these devices raises further challenges. For example, technicians must go on-site and manually initiate the installation of software and updates. The devices in the field often do not provide a user interface, making feedback to the technician difficult and non-transparent. Besides this, there is a lack of standardization in the installation process of updates. They are not tracked, but reliable auditing and traceability are required. New payment models, such as purchasing time-based functionalities, are also attractive to companies using IIoT. A lack of technological possibilities currently prevents the establishment of such payment models. These challenges and problems, as well as the individuality of applications and use cases, make it difficult to develop a platform that meets these heterogeneous requirements. The IIoT Bazaar addresses these challenges and offers solutions.

Visionary Scenario

IIoT Bazaar's objective is to simplify and accelerate the installation and updating of apps as easily on field devices as on smartphones. The visionary scenario presents the functionality of the IIoT Bazaar.

Field devices serve as enablers for smart factories and establish continuous communication among different data sources that connect machines with sensors, actuators, and cloud applications. These field devices enable predictive maintenance, availability, and effectiveness. Technicians are responsible for regularly updating the software and applications on these devices, which is a process that is cumbersome and complex. Information about new software or updates is usually published via newsletters or internal portals. The technician must download the software, drag it onto a USB drive, and transfer it to the device that requires an update. With the increasing number of field devices, this method of updating poses a scaling problem. Many of the devices do not offer user interfaces and provide installation status feedback via LEDs. This process is error-prone and in need of improvement. In the consumer world, there is already a simple and elegant way which can also be used for the industrial domain:

marketplaces for apps. AIIoT enables technicians to detect and identify devices by using a smartphone with AR technology and thus provide a user interface for a field device that would otherwise lack one. The usually large number of manual steps for installation is reduced to a simple drag & drop operation. When the technician points to a device, the technician's smartphone shows which applications are running on the device, which updates are available, and which new applications can be installed. The technician takes an app from a list and drags it to the device in AR. The process is started after authentication and visually displayed in AR. If there is an error, the technician receives feedback in AR. IIoT Bazaar also provides the technician with additional information about the apps: ratings, version number, descriptions, developer information, and price. The metadata about the availability of apps and the traceability of installations are stored in the Blockchain, allowing transparency and traceability. The Blockchain, as a distributed and open system, enables several distributed marketplaces to work in parallel.

To realize the visionary scenario, the following requirements must be met:
(1) Open Platform: The IIoT Bazaar is an open platform. Barriers to entry for both developers and users must be low. The platform must create transparency for all parties involved.
(2) End-To-End Delivery: The IIot Bazaar supports the process of providing an application to install it on a field device. End-to-end means from uploading the app to the marketplace through the actual installation of the app on the field device to payment. Furthermore, there must be a possibility for updates.
(3) User-Centric Design: Usability must meet the same expectations as in the consumer world, i.e., the rapid notification of the current status of the devices, easy installations, and updates.
(4) Independence: The IIoT Bazaar is a decentralized system with no regulating authority.
(5) Payment Model Flexibility: The IIoT Bazaar supports different payment models. Developers offer and provide flexible payment models.
(6) On- and Off-Site Remote Update Management: Installation and update management must be available on-site and remotely. On-site technicians use their smartphones for interaction but also manage the devices remotely.
(7) Flexibility and Extensibility: IIoT Bazaar must be flexible and expandable to support different IIoT technologies and devices and be adaptable for the technicians.
(8) Traceability: All installations and updates must be documented for regulatory reasons. The respective processes must be visible and traceable to the different parties.

The analysis object model in Figure 7.7 describes the structure of AIIoT according to the visionary scenario.

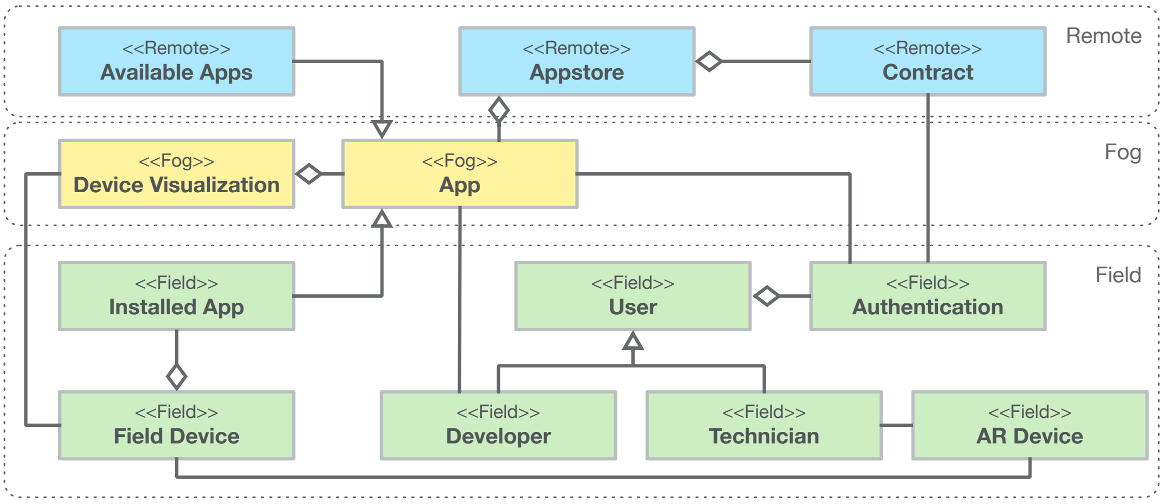

Figure 7.7: IIoT Bazaar analysis object model

IIoT Bazaar has two actors: application *Developers* who want to distribute their apps and *Technicians* who are responsible for the installation and availability of applications. *Developers* create applications (*Apps*) to run on *Field Devices*. As soon as an *App* is deployed on a *Field Device*, it is considered as *Installed App*. Apps available for installation are called *Available Apps* and subclasses of *App*. *Technicians* oversee the installation and updating of those applications on the *Field Devices*. Both the *Technician* and the *Developer* require *Authentication*, the *Developer* to upload *Apps* to the *App Store*, and the *Technician* to install and update *Apps* on *Field Devices*. It is essential to keep track of who developed which application and ensure that the correct applications are being installed by the *Technician* on the *Field Device*. All interactions with *Apps* are recorded in *Contracts*, such as uploading or installation. The *Contract* validates and confirms this information. For interaction with a *Field Device*, a *Technician* uses an *AR Device*. With the help of a camera, the *Field Device* can be identified and the *Device Visualization* digitally represents the device and thus enables interaction.

Design

Figure 7.8 shows the top-level design for the IIoT Bazaar. The design is based on the Fogxy architectural style, the AIIoT architecture and the hybrid approach as described in [RMC$^+$18]. We discuss the individual components and their roles within the IIoT Bazaars ecosystem.

The *IIoT Bazaar App* is the link between the *Field Devices* and the *Technician*.

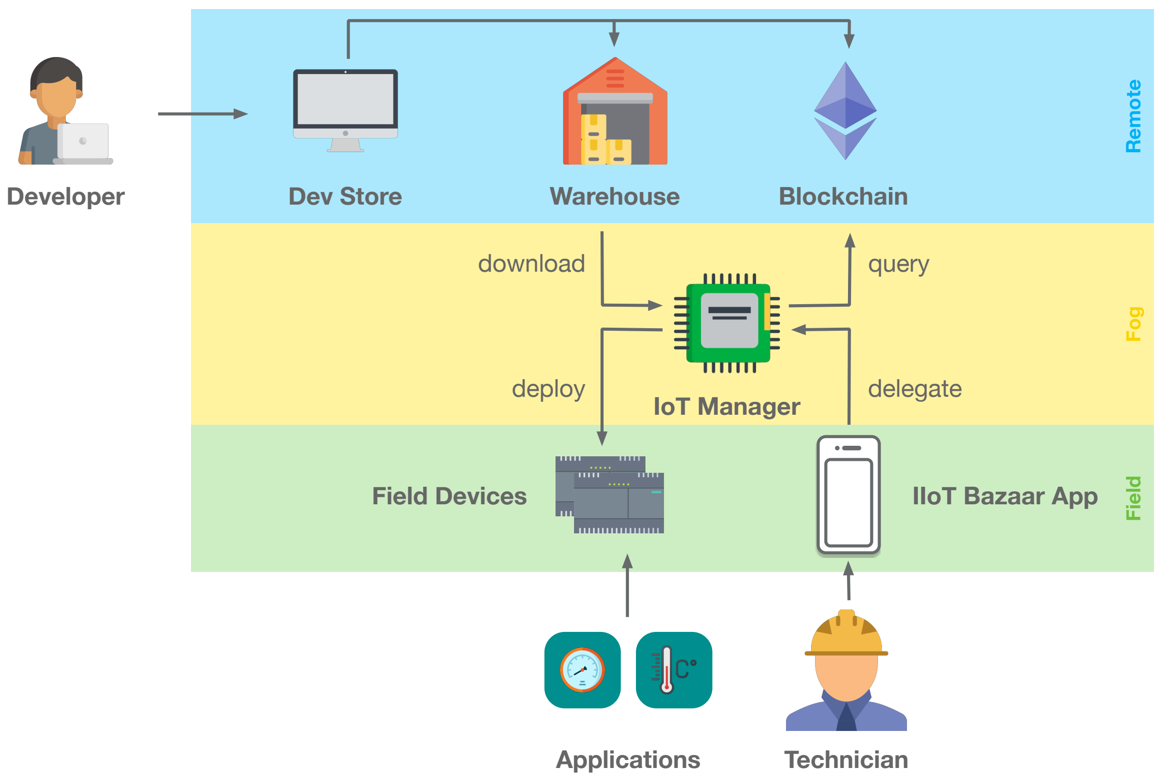

Figure 7.8: IIoT Bazaar top-level design

Field Devices rarely have user interfaces. Once the *IIoT Bazaar App* has been installed, *Technicians* use their smartphones to detect, identify, and interact with the *Field Devices* in AR. After successful authentication, technicians can install or uninstall apps using drag & drop operations. Furthermore, the *IIoT Bazaar App* provides information about the current status of the *Field Devices* and their interactions concerning app installations and updates. Running on the *Fog Node*, the *IoT Manager* is the heart of the architecture and central point of contact. The *Fog Node* establishes the connection between the components in the remote layer and those in the field. The *IoT Manager* compensates for the missing resources on the *Field Device* to enable on- and off-site maintenance and installation of apps. The *Dev Store* is the interface of the IIoT Bazaar ecosystem for developers. They can provide their applications, upload updates, or view metrics about the distribution of their apps. An app in the IIoT Bazaar consists of a binary file, images such as icons and screenshots, and metadata such as name, price, and version number. The *Dev Store* passes the metadata with reference to the *Warehouse* and *Blockchain*. There are multiple reasons for using a *Blockchain*. First of all, it provides a single source of truth in an environment with a possible lack of transparency. Second, it is open and therefore feasible to introduce further parties such as additional marketplaces to the IIoT Bazaar environment. Third, the Blockchain offers smart contracts, which deliver data storage and logic layers in a distributed fashion. Fourth, cryptography provides a tamper-proof way of recording transactions to enhance trust and traceability. The apps and other information such as screenshots or app icons are stored in the *Warehouse*. App entries in the Blockchain refer to the data in the *Warehouse*. *Technicians* refer to the information which apps are available from the *Blockchain*. Suitable apps are downloaded from the *Warehouse* to the selected *Field Device*.

Realization

Working together with an industrial partner, we implemented the IIoT Bazaar in spring 2018. The developed prototype serves to validate the concept of the IIoT Bazaar. The industrial partner uses the prototype to verify the acceptance of the system by field technicians. The objective is to create an innovative AR-based user interface that supports the deployment of edge applications on field devices. Furthermore, the Blockchain as trending technology from a software engineering point of view is of particular interest. We investigate the integration within Fogxy. In the following, we describe the technical details, the test setup, the implementation of the user interface, and the integration with Seamless Computing.

Figure 7.9 shows the hardware/software mapping as a UML deployment diagram that includes the components and communication mechanisms. The components are contained in the following environments:

Remote Environment. The *RemoteEnvironment* contains three subsystems: *Dev Store*, *Warehouse* and *App Chain*. The components *Dev Store* and *Warehouse* are deployed on a self-hosted virtual machine. The *App Chain* runs on AWS. The *Dev Store* component is implemented using Angular 6.0 and runs on a nginx web server. The web application includes two components: user management and application management. The former allows users to log in with their Ethereum key pair via the MetaMask[10] browser extension. We used the JavaScript implementation of the web3[11] framework to communicate with the *App Chain* via JSON RPC. The *Warehouse* component is implemented in Kotlin[12] on the Spring[13] framework. The *Warehouse* provides a REST interface that receives and stores the additional app information from the *Dev Store*. Furthermore, it offers an interface for the *IoT Manager* where the Dockerfiles of the app can be obtained to enable the installation on the *FieldDevices*. The component is packed in a Docker container and can be deployed to any cloud provider. A private

[10]`https://metamask.io/`
[11]`https://github.com/ethereum/web3.js`
[12]`https://kotlinlang.org/`
[13]`https://spring.io/`

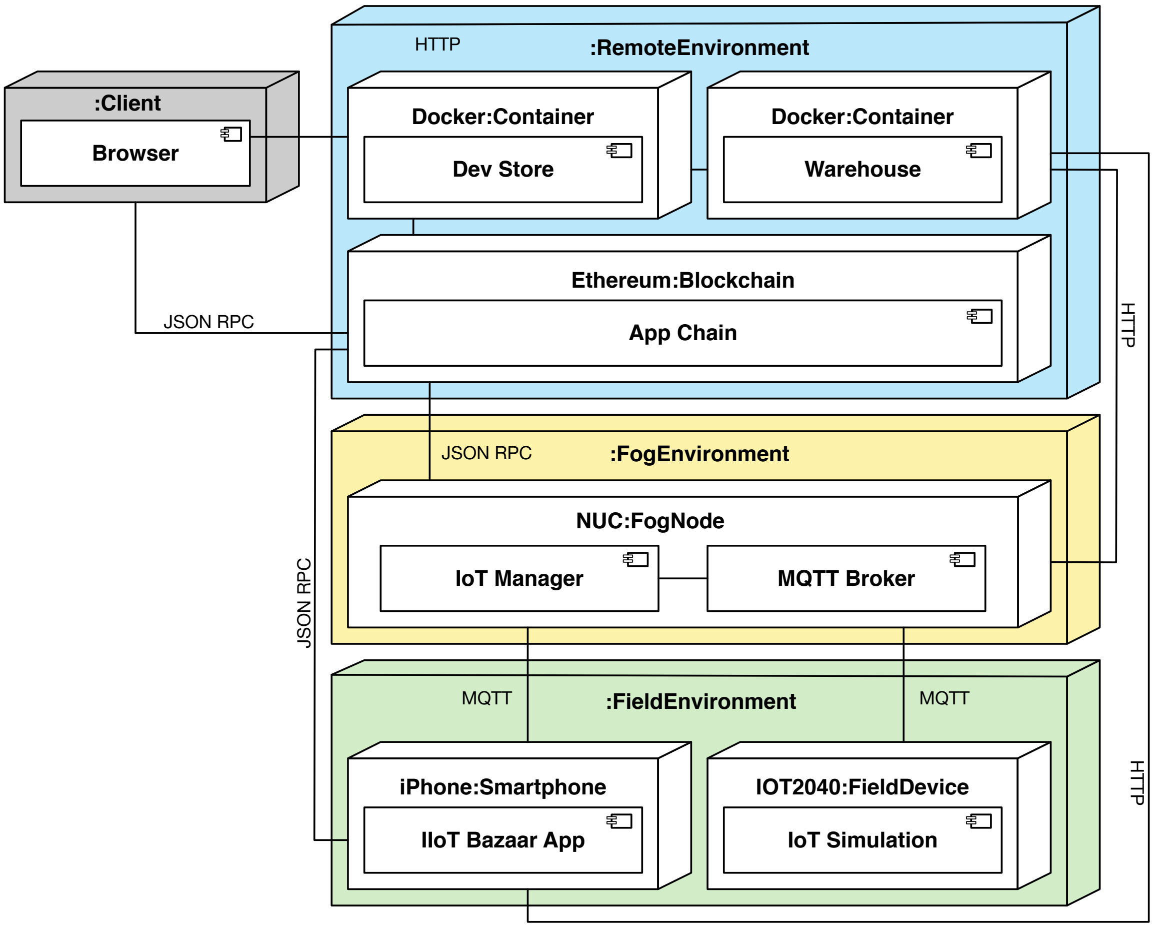

Figure 7.9: IIoT Bazaar deployment diagram

Ethereum Blockchain is used to implement the Blockchain component, which runs on three AWS Ethereum nodes. They find consensus on the state of the Blockchain using Proof-of-Authority. Further Ethereum nodes can be used to scale the system. IIoT Bazaar establishes three Smart Contracts: the *IIoTAppRecords* includes the application metadata containing the following characteristics: name, version, owner (developer's Ethereum address), release date, price (token), and a reference to retrieve additional data that may contain the application binaries (*Warehouse* base URL). The *IIoTInstallationRecords* stores the information about the installation of an app to the *FieldDevices*. DeviceId, appId, purchaseDate, installDate, and a flag indicating whether the installation of an app is still active or the app has been uninstalled are taken into account. The *IIoTToken* enables the payment and is derived from the EC20 standard token – the standard implementation for currency management on Ethereum. IIoT Bazaar uses a private Ethereum Blockchain for performance reasons and to verify the participants of the ecosystem via the nodes. Therefore, the Blockchain establishes transparency and visibility for all stakeholders.

Fog Environment. The *IoT Manager* and the *MQTT Broker* running on an *Intel NUC* with Ubuntu as operating system represent the central point of communication and computation in the *FogEnvironment*. The *IoT Manager* acts as a proxy between the remote components and the components in the *FieldEnvironment*. The *IoT Manager* is implemented in Typescript and uses the Node.js JavaScript runtime. As a proxy, the *IoT Manager* provides applications from the *Warehouse* to the respective *FieldDevices*. Edge applications from the *Warehouse* are loaded to the *FieldDevice* via the *IoT Manager* after a technician has completed the installation. The MQTT Broker distributes data among all the components of the system.

Field Environment. The *FieldEnvironment* contains two subsystems: the *IIoT Bazaar App* running on an iPhone and the *IoT Simulation* component running on IOT2040 devices. iPhones with the *IIoT Bazaar App*, the *FieldDevices*, and the *Fog Nodes* must be available on the same local network to communicate. At start-up time, the app connects to the *MQTT Broker* and subscribes to MQTT topics. MDNS resolves the IP address of the broker. MQTT messages contain information about nearby *FieldDevices* and applications executed on them. A technician uses a smartphone camera to capture the surrounding environment. The trained machine learning model of AIIoT is reused and achieves an accuracy of 95 % to detect IOT2040 devices. After detection, the *IIoT Bazaar App* scans the device for a QR code to identify it. The technician sees the installed apps on the device and the apps available for installation. Apple's ARKit enables the interaction in AR and the integration of the machine learning model using CoreML. JSON RPC and the web3swift[14] framework realize the communication with the *App Chain*. The test environment provides three *FieldDevices* with several sensors. Each of the *FieldDevices* runs the *IoT Simulations*. The *IoT Simulation* represents edge applications that can be deployed. The *FieldDevices* run Docker and portainer[15] to enable the remote deployment of edge applications packaged in Docker containers.

The *IIoT Bazaar App* provides information about the identified *FieldDevices* and their applications in AR. The intuitive interaction with the *FieldDevice* in AR is essential for the IIoT Bazaar to gain acceptance by technicians. We tested different user experience concepts in several iterations, and evaluated them with our industrial partner and finally decided on a drag & drop solution. The user interface adapts itself according to the camera angle and the *FieldDevice* that is in the focus of the AR

[14]`https://github.com/BANKEX/web3swift`
[15]`https://www.portainer.io/`

view. If several *FieldDevices* are available, the lowest common denominator of all the apps installed on these devices is displayed. Figure 7.10 shows the installation of the *Idle Time* application. It is dragged from the static area below into the dynamic AR area. A visual confirmation gives the *FieldDevice* a green frame around the device indicating that the *Technician* can install the app on the selected device. The installation process is initialized after the drop and visualized in the AR view (loading bar on the app icon). The *IoT Manager* ensures error-free installation and handles the billing for the app's costs.

Seamless Computing enables the management and deployment of edge applications. According to [BZ17], we use Docker as a software container platform to deploy edge applications. Portainer running on the field devices enables the deployment and management of applications. We deliberately decided against the use of a full Kubernetes cluster to keep the footprint small on the field devices. Portainer provides interfaces for the Docker runtime to install containers which are provided on the *IoT Manager*.

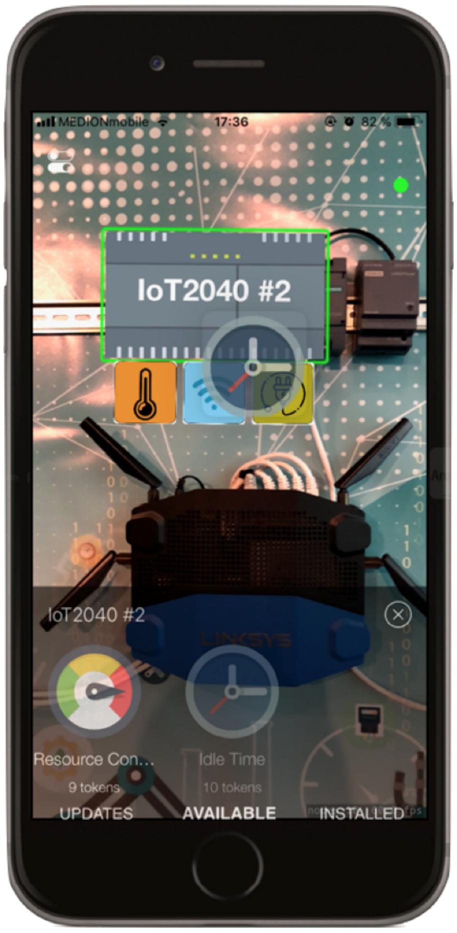

Figure 7.10: IIoT Bazaar AR user interface

The IIoT Bazaar extended the AIIoT system by adding real-time device interaction to obtain information about the installed applications and the ability to add or uninstall apps on field devices. The Blockchain stores all interactions performed by a technician, which increases transparency and traceability. It enables the realization of a marketplace for edge applications. Furthermore, Smart Contracts establish micropayments, including automation to create efficient and transparent processes. The IIoT Bazaar prototype was successfully demonstrated to our industrial partners, who provided valuable feedback. IIoT Bazaar achieves the following requirements: (1) Open Platform, (2) End-to-end Delivery, (3) User-Centric Design, (4) Independence, (5) Payment Model Flexibility, (6) On- and Off-Site Remote Update Management, (7) Flexibility and Extensibility, and (8) Traceability. However, we identified limitations posed by the current state of the prototype. We encountered difficulties that had not been apparent in the design of the system. Due to faulty frameworks and the lack of expert knowledge, development with a Blockchain component proved to be particularly challenging. Although we had used a private Proof-of-Authority Ethereum network, the confirmation times for the transactions took a long time. The IIoT Bazaar concept offers the potential for human-machine interactions and should be extended for machine-machine collaborations by using Smart Contracts for autonomous machines that are utilized as both producers and consumers, also called prosumers.

Chapter 8

Conclusion

This dissertation addresses developers of Fog Computing applications and provides them with formalizations to simplify the development of fog applications. The application of the formalizations allows the simultaneous realization of non-functional requirements such as real-time access and synchronization. The case studies and application examples from different domains demonstrated and represented innovative solutions to their problem statements. Section 8.1 describes the contributions and summarizes them. Section 8.2 gives an outlook on future work and presents research opportunities to improve the formalizations.

8.1 Contributions

This dissertation is a systems engineering work that contributes a metamodel, an architectural style, a pattern evaluation method, and a concept for build- and release-management of Fog Computing applications. The formalizations facilitate communication of stakeholders and establish a shared mental model among them. Figure 8.1 visualizes these contributions.

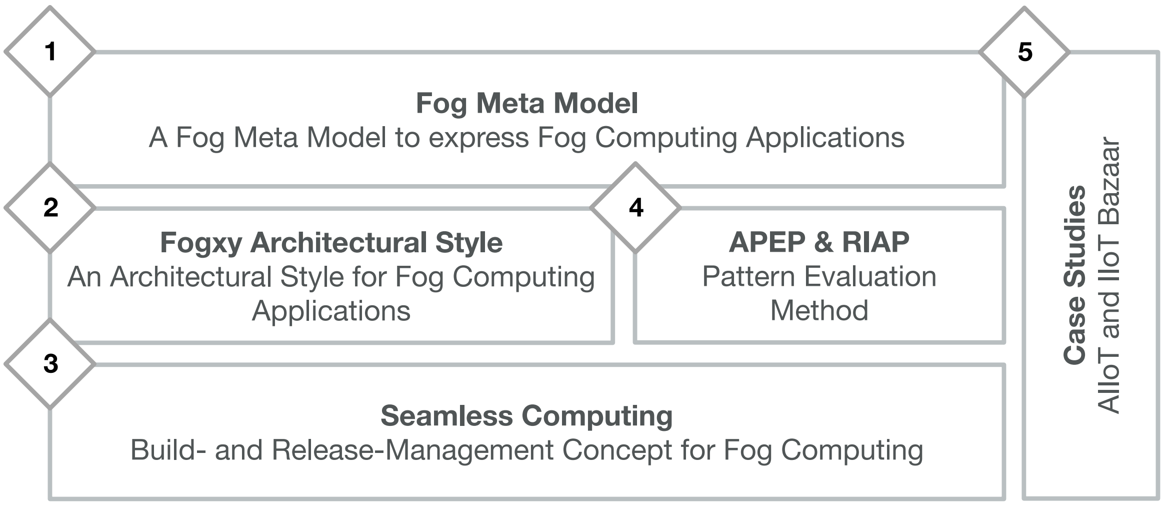
1
Fog Meta Model
A Fog Meta Model to express Fog Computing Applications
2
Fogxy Architectural Style
An Architectural Style for Fog Computing Applications
4
APEP & RIAP
Pattern Evaluation Method
3
Seamless Computing
Build- and Release-Management Concept for Fog Computing
5
Case Studies
AIIoT and IIoT Bazaar

Figure 8.1: Contributions overview

Fog Meta Model. The Fog Meta Model provides the basis for Fog Computing applications. It serves as a means of communication for the developers and allows Fog Computing specific entities to be typed as early as in the analysis phase for domains such as manufacturing, smart environments, and digital health. Three academic innovation projects and two application examples (FRODO and FARADAY) in these domains demonstrated the applicability of the metamodel.

Fogxy Architectural Style. The Fogxy architectural style is based on the Fog Meta Model using a combination of established design patterns. It minimizes the usual trade-off problems between real-time access and synchronization, which has been demonstrated in detail by means of the FEAt and FARADAY application examples from the manufacturing domain.

Seamless Computing. The Seamless Computing concept enables the static and dynamic allocation of Fog Computing components, providing the developer with a design continuum treating field, fog, and remote devices homogeneously. This has been demonstrated in the Fogernetes and DYSCO application examples.

APEP and RIAP. RIAP is a lightweight method for evaluating architectural styles. APEP, an iterative process for performing architectural pattern assessments, integrates RIAP. RIAP helped to improve the Fogxy architectural style iteratively and is applicable to other architectural patterns.

AIIoT and IIoT Bazaar Case Studies. The AIIoT and IIoT Bazaar case studies demonstrated the applicability of the Fog Meta Model and the Fogxy architectural style in the industrial manufacturing domain. IIoT Bazaar used Seamless Computing to provide application components for field devices, demonstrating the feasibility of the concept. The case studies presented innovative solutions including the use of emojis and the establishment of a marketplace.

8.2 Future Work

In this section, we provide an outlook on future research directions and improvements to the Fog Meta Model, the Fogxy architectural style, and Seamless Computing. The Fog Meta Model is an enhancement of the UML metamodel and thus supports the extension to describe the context of hardware and software in the area of Fog Computing. In addition to real-time access, Fog Computing promises advantages for security, data protection, scalability, and reduced operating costs. The Fog Meta

Model does not yet take these into account, but they do represent attractive stereotype candidates. The integration of further non-functional requirements implies changes to the Fogxy architectural style, which must be adapted accordingly. In addition to the vertical distribution of component across layers, the horizontal partitioning of subsystems within the different layers must be considered to address scalability and reduced operating costs. The case studies showed configurations with isolated devices within a layer that communicated exclusively across layers. It might be interesting to generalize the approach to allow devices to communicate within a layer. An open research topic is the definition of the wisp around a fog node so that devices can see and communicate with each other. Since Fog Computing is a continuum between the remote and the field layer, it is feasible for applications to consist of more than three layers that blur the characteristics of the layers. Fog nodes might become devices in the field layer and use the services of nodes from the layer above, which might be another fog layer or a remote layer.

Seamless Computing targets applications in the manufacturing domain where hard real-time access plays a crucial role. However, it is an open research question whether the applied container technology meets the requirements of automation technology. Real-time systems require orders of magnitude less processing time than is currently feasible with container technologies. It is interesting to see the extent to which the concepts are compatible. The application of Seamless Computing in the field of mobile computing is also exciting and needs to be considered.

Furthermore, we want to apply the formalizations in other application domains and to other project types. No statement can be made about the applicability of formalizations in interface or reverse engineering projects since the application examples were mostly greenfield engineering projects. This needs to be explored further. In the digital health domain, the application of the formalizations is promising. For example, the interpretation of patient-related sensor data must be carried out in real-time to prevent injury after a hip operation, while considering the accuracy of data analysis, power consumption, and size and weight of the device attached to the patient. The movement data is synchronized with the physician to guarantee optimal treatment.

The close collaboration between industry and researchers is helping Fog Computing to become a technological driver of digitalization. The formalization of methods and models for Fog Computing applications is a contribution to facilitate the implementation for developers and increase the understanding of all involved parties. It is a great opportunity to be part of this movement and to drive its evolution.

Bibliography

[ABC$^+$97] Gregory Abowd, Len Bass, Paul Clements, Rick Kazman, Linda Northrop, and Amy Zaremski. Recommended Best Industrial Practice for Software Architecture Evaluation. Technical report, Software Engineering Institute, Carnegie Mellon University, 1997.

[AG10] Lee Ackerman and Celso Gonzalez. *Patterns-Based Engineering: Successfully Delivering Solutions via Patterns*. Addison-Wesley Longman Publishing Co., Inc., 2010.

[AIS77] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Center for Environmental Structure. Oxford University Press, 1977.

[Ale79] Christopher Alexander. *The timeless way of building*. Center for Environmental Structure. Oxford University Press, 1979.

[ASB19] Mariana Avezum, Andreas Seitz, and Bernd Bruegge. MODCAP: A platform for cooperative search and rescue missions. In *Software Engineering (Workshops)*, volume 2308 of *CEUR Workshop Proceedings*, pages 63–66, 2019.

[BBL76] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative Evaluation of Software Quality. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE, pages 592–605, Los Alamitos, CA, USA, 1976. IEEE.

[BBL01] Guillem Bernat, Alan Burns, and Albert Llamosi. Weakly Hard Real-Time Systems. *IEEE Transactions on Computers*, 50(4):308–321, April 2001.

[BBM13] Muhammad Ali Babar, Alan W. Brown, and Ivan Mistrik. *Agile Software Architecture: Aligning Agile Processes and Software Architectures*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2013.

[BCK98] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1998.

[BCK12] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice.* Addison-Wesley Longman Publishing Co., Inc., 3rd edition, 2012.

[BD09] Bernd Bruegge and Allen H Dutoit. *Object-Oriented Software Engineering Using UML, Patterns and Java.* Prentice Hall, 2009.

[Bec17] Alexander Becker. Integration of Machine Learning into a Fog Computing Environment in the Context of Smart Buildings. Bachelor's thesis, Technische Universität München, August 2017.

[BHS07] Frank Buschmann, Kevlin Henney, and Douglas Schmidt. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing (Volume 4).* John Wiley & Sons, 2007.

[BMMM98] William H. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley & Sons, 1998.

[BMR$^+$96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns.* Wiley Publishing, 1996.

[BMZA12] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, pages 13–16, New York, NY, USA, 2012. ACM.

[Bod18] Florian Bodlée. Autonomous Traffic Regulation using a Distributed Ledger. Bachelor's thesis, Technische Universität München, September 2018.

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999.

[Bro10] Manfred Broy. Cyber-Physical Systems: Innovation durch Softwareintensive eingebettete Systeme. *acatech DISKUTIERT*, 2010.

[BSS$^+$18] I. Bouzarkouna, M. Sahnoun, N. Sghaier, D. Baudry, and C. Gout. Challenges Facing the Industrial Implementation of Fog Computing. In *6th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 341–348. IEEE, 2018.

[Buc17] Dominik Buchinger. A Fog Computing Design Approach for Data Analytics in the Industrial Internet of Things. Master's thesis, Technische Universität München, November 2017.

[BZ17] Paolo Bellavista and Alessandro Zanni. Feasibility of Fog Computing Deployment Based on Docker Containerization over RaspberryPi. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*, ICDCN, New York, NY, USA, 2017. ACM.

[BZJ04] Muhammad Ali Babar, Liming Zhu, and Ross Jeffery. A framework for classifying and comparing software architecture evaluation methods. In *Australian Software Engineering Conference Proceedings*, pages 309–318. IEEE, 2004.

[CKK02] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures.* Addison-Wesley Longman Publishing Co., Inc., 2002.

[CLC11] Y. Chen, X. Li, and F. Chen. Overview and analysis of cloud computing research and application. In *International Conference on E-Business and E-Government (ICEE)*, 2011.

[Cle00] Paul C. Clements. Active Reviews for Intermediate Designs. Technical note, Carnegie Mellon, 2000.

[Con17] OpenFog Consortium. OpenFog Reference Architecture for Fog Computing. 2017.

[CSB17] C. Chang, S. Narayana Srirama, and R. Buyya. Indie Fog: An Efficient Fog-Computing Infrastructure for the Internet of Things. *Computer*, 50(9):92–98, 2017.

[CZS17] S. Chen, T. Zhang, and W. Shi. Fog Computing. *IEEE Internet Computing*, 21(2):4–6, 2017.

[DB16] A. V. Dastjerdi and R. Buyya. Fog Computing: Helping the Internet of Things Realize Its Potential. *Computer*, 49(8):112–116, 2016.

[DD17] Koustabh Dolui and Soumya Kanti Datta. Comparison of edge computing implementations: Fog computing, cloudlet and mobile edge computing. In *Global Internet of Things Summit (GIoTS).* IEEE, 2017.

[DMR16] Manuel Díaz, Cristian Martín, and Bartolomé Rubio. State-of-the-art, challenges, and open issues in the integration of Internet of things and cloud computing. *Journal of Network and Computer Applications*, 67:99 – 117, 2016.

[EA12] Peter C. Evans and Marco Annunziata. Industrial internet: Pushing the boundaries of minds and machines. *General Electric*, 2012.

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine, 2000.

[FM05] Elgar Fleisch and Friedemann Mattern. *Das Internet der Dinge.* Springer-Verlag GmbH, 2005.

[Fow97] Martin Fowler. *Analysis patterns: reusable object models.* Addison-Wesley Longman Publishing Co., Inc., 1997.

[FS99] Jason Flinn and M. Satyanarayanan. Energy-aware Adaptation for Mobile Applications. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP, pages 48–63, New York, NY, USA, 1999. ACM.

[Gaß17] David Gaßmann. Architectural Styles for Real-Time Applications in the Area of Fog Computing. Master's thesis, Technische Universität München, December 2017.

[GBMP13] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions. *Future Generation Computer Systems*, 29(7):1645–1660, 2013.

[GF94] O. C. Z. Gotel and C. W. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of IEEE International Conference on Requirements Engineering*, pages 94–101, 1994.

[GGdFP$^+$16] N. M. Gonzalez, W. A. Goya, R. de Fatima Pereira, K. Langona, E. A. Silva, T. C. M. de Brito Carvalho, C. C. Miers, J. E. Mångs, and A. Sefidcon. Fog computing: Data analytics and cloud distributed processing on the network edges. In *35th International Conference of the Chilean Computer Science Society (SCCC)*, 2016.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GLM+15] V. Gazis, A. Leonardi, K. Mathioudakis, K. Sasloglou, P. Kikiras, and R. Sudhaakar. Components of fog computing in an industrial internet of things context. In *12th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON Workshops)*, 2015.

[GMP+18] Spyridon V. Gogouvitis, Harald Mueller, Sreenath Premnadh, Andreas Seitz, and Bernd Bruegge. Seamless Computing in Industrial Systems using Container Orchestration. *Future Generation Computer Systems*, 2018.

[Gra92] Robert B. Grady. *Practical Software Metrics for Project Management and Process Improvement*. Prentice Hall, 1992.

[HA10] Neil B. Harrison and Paris Avgeriou. How Do Architecture Patterns and Tactics Interact? A Model and Annotation. *J. Syst. Softw.*, 83(10):1735–1758, 2010.

[HAHZ15] K. Habak, M. Ammar, K. A. Harras, and E. Zegura. Femto Clouds: Leveraging Mobile Devices to Provide Cloud Service at the Edge. In *IEEE 8th International Conference on Cloud Computing*, pages 9–16, 2015.

[Hel18] Till Hellmund. Applying the Fogxy Pattern for Real-Time Smart Glasses Applications. Bachelor's thesis, Technische Universität München, July 2018.

[HESB18] M. Heck, J. Edinger, D. Schaefer, and C. Becker. IoT Applications in Fog and Edge Computing: Where Are We and Where Are We Going? In *27th International Conference on Computer Communication and Networks (ICCCN)*, 2018.

[HNYL17] Z. Hao, E. Novak, S. Yi, and Q. Li. Challenges and Software Architecture for Fog Computing. *IEEE Internet Computing*, 21(2):44–53, 2017.

[HSHB18] Till Hellmund, Andreas Seitz, Juan Haladjian, and Bernd Bruegge. IPRA: Real-Time Face Recognition on Smart Glasses with Fog Computing. In *Adjunct Proceedings of the ACM International Joint Conference on Pervasive and Ubiquitous Computing and the International Symposium on Wearable Computers*. ACM, 2018.

[Hum17] Jez Humble. Continuous Delivery Sounds Great, but Will It Work Here? *Queue*, 15(6):70:57–70:76, 2017.

[ISO00] ISO/IEC. Software Engineering - Product Quality (ISO/IEC 9126). Technical report, 2000.

[ISO14] ISO/IEC. Internet of Things (ISO/IEC JTC 1). Technical report, 2014.

[Jeu09] Manfred A. Jeusfeld. *Metamodel*, pages 1727–1730. Springer US, Boston, MA, 2009.

[KABW14] Stephan Krusche, Lukas Alperowitz, Bernd Bruegge, and Martin O. Wagner. Rugby: An Agile Process Model Based on Continuous Delivery. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, RCoSE 2014, pages 42–50, New York, NY, USA, 2014. ACM.

[Kat18] Florian Katenbrink. DYSCO: Dynamic Scheduling for Seamless Computing. Master's thesis, Technische Universität München, September 2018.

[KBAW94] R. Kazman, L. Bass, G. Abowd, and M. Webb. SAAM: a method for analyzing the properties of software architectures. In *Proceedings of 16th International Conference on Software Engineering*, pages 81–90, 1994.

[KKC00] Rick Kazman, Mark Klein, and Paul Clements. ATAM: Method for architecture evaluation. Technical report, Software Engineering Institute, Carnegie Mellon University, 2000.

[KMS$^+$18] Florian Katenbrink, Ludwig Mittermeier, Andreas Seitz, Harald Mueller, and Bernd Bruegge. Dynamic Scheduling for Seamless Computing. In *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*, pages 41–48, Nov 2018.

[Kra18] Paul Johannes Kraft. FogxyAR - Towards a Fog Computing Framework for Collaborative Augmented Reality. Bachelor's thesis, Technische Universität München, July 2018.

[KRM16] M. A. Kabir, M. U. Rehman, and S. I. Majumdar. An analytical and comparative study of software usability quality factors. In *7th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 800–803, Aug 2016.

[Kru95] P. B. Kruchten. The 4+1 View Model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[LDZQ18] J. Luo, X. Deng, H. Zhang, and H. Qi. Ultra-Low Latency Service Provision in Edge Computing. In *IEEE International Conference on Communications (ICC)*, 2018.

[Lee08] E. A. Lee. Cyber Physical Systems: Design Challenges. In *11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369, 2008.

[LES^{+}14] Grace A. Lewis, Sebastian Echeverría, Soumya Simanta, Ben Bradshaw, and James Root. Cloudlet-based Cyber-foraging for Mobile Systems in Resource-constrained Edge Environments. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion, pages 412–415, New York, NY, USA, 2014. ACM.

[LJY^{+}15] J. Li, J. Jin, D. Yuan, M. Palaniswami, and K. Moessner. EHOPES: Data-centered Fog platform for smart living. In *International Telecommunication Networks and Applications Conference (ITNAC)*, pages 308–313, 2015.

[LKY14] Jay Lee, Hung-An Kao, and Shanhu Yang. Service Innovation and Smart Analytics for Industry 4.0 and Big Data Environment. *Procedia CIRP*, 16:3 – 8, 2014.

[MGH^{+}16] Harald Mueller, Spyridon V. Gogouvitis, Houssam Haitof, Andreas Seitz, and Bernd Bruegge. Poster Abstract: Continuous Computing from Cloud to Edge. In *IEEE/ACM Symposium on Edge Computing (SEC)*, pages 97–98, 2016.

[MGSB17] Harald Mueller, Spyridon V. Gogouvitis, Andreas Seitz, and Bernd Bruegge. Seamless Computing for Industrial Systems Spanning Cloud and Edge. In *International Conference on High Performance Computing Simulation (HPCS)*, pages 209–216, 2017.

[MMA+16] Eva Marín-Tordera, Xavier Masip-Bruin, Jordi Garcia Almiñana, Admela Jukan, Guang-Jie Ren, Jiafeng Zhu, and Josep Farre. What is a Fog Node A Tutorial on Current Concepts towards a Common Definition. *CoRR*, 2016.

[MRW77] Jim A. McCall, Paul K. Richards, and Gene F. Walters. Factors in Software Quality. Concepts and Definitions of Software Quality. 1977.

[New15] Sam Newman. *Building Microservices.* O'Reilly Media, Inc., 1st edition, 2015.

[NSN+97] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile Application-aware Adaptation for Mobility. *SIGOPS Oper. Syst. Rev.*, 31(5):276–287, 1997.

[NZS15] David Sousa Nunes, Pei Zhang, and Jorge Sá Silva. A Survey on Human-in-the-Loop Applications Towards an Internet of All. *IEEE Communications Surveys Tutorials*, 17(2), 2015.

[Obj11] Object Management Group. OMG Unified Modeling Language Version 2.4.1, 2011.

[Obj16] Object Management Group. Meta Object Facility Specification Version 2.5.1, 2016.

[OSB15] J. Oueis, E. C. Strinati, and S. Barbarossa. The Fog Balancing: Load Distribution for Small Cell Cloud Computing. In *IEEE 1st Vehicular Technology Conference (VTC Spring)*, 2015.

[Par72] D. L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[Pet16] Sebastian Matthias Peters. *MIBO – A Framework for the Integration of Multimodal Intuitive Controls in Smart Buildings.* Dissertation, Technische Universität München, München, 2016.

[PMV+18] C. Puliafito, E. Mingozzi, C. Vallati, F. Longo, and G. Merlino. Virtualization and Migration at the Network Edge: An Overview. In *IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 368–374, 2018.

[PTW18] Yi Pan, Parimala Thulasiraman, and Yingwei Wang. Overview of cloudlet, fog computing, edge computing, and dew computing. In *The 3rd International Workshop on Dew Computing*. IEEE, 2018.

[PW85] David L. Parnas and David M. Weiss. Active Design Reviews: Principles and Practices. In *Proceedings of the 8th International Conference on Software Engineering*, pages 132–136. IEEE Computer Society Press, 1985.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the Study of Software Architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992.

[RLSS10] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic. Cyber-physical systems: The next computing revolution. In *Design Automation Conference*, pages 731–736, 2010.

[RMC$^+$18] Ana Reyna, Cristian Martín, Jaime Chen, Enrique Soler, and Manuel Díaz. On blockchain and its integration with IoT. Challenges and opportunities. *Future Generation Computer Systems*, 88:173 – 190, 2018.

[Roh17] Johannes Rohwer. Discovery of Nearby Nodes and Services in the Context of Fog Computing. Bachelor's thesis, Technische Universität München, September 2017.

[Sat01] Mahadev Satyanarayanan. Pervasive computing: vision and challenges. *IEEE Personal Communications*, 8(4):10–17, 2001.

[Sat15] Mahadev Satyanarayanan. A Brief History of Cloud Offload: A Personal Journey from Odyssey Through Cyber Foraging to Cloudlets. *GetMobile: Mobile Comp. and Comm.*, 18(4):19–23, 2015.

[Sat17a] Mahadev Satyanarayanan. Edge computing for situational awareness. In *IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*. IEEE, 2017.

[Sat17b] Mahadev Satyanarayanan. The Emergence of Edge Computing. *Computer*, 50(1):30–39, 2017.

[SB18] Andreas Seitz and Bernd Bruegge. Teaching Pattern-Based Development. In *Combined Proceedings of the Workshops of the German Software Engineering Conference (SE), Ulm, Germany.*, pages 20–23, 2018.

[SBB18] Andreas Seitz, Dominik Buchinger, and Bernd Bruegge. The Conjunction of Fog Computing and the Industrial Internet of Things - An Applied Approach. In *IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, 2018.

[SBCD09] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4):14–23, 2009.

[SCH$^+$14] M. Satyanarayanan, Z. Chen, K. Ha, W. Hu, W. Richter, and P. Pillai. Cloudlets: at the leading edge of mobile-cloud convergence. In *6th International Conference on Mobile Computing, Applications and Services*, 2014.

[Sch17] Constantin Scheuermann. *A Metamodel for Cyber-Physical Systems.* Dissertation, Technische Universität München, München, 2017.

[SCM18] S. Sarkar, S. Chatterjee, and S. Misra. Assessment of the Suitability of Fog Computing in the Context of Internet of Things. *IEEE Transactions on Cloud Computing*, 6(1):46–59, 2018.

[SCZ$^+$16] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[SFI16] Madiha H. Syed, Eduardo B. Fernandez, and Mohammad Ilyas. A Pattern for Fog Computing. In *Proceedings of the 10th Travelling Conference on Pattern Languages of Programs*, VikingPLoP, pages 13:1–13:10, New York, NY, USA, 2016. ACM.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[SGFW10] Harald Sundmaeker, Patrick Guillemin, Peter Friess, and Sylvie Woelfflé. Vision and challenges for realising the Internet of Things. Technical report, Cluster of European Research Projects on the Internet of Things, European Commission, 2010.

[SHM$^+$18] Andreas Seitz, Dominic Henze, Daniel Miehle, Bernd Bruegge, Jochen Nickles, and Markus Sauer. Fog Computing as Enabler for Blockchain-Based IIoT App Marketplaces - A Case Study. In *Fifth International Conference on Internet of Things: Systems, Management and Security*, pages 182–188, 2018.

[SHS$^+$18] Andreas Seitz, Dominic Henze, Markus Sauer, Jochen Nickles, and Bernd Bruegge. Augmenting the Industrial Internet of Things with Emojis. In *International Workshop on Smart Living with IoT, Cloud, and Edge Computing (SLICE)*, Barcelona, Spain, 2018.

[SJB$^+$17] Andreas Seitz, Jan Ole Johanssen, Bernd Bruegge, Vivian Loftness, Volker Hartkopf, and Monika Sturm. A Fog Architecture for Decentralized Decision Making in Smart Buildings. In *Proceedings of the 2nd International Workshop on Science of Smart City Operations and Platforms Engineering*, SCOPE, pages 34–39, New York, NY, USA, 2017. ACM.

[SL08] Gérald Santucci and Sebastian Lange. Internet of Things in 2020 a Roadmap for the Future. In *Joint EU-EPoSS Workshop Report*, 2008.

[SSZ15] Eric Simmon, Sulayman K. Sowe, and Koji Zettsu. Designing a Cyber-Physical Cloud Computing Architecture. *IT Professional*, 17(3), 2015.

[Sta06] OASIS Standard. Reference Model for Service Oriented Architecture 1.0. Technical report, 2006.

[STB17] Andreas Seitz, Felix Thiele, and Bernd Bruegge. Focus Group: Patterns for Fog Computing. In *Proceedings of the 22nd European Conference on Pattern Languages of Programs*, EuroPLoP, New York, NY, USA, 2017. ACM.

[STB18a] Andreas Seitz, Felix Thiele, and Bernd Bruegge. APEP - An Architectural Pattern Evaluation Process. In *Proceedings of the 12th Latin American Conference on Pattern Languages of Programs*, SugarLoafPLoP, New York, NY, USA, 2018. ACM.

[STB18b] Andreas Seitz, Felix Thiele, and Bernd Bruegge. Fogxy: An Architectural Pattern for Fog Computing. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, EuroPLoP, New York, NY, USA, 2018. ACM.

[Ste16] Christine Steinhoff. Aktueller Begriff Industrie 4.0. Wissenschaftliche Dienste des Deutschen Bundestages - Fachbereich WD 8, 2016.

[Sto14] Ivan Stojmenovic. Machine-to-Machine Communications With In-Network Data Aggregation, Processing, and Actuation for Large-Scale Cyber-Physical Systems. *IEEE Internet of Things Journal*, 2014.

[SW18] Dimitrios Serpanos and Marilyn Wolf. *Industrial Internet of Things*, pages 37–54. Springer International Publishing, Cham, 2018.

[Sye17] Asad Ullah Hussain Syed. Deployment and Orchestration of Edge Computing Applications. Master's thesis, Technische Universität München, October 2017.

[Tan09] Wei Tang. *Meta Object Facility*, pages 1722–1723. Springer US, Boston, MA, 2009.

[Thi17] Felix Thiele. Analysis and Evaluation of the Fogxy Pattern for Fog Computing. Master's thesis, Technische Universität München, September 2017.

[TK13] T. Taleb and A. Ksentini. Follow me cloud: interworking federated clouds and distributed mobile networks. *IEEE Network*, 27(5):12–19, 2013.

[VRM14] Luis M. Vaquero and Luis Rodero-Merino. Finding Your Way in the Fog: Towards a Comprehensive Definition of Fog Computing. *SIGCOMM Comput. Commun. Rev.*, 44(5):27–32, 2014.

[VS17] Prateeksha Varshney and Yogesh Simmhan. Demystifying Fog Computing: Characterizing Architectures, Applications and Abstractions. *CoRR*, 2017.

[VWB+16] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos. Challenges and opportunities in edge computing. In *IEEE International Conference on Smart Cloud (SmartCloud)*, pages 20–26, 2016.

[Wan15] Yingwei Wang. Cloud-dew architecture. *International Journal of Cloud Computing*, 4(3):199–210, 2015.

[Wan16] Oliver Wangler. Applicability of Fog Computing in the Context of Industrial Internet. Master's thesis, Technische Universität München, December 2016.

[WB97] Mark Weiser and John Seely Brown. The Coming Age of Calm Technolgy. pages 75–85. Copernicus, New York, NY, USA, 1997.

[Wei99] Mark Weiser. The computer for the 21st century. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3(3):3–11, 1999.

[WF12] Tim Wellhausen and Andreas Fiesser. How to Write a Pattern?: A Rough Guide for First-time Pattern Authors. In *Proceedings of the 16th European Conference on Pattern Languages of Programs*, EuroPLoP, New York, NY, USA, 2012. ACM.

[Woe17] Cecil Woebker. Deployment and Management of Fog Computing Applications using Cloud Technologies. Bachelor's thesis, Technische Universität München, August 2017.

[WSMB18] Cecil Wöbker, Andreas Seitz, Harald Mueller, and Bernd Bruegge. Fogernetes: Deployment and Management of Fog Computing Applications. In *IEEE/IFIP International Workshop on Decentralized Orchestration and Management of Distributed Heterogeneous Things (DOMINOS)*, 2018.

[WYG+17] Z. Wen, R. Yang, P. Garraghan, T. Lin, J. Xu, and M. Rovatsos. Fog Orchestration for Internet of Things Services. *IEEE Internet Computing*, 21(2):16–24, 2017.

[YA03] Sherif Yacoub and Hany Ammar. *Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[YHQL15] S. Yi, Z. Hao, Z. Qin, and Q. Li. Fog computing: Platform and applications. In *Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, pages 73–78, 2015.