Distributed Architecture With Complex Event Processing For Support of Program Comprehension and Debugging In Smart Systems

by

Tobias Peter Eichler

Thesis submitted in partial fulfilment of the requirements
of the University of the West of Scotland
for the award of Doctor of Philosophy

30th November 2022

Declaration

The research presented in this thesis was carried out by the undersigned. No part of the research has been submitted in support of an application for another degree or qualification at this or another university.

Signed: ______________________________

Date: ______________________________

Abstract

This work contributes to smart systems research with a novel software architecture especially designed for the support of the program comprehension and debugging tasks.

The aim of this thesis is to design a software architecture to ease development tasks and allow fast experimentation in smart system research laboratories. The architecture consists of a distributed publish/subscribe middleware, agent-based frameworks for development, and runtime environments for agent execution. Complex Event Processing (CEP) is integrated into this agent-based system in a novel, seamless way for both context handling and system status inspection to assist in program comprehension and debugging tasks.

This work uses a mixed-methods methodology and contains three studies. Firstly, seven expert interviews were conducted to analyse requirements for architectures in smart systems laboratory environments. Secondly, experiments to measure message latency and scalability of the implementation of the designed software architecture. And finally, an evaluation of the architecture using the Architecture Trade-off Analysis Method (ATAM).

In addition to the requirements for software architectures for smart systems from the literature, such as scalability and heterogeneity, the expert interviews show that for laboratory environments, support for program comprehension and debugging is an important requirement. The conducted experiments revealed that the developed software architecture meets the latency and scalability requirements because the system scales with the number of middleware nodes and still has average message round trip times of less than 4.6 ms with up to 10,000 agents. Furthermore, the results of the ATAM showed that the architecture meets all the requirements, both from the literature and the expert interviews. Thus, it can be used to support software development and rapid experimentation in smart system environments.

The results of this thesis could be used in the future to ease smart system research experiments with the presented architecture and the seamlessly integrated CEP engine.

Contents

List of Publications

Eike Langbehn, Tobias Eichler, Sobin Ghose, Kai von Luck, Gerd Bruder, and Frank Steinicke. Evaluation of an omnidirectional walking-in-place user interface with virtual locomotion speed scaled by forward leaning angle. In *Proceedings of the GI Workshop on Virtual and Augmented Reality (GI VR/AR)*, pages 149–160, 2015

Tobias Eichler, Susanne Draheim, Christos Grecos, Qi Wang, and Kai von Luck. Scalable context-aware development infrastructure for interactive systems in smart environments. In *2017 IEEE 13th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 147–150, 2017. doi: 10.1109/WiMOB.2017.8115848

Jonathan Becker, Uli Meyer, Tobias Eichler, and Susanne Draheim. A supernatural vr environment for spatial user rotation. In *2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, pages 850–851, 2019. doi: 10.1109/VR.2019.8798290

Tobias Eichler, Susanne Draheim, Kai von Luck, Christos Grecos, and Qi Wang. Integration of complex event processing into multi-agent systems: two use cases for distributed software development support. *Thirteenth International Tools and Methods of Competitive Engineering Symposium*, 2020

List of Figures

List of Tables

Listings

Abbreviations

AAL	Ambient Assisted Living
API	Application Programming Interface
AST	Abstract Syntax Tree
ATAM	Architecture Trade-off Analysis Method
CAN	Controller Area Network
CEP	Complex Event Processing
CML	Context Modelling Language
CPN	Cyber-Physical Network
CPS	Cyber-Physical System
CSTI	Creative Space for Technical Innovations
DSL	Domain Specific Language
EP	Event Processing
HAW	Hamburg University of Applied Sciences
HCI	Human-Computer Interaction
IDE	Integrated Development Environment
IE	Intelligent Environments
IoT	Internet of Things
JMS	Java Messaging Services

JSON	JavaScript Object Notation
JVM	Java Virtual Machine
MAS	Multi-Agent System
MQTT	Message Queuing Telemetry Transport
ORM	Object-Role Modelling
QCA	Qualitative Content Analysis
RAM	Random-Access Memory
REST	Representational State Transfer
SAAM	Software Architecture Analysis Method
SoS	System of Systems
UML	Unified Modelling Language
VR	Virtual Reality
WIP	Walking-In-Place
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Motivation

In 1991, Weiser described the concept of ubiquitous computing as "specialized elements of hardware and software, connected by wires, radio waves and infrared [that] will be so ubiquitous that no one will notice their presence" [Weiser, 1991, page 1].

Since then, several types of smart systems have emerged in research. The first of these was smart environments, in which users are supported in their lives or work by ubiquitous intelligent components [Cook and Das, 2004]. After that Cyber-Physical System (CPS), consisting of seamlessly integrated physical components and network infrastructure [National Science Foundation, 2021], and the Internet of Things (IoT), consisting of smart devices connected over the internet [Gokhale et al., 2018], arose.

Over time, very large and complex systems developed [Laghari et al., 2022]. This complexity arises primarily from the interconnections of the components [Fortino et al., 2021], context awareness [Bettini et al., 2010] and dynamic reconfigurations [Shi et al., 2011] at runtime. In addition, the interconnection of several of these systems creates a System of Systems (SoS) that adds yet another abstraction layer [Santos et al., 2022]. These complex distributed systems pose a challenge to traditional software engineering methods [Fang, 2021].

Various methods are used to make this complexity manageable for developers. A common method is the use of middleware, which implements the communication layer in the system and thus provides a higher abstraction layer independent of heterogeneity and communication complexity [Deohate and Rojatkar, 2021]. Middleware can include frameworks and toolkits for developers to simplify programming [Henricksen et al., 2005].

Yet questions remain regarding how to check if the system is working as expected and how to locate errors when they occur. With the help of formal design methods, it is possible to model distributed systems mathematically, to establish theorems about the behaviour of the programme and prove them [Beschastnikh et al., 2016]. If the formal specification is then transferred into a programme, it can be ensured that the system functions as expected. However, it can be shown that formal design methods are insufficient even for some supposedly simple multi-agent systems [Edmonds and Bryson, 2004]. Therefore, an experimental approach is often necessary to show that a complex system functions as expected. To do this, it is necessary to improve the methodology and technology for testing and adapting software because this – in contrast to the specification and implementation – accounts for a large part of the work with complex systems [Edmonds and Bryson, 2004].

Furthermore, the growing dependence of systems and their environments creates the need to analyse and comprehend software at runtime [Baresi and Ghezzi, 2010]. For this reason, the traditional separation between the development time and the runtime of a programme is becoming increasingly obsolete [Baresi and Ghezzi, 2010], and tools are needed to understand and modify complex software at runtime. Hence, there is a need for a middleware with a framework and tools for smart systems [Fortino et al., 2021] that can deal with the complexity of the systems, support the development process, and analyse software at runtime.

The main motivation for this thesis is the lack of architectures for smart systems designed to support the development processes of software in laboratory environments. Furthermore, there is a lack of qualitative studies in this area that investigate the interrelationships between the requirements for new architectures. This thesis contributes to the current knowledge on the one hand with a qualitative analysis of requirements for smart system laboratory environments and on the other hand with an architecture that implements these requirements. The availability of such architectures is important because both the interest in and complexity of smart systems continue to grow. Therefore, supporting the research and development of these systems with suitable architectures is important for their future development.

My personal motivation is based on my seven years of professional experience in two research laboratories at the University of Applied Sciences Hamburg. This began with the design of a low-latency middleware for a smart home laboratory, the Living Place Hamburg. The design of the middleware was published as part of my master's thesis [Eichler, 2014] and serves as a preliminary study for the messaging layer of the architecture in this thesis. In the following years, the Creative Space for Technical Innovations (CSTI), an interdisciplinary research laboratory, was established. This is where the approaches and studies developed in this thesis were carried out.

1.2 Research Aim and Objectives

The aim of this thesis is to design a software architecture to improve development tasks in smart systems in general and in smart system research laboratories in particular; this includes both program comprehension and debugging support, and it is intended to ease the development of smart systems and allow rapid experiments for research projects in smart system laboratories.

The research aim of this thesis comprises four research objectives:

The first objective is to identify the requirements for software architectures for smart system laboratory environments. For this purpose, multiple interviews were conducted with several experts who have worked in such software labs (see Chapter 6). In addition, the stakeholders in two software labs were surveyed to confirm the identified requirements (see Chapter 8). The requirements were then used for the design and evaluation of the architecture.

- **O1: Smart Systems Research Lab Middleware Requirements**
  - To conduct a study to determine the requirements for software architectures in smart systems research labs.

The next objective is the design and implementation of the base architectural layer. This layer is responsible for the monitoring of all components as well as the communication between them. For the architecture presented here, a publish/subscribe approach was chosen because it provides a loose coupling between the components and allows easy developers access to the communication for program comprehension and debugging tasks. The publish/subscribe messaging must be scalable and have a low message latency, which was evaluated in two experiments with a large number of agents in Chapter 7. This goal is achieved via non-blocking actor programming techniques (see Section 4.2).

- **O2: Basic System Architecture and Messaging**
  - To design a scalable low latency publish/subscribe-based middleware to improve software development tasks in smart systems.

The third objective concerns the second layer of the architecture, the CEP layer. Complex Event Processing (CEP) allows for the processing of event streams [Luckham, 2002] and is used in the architecture for the processing of context information as well as for the support of program comprehension and debugging tasks (see Section 4.3). The integration was developed using an agent-based architecture and evaluated by multiple case studies as well as a scenario-based evaluation (see Chapter 8).

- **O3: Seamless Integration of Complex Event Processing**
  - To integrate CEP into the system to further improve the software development tasks with minimal added complexity and comparable latency and scalability characteristics.

The fourth objective concerns the third layer of the architecture, the interaction layer. This layer is intended to support developers with program comprehension and debugging tasks through a graphical user interface and to allow them to interact directly with the system using CEP queries (see Section 4.4). The functionality of this layer was evaluated with expert interviews (see Chapter 6) and a scenario-based evaluation (see Chapter 8).

- **O4: System Inspection and Interactive Debugging**
  - To design an interactive tool to analyse the system state of smart systems, including interaction with CEP queries to analyse components and their interactions during development, testing, and runtime.

Together, these four objectives provide three contributions to the research on software architectures and software engineering for smart systems (see Section 9.2). The contributions concern the improvement of development tasks with the design and implementation of the software architecture presented here. This includes, firstly a scalable middleware that explicitly monitors all components and their communication and makes this collected information available to developers, secondly the seamless CEP integration, which provides interactive access to this information for the developers, and finally, the support of program comprehension and debugging tasks in smart systems with the visualisation of the collected data and CEP queries.

1.3 Research Questions

The research objectives lead to the following four research questions:

- **Q1**: What are the requirements for a software architecture in research labs, that uses loosely coupled distributed applications in smart systems?
- **Q2**: How is it possible to create a software architecture that supports development tasks in research labs without compromising the performance of the system?
- **Q3**: How can Complex Event Processing (CEP) be integrated into a middleware for smart systems without increasing the complexity of the system for developers?
- **Q4**: How can such a middleware with an integrated CEP engine support program comprehension and debugging tasks in smart systems?

1.4 Thesis Outline

The structure of the thesis is illustrated in Figure 1.1.

This thesis starts in this chapter with an introduction and the definition of the aim of the thesis, research objectives, and research questions.

Chapter 2 includes an analysis of related work and defines the research gaps that this thesis aims to address. The chapter first explains the process of the literature review, after which, related works to the main topics of this thesis are analysed. These include smart systems, middleware architecture, CEP, and program comprehension.

Chapter 3 presents the research labs in which the results of this thesis were tested. Based on the literature presented in the previous chapter and the evaluations conducted in the research laboratories, requirements for an architecture for smart system laboratory environments are defined.

Chapter 4 presents the middleware architecture that was designed based on the requirements identified in the previous chapter. Firstly, a summary of the architecture is described and then the architecture is presented in detail from the bottom

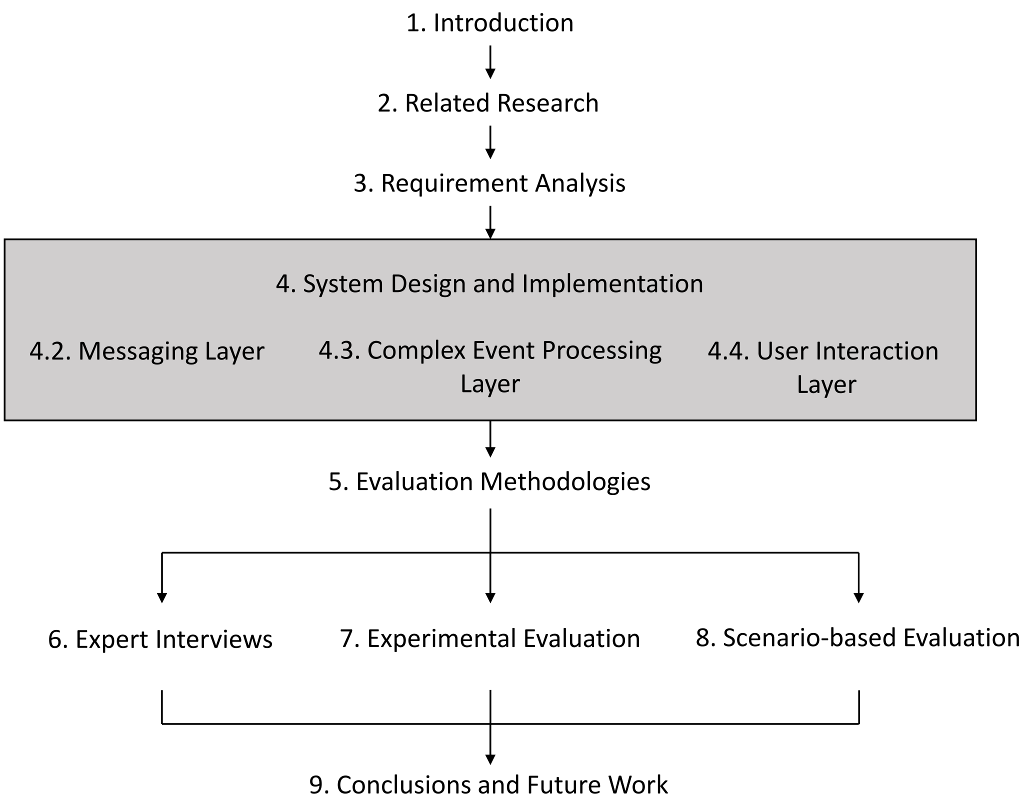

1. Introduction
2. Related Research
3. Requirement Analysis
4. System Design and Implementation
4.2. Messaging Layer
4.3. Complex Event Processing Layer
4.4. User Interaction Layer
5. Evaluation Methodologies
6. Expert Interviews
7. Experimental Evaluation
8. Scenario-based Evaluation
9. Conclusions and Future Work

FIGURE 1.1: Thesis structure

up in three layers, beginning with, the messaging layer and followed by the CEP layer and, finally, the user interaction layer. Important parts of the implementation of the architecture are presented, and, finally two case studies for the support of program comprehension and debugging tasks are described. This includes a sensor network and the construction of an omnidirectional tracking sensor with the help of sensor fusion.

Chapter 5 starts with an explanation of the evaluation methodology for the architecture presented here. This is followed by a literature analysis of the evaluation methods used. Firstly, expert interviews are presented. With the help of this method, the identified requirements for laboratory environments and the transferability of the architecture to other research laboratories are examined. In addition, the Qualitative Content Analysis (QCA) procedure for evaluating the interview results using existing literature is introduced. Afterwards, the Architecture Tradeoff Analysis Method (ATAM), a scenario-based evaluation method for software architectures that is used to evaluate the architecture presented here, is described in detail.

The evaluation of the presented software architecture starts in **Chapter 6** with expert interviews. The interview guideline is presented and the general conditions for the interviews are explained. This is followed by a structured qualitative analysis of the interviews. Finally, the results are evaluated in relation to the research questions.

Chapter 7 contains the experimental part of the evaluation. Firstly, the experiment setup for the latency and scalability measurements is described. Then, the results are presented and evaluated against the requirements.

Chapter 8 concludes the evaluation of the architecture with an analysis of several scenarios. Scenarios are collected and prioritised through a questionnaire for stakeholders. With the help of these scenarios, the architecture is then evaluated in a workshop with the participation of the stakeholders. Finally, the results of the evaluation are presented and assessed with the help of the established requirements.

Finally, a summary of the thesis is given in **Chapter 9**. The main contributions and limitations are detailed, and the chapter also discusses the research implications and makes suggestions for future research.

Chapter 2

Related Research

To achieve the aim of this thesis, the first step is to analyse related literature in relevant areas. This includes an analysis of the various forms of smart systems, their defining characteristics, and common software architectures. Furthermore, Complex Event Processing (CEP) and its use in smart systems is analysed. This technology can be utilized for, among other things, the processing and analysis of contextual information and is therefore relevant for this thesis. In addition, literature from the field of program comprehension and debugging in complex and agent-based systems is examined. Finally, the results of the literature review are summarised, and the identified research gaps are highlighted.

2.1 Process of Literature Review

The literature review for this thesis started based on the literature research for the first paper [Eichler et al., 2017] about a middleware architecture for a smart home laboratory. Given the aim of this thesis, three main topics were addressed in this literature review:

- **Smart Systems** - This research and all of the analysed software relate to smart systems. To support the development of software in this area, the targeted environments and their special requirements have to be understood.

- **Event Processing and Messaging** - Due to the complexity of communication in smart system software, the processing of large amounts of events is an essential part of this thesis. In the following, the handling of complex distributed events and messages is further analysed. Two of the main topics for this thesis are publish/subscribe messaging and complex event processing.

- **Program Comprehension** - Research on program comprehension in other areas and especially in agent-based systems must be considered in this thesis because its main goal is to support the software development process. This includes the analysis of legacy systems and the debugging of errors in running systems.

Figure 2.1 shows the process of the literature review, which is further explained here. Starting with the main topics, search phrases and keywords were chosen to find publications on ACM Digital, IEEE Explore, and Google Scholar.

The following keywords were used in different combinations for the searches in the mentioned digital libraries:

- Smart Environment
- Ambient Assisted Living
- Smart Home
- Smart Object
- Context Handling
- Internet of Things
- Cyber Physical System
- System of Systems
- Middleware
- Debugging
- Program Comprehension
- Complex Event Processing
- Actor/Agent Framework
- Distributed Messaging
- Publish/Subscribe
- Multi-agent Systems

The relevance of the papers was then assessed based on their title, abstract, and keywords. In cases of ambiguity, the introduction and conclusion of the paper were included in the selection process. All relevant publications were then scanned for references recursively that matched one of the keywords or search phrases. Additionally, other publications that were submitted to discovered conferences and journals were included in the search. This also includes recommendations from the double-blind reviews of other published research papers and recommendations from people at attended conferences.

Some topics were excluded from the literature search, including research about real-time processing, security aspects, network protocols, and agent-based simulations. These topics are not the focus of this thesis and are therefore not considered. The selection for exclusion was also made based on the title, keywords, and abstract of the paper.

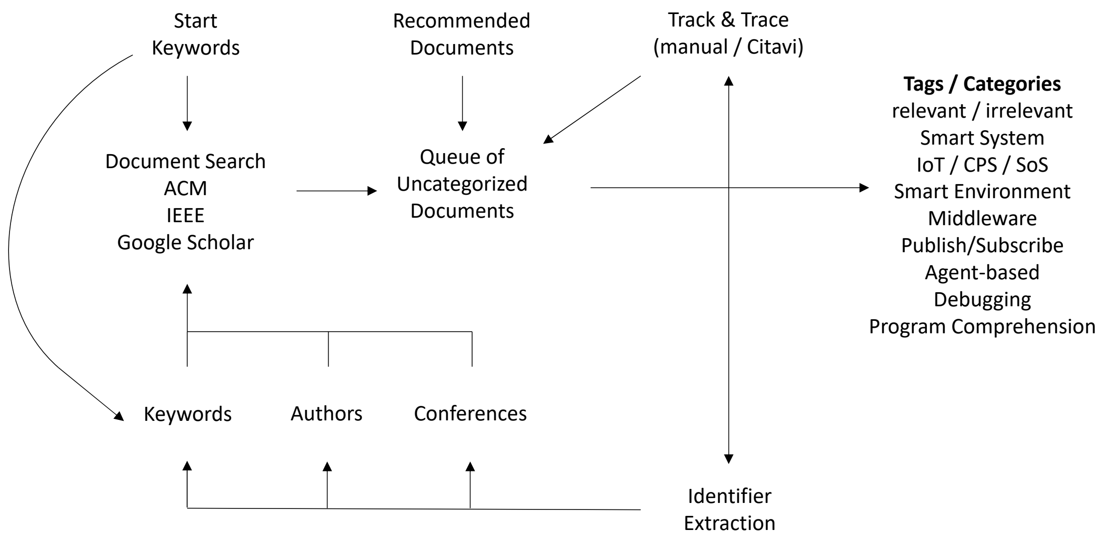


FIGURE 2.1: Literature review process

2.2 Smart Systems

This section analyses the target environment for the architecture. The different types of smart systems and their special characteristics are explained, and then known architectures and middleware approaches in this area are compared. Table 2.1 offers an overview of various smart systems, their properties, and places of use. The table is intended to clarify the definition and differentiation of the various smart system research areas, which are presented one by one in the following. Starting with smart environments, then Cyber-Physical System (CPS), Internet of Things (IoT) and finally System of Systems (SoS).

2.2.1 Smart Environments

A smart environment is defined by Cook and Das [2004] as "a small world where all kinds of smart devices are continuously working to make inhabitants' lives more comfortable" [Cook and Das, 2004, page 3].

Weiser [1991] describes his vision of pervasive embedded computing in his 1991 article stating that "the most profound technologies are those that disappear." [Weiser, 1991, page 3]. He discusses hardware and software components in our daily environments that become so small and integrated into their surroundings, that they disappear from the user's field of attention and become ubiquitous. This forms the origins of ubiquitous computing [Weiser et al., 1999, Poslad, 2009] and since then become reality in our everyday lives.

Coen et al. [1998] refer to these highly interactive environments as Intelligent Environments (IE) and describe a prototype laboratory set up to explore IE called the Intelligent Room. In this lab, natural and multimodal Human-Computer Interaction (HCI), computer vision, and gesture and speech recognition were explored.

The ever-increasing availability of computing power and the ability to fit powerful computing devices inside small form factors make humans the limiting factor in ubiquitous computing environments [Cook and Das, 2004]. The user's time, attention, and decision-making resources are limited and do not scale over time like the power of computing devices. This forces us to think about the HCI to improve the assistance of humans in their environments further. This is the intention of Smart Environment research [Cook and Das, 2004].

According to Cook and Das [2004] common features of smart environments include the following.

	Smart Systems			
Single System		Cyber-Physical Systems	Internet of Things	Smart Environments
Network of Systems	System of Systems	Cyber-Physical Network		
Examples for Typical Use Cases	- education - transportation - smart energy grids - emergency management - e-commerce	- medicine - power grid - intelligent roads - self-driving cars	- smart homes - wearables - Digital Health	- smart homes - smart workspace - Ambient Assisted Living (AAL)
Defining Characteristics	- concept - autonomity - high diversity	- embedded components - real time requirements - constraint resources	- connection over internet - heterogenous components	- component automation - context-aware - decision making - user interaction

TABLE 2.1: Overview of the different smart systems analysed in this thesis. As the boundaries between the individual environments are not clearly defined in the literature, there are overlaps. For classification purposes, this table contains examples of typical use cases and defining characteristics for all environments.

- **Remote Control of Devices** - Remote control of devices is a basic feature of a smart environment. This could be basic appliances, like a coffee machine at home or desk lights at work. A smart context dependent control can free the user of basic interaction task that would otherwise be necessary with older technology like remote light switches.

- **Device Communication** - With today's technology, devices can communicate wirelessly with a large network. This allows all smart devices inside the environment to communicate with each other, which enables interaction between them. This interaction can be utilised to realise more complex tasks and whole scenarios for the user.

- **Information Acquisition/Dissemination from Intelligent Sensor Networks** - Sensors are important to collect context information about the environment. Sensor networks can produce and process huge amounts of data, that can be utilised by smart components to make adequate content dependent decisions.

- **Enhanced Services by Intelligent Devices** - Intelligent devices or smart objects [García et al., 2017], can assist the user with complex tasks. Many devices in human environments can be altered with smart functions. Examples include vacuum cleaners that clean automatically or smart flooring, that detects falls and calls for help. Over the years, many small devices have been developed and have become part of the consumer market.

- **Predictive and Decision-Making Capabilities** - Predictive and context dependent decision-making are important tasks for devices to support users intelligently [Youngblood et al., 2005]. Predictions could be based on simple rules or complex machine learning algorithms. Decisions can be based on context and depend on multiple sources. For example, based on the user's calendar entries and current traffic reports, an application could determine the optimal time to prepare a cup of coffee in the morning.

- **Networking Standards and Regulations** - Various networking standards are an important part of a smart environment, because they are used for communication between devices. Examples of commonly used wireless communication standards are WLAN, Bluetooth, and Zigbee.

Smart environments contain different types of software and hardware components. Two important components are sensors and actors [Cook and Das, 2007]. Sensors collect data about their environment and make this information available for the rest of the system. Actors can perform actions to change their environment, such as the manipulation of physical objects. Automation inside a smart environment can be modelled by a sense-and-act cycle [Cook and Das, 2007]. For example, a temperature sensor can measure the temperature of the room. This information is then used by another component to decide when to increase or decrease the gauge of the heating system, which is controlled by an actor. When the heating control is changed, the temperature changes after some time, and the cycle continues.

Smart environment is thus a collective term for various environments with smart components. The following paragraph details some of the environments frequently described in the literature.

- **Smart Homes** - In smart homes pervasive technology and context-awareness are used to assist people at home [Augusto and Nugent, 2006]. This includes not only context-aware home automation, such as light and window control, but also the support of inhabitants in various scenarios [Meyer and Rakotonirainy, 2003] with the help of intelligent processes [Zaidan and Zaidan, 2020].
- **Ambient Assisted Living** - Research on Ambient Assisted Living (AAL) tries to improve the everyday lives of elderly and disabled people with pervasive technology [Costa et al., 2009]. Smart homes are often associated with AAL research and efforts tho allow older people to continue living independently at home for longer [Suryadevara and Mukhopadhyay, 2015].
- **Smart Workplaces** - Smart workplaces are working environments that are enhanced with pervasive technology. The aim is to facilitate both routine and

specialised tasks in a natural and intuitive way for the user [Mikulecky, 2012]. These environments include smart offices and smart classrooms. In addition, ambient assisted working is a research area that deals with the support of people with disabilities in the working environment [Bühler, 2009].

- **Smart Cities** - The term *smart city* refers to projects to improve life in cities through technology and to make them more efficient [McClellan et al., 2018]. An important aspect of this is to improve sustainability, such as by reducing the consumption of natural resources and the emission of harmful substances such as CO2. These projects can concern areas intelligent power distribution, intelligent passenger transportation or public security. Smart cities combine several aspects, such as smart living, smart economy, and others. This creates very large, ver complex networked systems [Zygiaris, 2013].

2.2.1.1 Context-awareness

One of the key aspects of a smart environment is context-awareness [Cook and Das, 2007]. Context information is gathered by sensors and can be processed ad hoc or be stored in and later queried from databases. Context-awareness was first mentioned by Schilit and Theimer [1994] as applications that are aware of the location of the user and nearby objects that adapt their behaviour to mobile users. This definition of context was later broadened by Abowd et al. [1999] to the following: "Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves" [Abowd et al., 1999, page 4-5].

Information about the context of an application can be used to simplify and improve the user interaction [Abowd et al., 1999], which can reduce the amount of user attention needed to support humans with their tasks [Cook and Das, 2007]. This is what makes a component act smartly in the current application context [Cook and Das, 2007].

The use of context information for decision-making in applications can be challenging for the comprehension of a programme and the development process. One of the reasons is that context information is often not reproducible and can be very complex. At the time of writing code or debugging of a programme, it is often not clear what the context is or will be in the future. This high complexity also affects distributed context-aware systems, which in addition are often heterogeneous [Henricksen et al., 2005]. As a result, components often use different hardware and communication methods and are written in different programming languages.

There are different ways to model contextual information and make decisions based on this information [Bettini et al., 2010]. Challenges can be the quality as well as the quantity of the data [Bettini et al., 2010]. This is due to the potentially high number of sensors in a context-aware system. All these sensors may have measurement errors, or individual measurements may be lost during transmission. Early approaches to context-aware systems were based on key-value databases or XML-based context descriptions [Bettini et al., 2010]. However, these approaches had only limited possibilities for modelling dependencies, relationships, and temporal sequences [Bettini et al., 2010]. Later, a Context Modelling Language (CML) was developed based on the ORM approach to database modelling [Henricksen and Indulska, 2004]. Object-Role Modelling (ORM) is a data modelling technique used to model real world objects and their relationships to each other. For this purpose, the relationships between objects are described with simple sentences or diagrams, where the objects are the nodes and the relationships are the edges. In CML, context facts are represented as entities with different classes. Relationships between entities are modelled similarly to ORM. There are further additions in CML compared to ORM. This includes support for dependencies, imperfect sensor data, and the representation of past values. Queries regarding the data representation can be made with an SQL-like language [Bettini et al., 2010].

One advantage of CML is that it supports the modelling of contextual information in several phases of software development. However, not everything can be modelled in CML. For example, hierarchical structures cannot be modelled [Bettini et al., 2010]. In addition, it is difficult to achieve interoperability with other systems [Bettini et al., 2010] because the entire system must be modelled in CML in order to be able to query all data.

An alternative is modelling with the help of ontologies, such as the Web Ontology Language (OWL[1]) [World Wide Web Consortium, 2012]. The ontology-based approach has advantages in the area of heterogeneity and interoperability [Bettini et al., 2010]. However, this approach also requires a complete modelling of the context information in the description language.

Hybrid approaches have emerged that combine ontologies with CML features and combine their advantages [Bettini et al., 2010]. However, these approaches also require that all components use the prescribed model or that a complete model exists that represents all contextual information.

This can be a problem, especially for environments with fast development cycles and prototyping, because modelling can become complex and may require additional knowledge from the developers. Therefore, none of the modelling techniques are used for the architecture developed here. A more flexible and much more informal way of processing contextual information is through CEP. CEP and its application in smart systems for context processing is of particular interest for this thesis and is described separately in Section 2.4.

2.2.2 Cyber-Physical Systems

Cyber-Physical Systems (CPS) integrate computation seamlessly with physical components [National Science Foundation, 2021]. These components can be actuators, sensors or network infrastructure.

CPS have the following defining characteristics [Shi et al., 2011]:

- CPS are **closely integrated** with physical processes.

[1]Actually WOL, but the World Wide Web Consortium (W3C) chose OWL as the acronym.

- The physical component and embedded systems which contain the software have **restraint resources**, such as computing power and network bandwidth.
- CPS components **communicate over potentially large networks** with protocols such as WLAN, Bluetooth, and mobile networks.
- The components of a CPS have potentially **different granularities of time** and are **constrained by spatiality and real-time** requirements.
- CPS are adaptive and **can dynamically reorganise and reconfigure themselves** to adapt to specific circumstances.
- CPS support human interaction with the system through a **high degree of automation**.
- CPS should be **reliable and secure**.

Human interaction with the system is not mandatory but can also be part of a CPS. The typical applications of CPS are manifold; they are used, for example, in medicine, electrical power grids, intelligent roads, and self-driving cars [Shi et al., 2011]. Other areas of use include agriculture, education, energy management, and environmental monitoring [Chen, 2017]. In addition, CPS are used for Smart Systems, such as smart cities, smart homes and smart manufacturing Chen [2017].

CPS can become very large and complex as a result of networking on a large scale and interaction with the physical world through, for example, sensors. Therefore, failure detection is an important issue for CPS [Alippi et al., 2017].

By connecting multiple CPS over the network to achieve a common goal, a Cyber-Physical Network (CPN) is created [Brinkschulte et al., 2019]. Developing CPS and CPN is challenging due to their complexity [Brinkschulte et al., 2019]. The complexity of CPN arises, among other things, from the dynamic behaviour of the system, changing environmental conditions, and strict latency requirements [Brinkschulte et al., 2019, Wang et al., 2019]. As such, adapted middleware systems are necessary for CPS and CPN to reduce this complexity for the developers.

2.2.3 Internet of Things

The term Internet of Things (IoT) was first proposed by Ashton in 1999 [Ashton, 2009, Gokhale et al., 2018]. It describes the connection of physical devices and other objects via the internet [Gokhale et al., 2018]. An IoT device can be, for example, a tool, vehicle, building, or other object that is connected to a network by means of electronics and that can thus collect and exchange data with other devices.

The IoT is widely used for several applications. These include smart urban cities, connected automobiles, wearables, smart retail, digital health, and more [Laghari et al., 2022]. IoT components are also often used within smart systems, such as in smart environments, smart homes, smart cities, smart agriculture, and smart parking [Sathish and Smys, 2020]. In smart homes, IoT devices can be used as sensors and actuators to enrich the smartness of the environment and to monitor and control home appliances [Williams et al., 2019].

A variety of protocols can be used for IoT systems. These include many widely used communication standards such as WLAN, ZigBee, Bluetooth, RFID, NFC, and mobile network standards such as 3G to 5G [Li et al., 2015]. Messaging protocols, such as MQTT and others, are also often used [Yassein et al., 2017]. The Message Queuing Telemetry Transport (MQTT) protocol [OASIS, 2019-03-07] allows for communication with other components and middleware based on a publish/subscribe pattern. One advantage of MQTT is that it is supported on many platforms and is also suitable for use on resource-constrained devices. These and other properties mean that MQTT is often used in IoT applications. As a result, these systems can use group communication and thus achieve a loose coupling between the components.

Because of this message-based communication and the heterogeneity of the IoT components, agent-based computing is well suited for programming these systems [Savaglio et al., 2017]. Each IoT component can be implemented as an agent, and components can be addressed as agents via software adapters [Takahashi et al., 2005]. The latter has the advantage that the component itself does not have to implement the functions of an agent, which is particularly helpful for resource-constrained devices.

Among other challenges in IoT, such as security [Ammar et al., 2018], the complexity of IoT systems is currently one of the known disadvantages of the utilised architectures [Li et al., 2015, Laghari et al., 2022]. This can cause problems in the development of new systems and lead to errors. One way to reduce this complexity for the developer is to use of a middleware to manage the system [Razzaque et al., 2016] or a framework for development.

The development of IoT systems can be challenging for many reasons. These include the lack of a high level of abstraction to handle large systems, to deal with the huge heterogeneity in the field and the handling of life cycles [Patel and Cassou, 2015]. Here, frameworks can be used to help deal with these problems at a higher level of abstraction. Patel and Cassou [2015], for example, use a Domain Specific Language (DSL) to automate tasks for IoT devices and the process of integrating components.

There are currently several open research issues regarding IoT, including concerns related to a standardised IoT architecture, improvement of the user experience for IoT applications, and the improvement of security for more sensitive areas, such as health or automotive systems [Laghari et al., 2022]. This shows that there will be a need for IoT research systems and laboratory infrastructure to address these issues. Improvements in the development processes of IoT applications in such research environments could support further research in this area.

According to Savaglio et al. [2017], the full realisation of the potential of IoT is not hindered by hardware constraints or software performance limitations, but by other requirements that have not been adequately addressed yet. Agent-based computing can help address problems in the modelling, development, and simulation of IoT systems [Savaglio et al., 2017]. This simplifies and accelerates the development process for IoT systems and reduces the probability of errors, indicating that improving the development of agent-based systems can also be useful for IoT systems in the future.

2.2.4 System of Systems

A system consists of a set of entities that are related to each other. This interconnection creates an added value so that the system is more than the sum of its parts [Boardman and Sauser, 2006]. In order to continue to understand a system as its complexity grows, it is helpful to think not only about a particular part of the system, but about the whole. System thinking [Arnold and Wade, 2015], a common language, and a framework for sharing knowledge about complex systems are necessary. Since many of today's societal, economic, and environmental challenges have characteristics of complex systems [Plate, 2010], it is important that new or improved ways to think about complex systems are found. Plate [2010] posits that it is therefore also important in education to provide students with assistance in understanding complex systems.

A System of Systems (SoS) is the interconnection of multiple systems. SoS have special features compared to normal systems, due to their composition based on many systems [Boardman and Sauser, 2006]. The individual systems are often already existing systems that are connected to each other, producing an increased diversity within the SoS compared to a normal system. Another difference is the autonomy of the components. In a SoS, each individual system is expected to be designed to act fully autonomously. In comparison, within a system, it is common for individual components to subordinate themselves to the autonomy of the whole system.

Software and hardware components that communicate with each other can also form complex systems and SoS [Nielsen et al., 2015]. For example, the treatment of a person in a hospital depends, on the functioning of many individual systems such as the telephone system, the hospital management system, the patient database, and many other independently developed and maintained technical components. The individual systems were likely developed at different times, when there may have been different requirements for the hospital.

In engineering, SoS can be used for a wide range of application domains [Nielsen et al., 2015]. These include, for example, transportation, smart energy grids, emergency management, and e-commerce. An IoT system is also a SoS [Fortino et al., 2021] because it consists of heterogeneous, independent components that were often developed independently of each other. Additionally, CPS evolve from single independent systems into a network of collaborating systems, which is considered a SoS [Nousias et al., 2021].

The characteristics of SoS pose a challenge to traditional software engineering methods [Fang, 2021] that requires innovative methods and architectures. The current challenges in working with a SoS arise from its complexity and special properties [Santos et al., 2022]. For example, predicting the behaviour of a SoS is challenging because the behaviour depends directly on the properties of the individual systems. In addition, it is difficult to prepare suitable documentation for the system, especially during the development phase of a SoS, because SoS continue to evolve and because the initial complexity of the individual systems can be high.

Hence, that more research is needed to support the development and maintenance of complex systems like SoS, including the construction and investigation of complex systems in laboratory environments.

2.3 Smart Systems Middleware

A middleware is used in smart system architectures to make the high complexity of the systems more manageable and to manage the coordination of and communication between the distributed components [Henricksen et al., 2005]. The previous literature review has demonstrated that in smart environments, IoT, CPS and

SoS, the complexity of the systems is a challenge [Fortino et al., 2021]. Furthermore, research interest in this area remains high, and larger systems have been developed and investigated over the years [Laghari et al., 2022]. SoS and CPN are good examples of how the degree of interconnection between systems is further increased by connecting systems to each other. This creates even more complex systems requiring special software architectures that benefit from middleware.

In addition, traditional software engineering methods reach their limits when it comes to the development and maintenance of these complex systems [Fang, 2021]. For this purpose, tools are needed to support developers in their tasks, and simplify both debugging and program comprehension tasks [Fang, 2021]. These tools are often also part of the software architecture and depend on the capabilities of the middleware. Since the requirement for development support is central to this thesis, the analysis of comparable middleware architectures focuses on publications that support the development process.

The following sub-sections present a general middleware architecture and address two aspects of architecture that are important for smart systems and this thesis in particular: agent-based programming and publish/subscribe messaging. Afterwards, comparable related research is analysed and compared with the approaches in this thesis.

2.3.1 Generic Middleware Architecture

Henricksen et al. [2005] conducted a review of several middleware solutions. They defined a general architecture for middleware for context aware systems as follows.

The components of a context-aware system can be separated into five layers, as shown in Figure 2.2. Sensors and actors are in Layer 0. They sense context information and can perform actions to change the environment. Layer 1 includes the context processing components. This layer contains components that assist with the processing of sensor information and components that map update operations to the actors in the layer below. The context repositories in Layer 2 provide persistent storage of context information and assist with queries of contexts. Layer 3 contains decision support tools, which provide functions to select sensor information and actions based on the stored context information. The application components are at the top of Layer 4. Furthermore, programming toolkits that assist with the development of application components are also located in the top layer. They provide functions that support interaction with the components of the other layers. Layers 1-3 are called middleware.

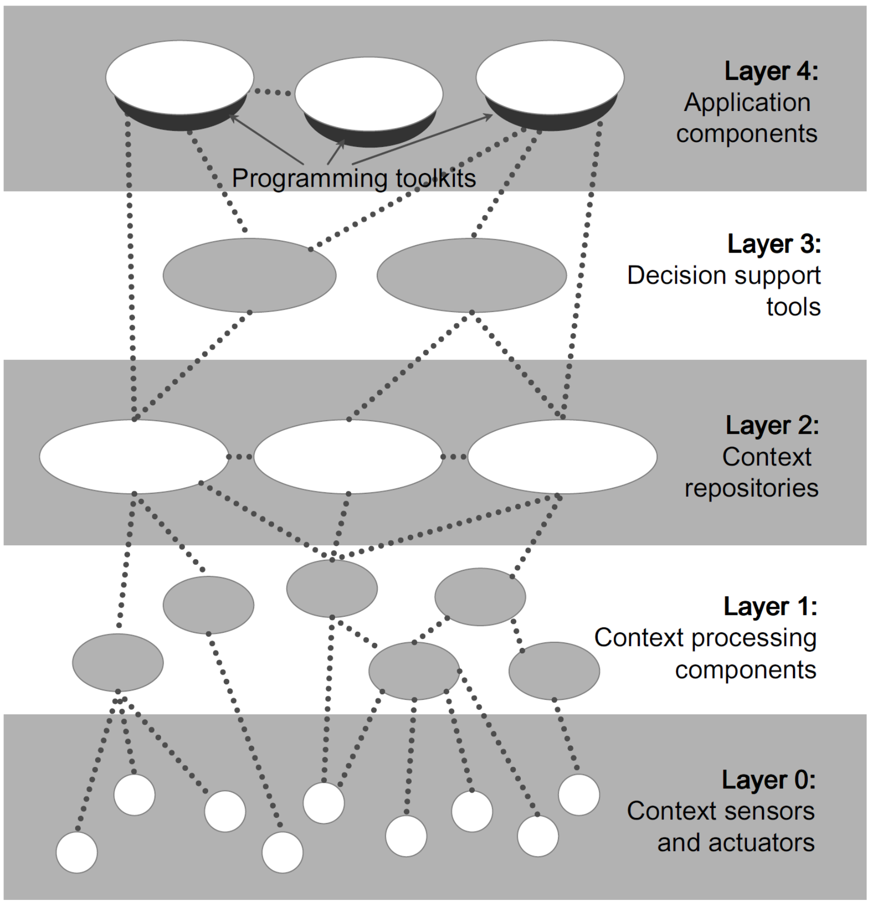


FIGURE 2.2: General middleware architecture Henricksen et al. [2005]

2.3.2 Agents and Actors

Smart systems are often developed using agent-based architectures [Cook, 2009, Savaglio et al., 2017]. Such systems commonly consist of a large number of software components that communicate with each other. The individual components can be regarded as intelligent agents as soon as they have some autonomy and follow an intelligent design [Cook, 2009].

The term *agent* is not completely uniformly defined. Wooldridge and Jennings [1995] give a definition of weak agency that is, by their own account, relatively uncontroversial. According to this definition, an agent is a hardware- or software-based computer system with the following properties:

- **Autonomy** - Agents can act without direct human intervention and have at least partial control over their actions and internal state.
- **Social ability** - Agents can communicate with each other via an agent communication language.
- **Reactivity** - Agents can perceive their environment and react to changes.
- **Pro-activeness** - Agents can not only react but also act pro-actively according to their own goals.

Depending on the definition, other properties can also play a role for agents [Wooldridge and Jennings, 1995]. These include, for example, mobility: the ability to change the location of the execution of a system component at runtime. Furthermore, there is a stricter definition according to which agents, especially intelligent agents, must be computer systems that are augmented with characteristics that are more commonly attributed to humans [Wooldridge and Jennings, 1995]. These can include knowledge, beliefs, intentions, obligations [Shoham, 1993], or emotions [Bates et al., 1994].

Another definition from Russell and Norvig [2016] states that an agent is anything that can perceive its environment through sensors and that can change the environment with actuators. A visualisation of this definition is shown in Figure 2.3. In that case, both humans and robots are agents. They can perceive their environment through their eyes and cameras and change the world around them through muscles and motors. This definition also applies to software agents, where the environment would be, for example, a graphical user interface or communication with another agent, and the change in the environment could be a display on the screen or a message to other parts of the system.

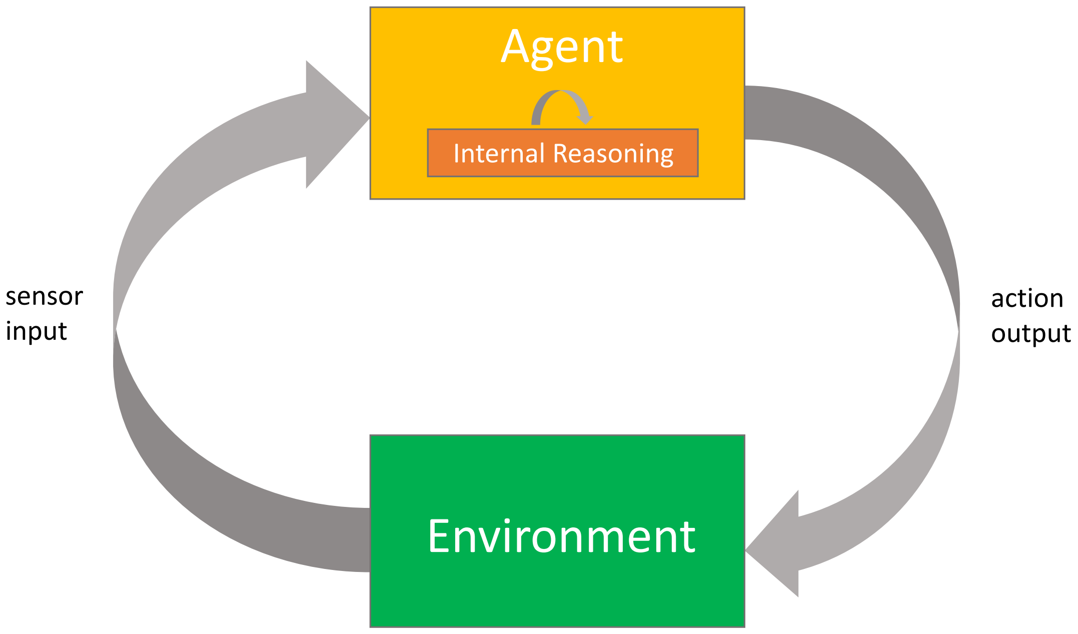


FIGURE 2.3: Agent sense reason act cycle

In this thesis, both the definition from Russell and Norvig [2016] and the properties of agents [Wooldridge and Jennings, 1995] are used in combination, because the overall definition is flexible and can therefore be applied to all components of a smart systems and the properties can be used to explain the behaviour of an agent. This definition also fits with Cook and Das [2007] description of the behaviour of composites in a smart environment using a sense and act cycle.

A predecessor of agent-based programming is the actor model [Wooldridge and Jennings, 1995]. The actor model is a mathematical theory in which actors can be used to describe concurrent processes [Hewitt and Baker, 1978]. The actor model can be used both as a framework for the theoretical understanding of concurrency and for the implementation of concurrent systems [Hewitt, 2010]. In such systems, actors can send messages to other actors via addresses. This triggers an event at the receiver, which can send further messages, change its state, or create new actors.

The term *actor* is used for different things in related work. In smart environments, components that are used to change their environment are called actors. In contrast, the term *actor* used in this section stands for a reactive software unit that can react to messages in the system. In practice, there are different ways to implement actors [Ricci, 2014]. In object-oriented languages such as Java or Scala, actors are often modelled as objects that implement one or more receive methods [Ricci, 2014, Lightbend]. These methods define the behaviour when a message arrives. A change in the behaviour of an actor can be implemented by switching to another receive method [Ricci, 2014].

In contrast to this event-based reactive implementation of actors, agents are mostly implemented on the basis of control loops [Ricci, 2014]. The control loop consists of the sense, reason, and act actions defined above and is repeated in a loop. Agent-based programming languages usually have a higher level of abstraction than actor frameworks. For example, agents can be implemented using goals, beliefs, and plans. Goals define the tasks of an agent, beliefs consist of facts and rules that represent the state of the agent and plans define the process when an event occurs. Both changes to the beliefs and the creation of new goals or the failure of goals can serve as events. This control-loop model allows activations and processes to be implemented more naturally [Ricci, 2014]. However, more complex programming structures are needed for implementation, where for example each plan is held in a stack.

Multiple agents in a distributed system are called a Multi-Agent System (MAS) [Dorri et al., 2018]. Each agent in this system should have a simple and distinct function it fulfils, and complex tasks are implemented by groups of communicating agents [Dorri et al., 2018]. External systems and other entities can be encapsulated by agents to allow communication [Takahashi et al., 2005].

Because all agents need to communicate this functionality is often implemented separately as a fundamental part of the software architecture. The software component responsible for the communication layer is often called middleware, and it can provide additional services, such as discovery, monitoring, or task scheduling

[Gazis and Katsiri, 2022]. A middleware can help handle the complexity of MAS by providing the fundamental service to the rest of the system [Gazis and Katsiri, 2022].

The architecture presented in this thesis is also agent-based and has the aim of reducing the complexity of the system by using individual components with dedicated tasks, and communication via messages. In addition, features of actors are used for the implementation of the agents to make them easy to programme and resource-efficient.

2.3.3 Publish/Subscribe

Erman et al. [1980] presented the Hearsey-II Speech-Understanding System in 1980, which uses a global database called blackboard for communication. All hypotheses generated by the knowledge sources, and independent interpretation programs, are recorded on this blackboard. Each knowledge source can publish new records or modify existing ones. This may satisfy the preconditions of other knowledge sources, that can read the stored information, process it, and then add more information to the global state. The blackboard is divided into multiple levels based on the steps of the language decoding process. This ensures the basic structuring of global knowledge. The blackboard architecture introduces a global state, which is no problem for a single-threaded application, but in a distributed system this is not desirable, e.g., to avoid race conditions and deadlocks.

In contrast, event-based systems are better suited for distributed systems. An event-based system is "a system in which the integrated components communicate by generating and receiving event notifications" [Fiege et al., 2002, page 4]. Events are any "transient occurrence of a happening of interest" [Fiege et al., 2002, page 4]. This could be information from a sensor or a state change within a component. When an event occurs, components can be notified by a message, that contains the relevant data about the event. Messages can be delivered over varying communication methods, depending on the application architecture. An event can produce multiple notifications containing different data depending on the needs of the receiver or security or privacy restrictions. Event notification services are responsible for the management of subscriptions from components that want to receive events matching a defined criteria. The notification service uses this subscription data to notify all components when an event they subscribed to occurs. Components can publish events via messages and consume them through subscriptions. As such, each component can act as an event producer and event consumer.

There are three types of subscriptions with increasing expressiveness [Fiege et al., 2002]:

- **Subject-based** - A subject-based subscription is the simplest type of subscription utilising a additional subject field that is added to each message and that can then be used as a filter. For example, all messages regarding location changes could have the subject *location_event*, and a subscriber could then register their interest by applying for all events with that subject.
- **Type-based** - A type-based subscription utilises type definitions (e.g. in the form of pre-defined data classes). This allows filters to be defined to match specific types or subtypes of events. Additionally, a filter can defined for known fields of the expected class. One example would be a location event that triggers when a person is seen in a specific location. This could be modelled as *LocationEvent('Conference Room', 'Person 1')* and subscriptions could, for example, filter all LocationEvents regarding *Person 1* and be notified when this person moves. An example of a type-based subscription model is given in Bates et al. [1998].
- **Content-based** - A content-based subscription contains a boolean expression that is evaluated based on the message content. This form of subscription is the most expressive but also the most challenging to implement.

There are different techniques for implementing an event-based system with the behaviour defined above. Some examples are publish/subscribe, IP multicast, and tuple spaces [Fiege et al., 2002]. Publish/subscribe systems are defined by the two actions they provide for clients to communicate: publish and subscribe. The publish action sends a message, that can include, depending on the system, meta information. With the subscribe action, a client can signal their interest in certain messages. All messages matching the defined criteria are then sent to the client. Publish/Subscribe is a common communication paradigm in smart systems and has still been used in recent related architectures. As shown for example in a study by Mishra and Kertesz [2020] analysing research publications about the usage of the MQTT messaging protocol in IoT systems. Their study determines that MQTT is one of the most widely used IoT protocols.

Additionally, beyond the two naming operations publish and subscribe, there are two other operations that are common for publish/subscribe systems. These are unsubscribe and notify [Fiege et al., 2002]. Unsubscribe counteracts the subscribe operation and causes no more messages matching defined criteria to be sent to the client. Notify is a method that is implemented by the client and which is called by the system to deliver a message that matches at least one of the criteria defined with a subscribe operation. Notify is a callback method that can be realised differently depending on programming language used and may be handled with anonymous functions.

Publish/Subscribe messaging is compatible with agent-based systems. It can provide a loose coupling between the communicating components because it does not require a component that sends a message to know its recipient [Fiege et al., 2002]. Messages are no longer sent directly from one agent to another, meaning one of the agents needs to know the address of the other one, but they are rather sent to messaging topics. An agent can send arbitrary messages to as many topics as it likes. To receive messages an agent is interested in, the agent has to subscribe to the appropriate groups. For these reasons this thesis focuses on publish/subscribe architectures.

2.3.4 Middleware Architectures

This sub-section identifies related work on middleware architectures and compares these with the approaches in this thesis. Since there are a large number of middleware architectures for different environments and with different goals, this section is focused on a certain selection of architectures that are comparable to the approaches in this thesis.

The following two sets of selection criteria are used for this purpose. The necessary criteria include:

1. **Research context** - Only architectures that originate in a research context are considered. This restriction is necessary to establish comparability since the architecture developed here is primarily aimed at use in research laboratories and this thesis also belongs to the research context.

2. **Development support** - The support of development processes in smart systems is the focus of this thesis. Therefore, comparable work must include methods to support developers. This can be, for example, a visualisation of the system or the provision of a framework, debugger, or Integrated Development Environment (IDE). Sometimes this is also referred to as support for fast experimentation or rapid prototyping.

The optional criteria include:

1. **Agent-based** - The architecture should be agent-based. It has been shown that agent-based architectures are often used in smart systems and that, among other things, they can make the complexity of these systems more manageable (see Section 2.3.2. This criteria, is helpful for comparison because the approach to development differs for other programming paradigms, such as a pure event-based approaches, which would make a comparison of architectures more difficult.

Author	Environment	Ab	P/S	Evaluation
Ranganathan and Campbell [2003]	Ubiquitous Computing Environments	x		-
Gu et al. [2005]	Context-aware Systems	x	x	performance measurements
Henricksen et al. [2005]	Context-aware Systems		x	case study
Soldatos et al. [2007]	Context-aware Systems	x	x	simulations
Bader et al. [2010]	Smart Environments	x		-
Fortino et al. [2013]	Smart Objects	x	x	case study
Maciel et al. [2015]	Smart Environments	x	x	systematic review
Soldatos et al. [2015]	IoT	x	x	case study
Pahl and Liebald [2019]	IoT	x	x	performance measurements, user study
Elhabbash et al. [2020]	SoS	x		case study, performance measurements
Elhabbash et al. [2022]	IoT		x	user study
Borges et al. [2023]	IoT		x	performance measurements
this thesis	Smart Systems	x	x	performance measurements, expert interviews, scenario-based evaluation (ATAM)

TABLE 2.2: Comparison with other middleware architectures that include developer support and are agent-based (Ab) and/or support publish/subscribe (P/S) communication.

2. **Publish/Subscribe messaging** - Another criterion is that the architecture should support publish/subscribe communication. Publish/subscribe communication is an important design decision of the architecture presented here and according to previous the literature analysis also often used in smart system architectures.

All publications describing a middleware architecture identified as relevant for this thesis using the process described in Section 2.1 were sorted according to the four criteria mentioned above. After that, all publications that did not fulfil the first two criteria (research context and development support) and at least one of the second criteria (agent-based, publish/subscribe) were sorted out. Table 2.2 contains an overview of all remaining publications.

Only a few of the resulting publications focus on developer support, such as the short paper by Bader et al. [2010] based on experience in research and teaching, but this contains no evaluation of the architecture. The architecture is based on tuple spaces and is designed to support debugging and to allow easy access to human readable communication between components [Bader et al., 2010]. Elhabbash et al. [2020] and Elhabbash et al. [2022] present a method to support the developer with the composition of multiple IoT components into a SoS, but it is based on ontologies and requires a compatible specification of all components. This requires additional know-how when working with the system. However, these studies do not examine support for debugging. In Borges et al. [2023] IoTvar, a middleware that supports the development of IoT components by abstracting and simplifying the connection to other IoT platforms, is presented, but, the support for program comprehension and debugging tasks is not evaluated.

Henricksen et al. [2005] define *ease of deployment and configuration* as one of seven requirements after evaluating middleware architectures, which is evaluated along with the other requirements via a single case study. Ranganathan and Campbell [2003] state that supporting developers is one of the main goals of their ontology-based middleware and that it supports rapid prototyping, but this claim is based only on the implementation of several agents using the middleware. In Gu et al. [2005] rapid prototyping is also mentioned as one of the features of the presented service-oriented middleware for context-aware services, but the middleware is only analysed with regard to its performance and not for its usability.

Only Pahl and Liebald [2019] describe a user study to evaluate the usability of the system. For this, 150 students were asked to implement a feature within the architecture. The time for implementation and the length of the code were evaluated. They were then asked whether they found the system easy to use and how easy the task was for them. This study demonstrated that the system is easy for students to use for the purpose analysed, but the requirements of other stakeholders or the use for other purposes were not analysed. Elhabbash et al. [2022] also include a user study with 26 participants, measuring the coding time and coding correctness of a given workflow with and without their architecture. Maciel et al. [2015] are the only ones to use a structured approach to evaluating the architecture, but they do not specify a method for doing so. During the evaluation, selected requirements are compared with architectural approaches to measure the suitability of the architecture. All other publications either do not contain an evaluation of usability for developers or justify the suitability of the architecture with case studies that are not evaluated according to a documented or structured method.

Notably, comparable approaches often focus more on system performance, despite the usability features that are sometimes emphasised. The qualitative approaches for the evaluation of middleware architectures are based on the analysis of one or more case studies, without defining the term *case study* or specifying a method. Soldatos et al. [2015] write here of "proof-of-concept applications". The suitability of the architecture is primarily shown by the author through the implementability of individual agents or scenarios.

The publications listed here are based on different technologies. Gu et al. [2005] and Henricksen et al. [2005] use Remote Procedure Calls (RPCs) for their communication, and Bader et al. [2010] use tuple spaces. Some of the architectures use open source agent-based middleware, for example, Fortino et al. [2013] use JADE, a FIPA2000 compliant agent development environment written in Java, which simplifies the development of multi agent systems [Bellifemine et al., 2001]. Moreover, most of the IoT architectures, like Elhabbash et al. [2020], Elhabbash et al. [2022] and Borges et al. [2023] use MQTT or are compatible with it. However, some, like Soldatos et al. [2015] or Pahl and Liebald [2019] use custom protocols.

In summary, it can be said from the analysis of the literature that there is a lack of a structured approach to the evaluation of usability aspects of architectures that focus on the support of developers in smart systems. Furthermore, it becomes apparent that the support of developers in these systems is often not a central requirement, and that usability aspects are less studied in the literature than, for example, performance requirements. This is also a research gap for software architectures in the field of smart systems.

2.4 Complex Event Processing

Event Processing (EP) in general is a paradigm in which streams of events are processed to make inferences about events in the real world [Dayarathna and Perera, 2018]. One of the challenges is the number of events that need to be processed, but distributed EP platforms can handle thousands to millions of events per second

[Dayarathna and Perera, 2018]. Complex Event Processing (CEP) is a subset of EP and can be utilised to process (simple) events into (more complex) events and reactions based on a defined rule set [Luckham, 2002]. The characteristic of CEP is the possibility of recognising patterns that describe complex event constellations.

The abilities of the CEP-system depend on the implementation, but there is a set of common features that are implemented by most systems. Rules can detect specific events or event groups based on their attributes or types, and they allow events to be combined, filtered and modified. Often, events can be detected based on timing, such as, to detect if a specific event happens before another.

Queries to the system are often realised with SQL-like languages [Dayarathna and Perera, 2018] that allow a descriptive specification of what is to be done. Listing 2.1 shows an example of a CEP query that reads all temperature events of a sensor and outputs all temperatures above 30°C together with the location of the sensor.

```
SELECT temp_sensor.celsius, temp_sensor.location
FROM temp_sensor
WHERE temp_sensor.celsius > 30
```

LISTING 2.1: CEP query that selects all temperatures from a sensor higher than 30°C

The basic functionality of a CEP engine is visualised in Figure 2.4. CEP queries are processed by the engine and stored as patterns and rules. The engine monitors one or more streams of events (shown here as a large arrow) in which events are sent. If a pattern is detected based on the current events in the stream, the associated events are processed by filtering them with the specified logical expressions in the query and executing any transformations specified in the query. Finally, the reactions are executed, which allows the generation of new events that can be published to the event stream. The CEP engine is a stateful application and can save the occurrence of certain events if required. This is necessary to realise time windows and patterns with temporal dependencies between events.

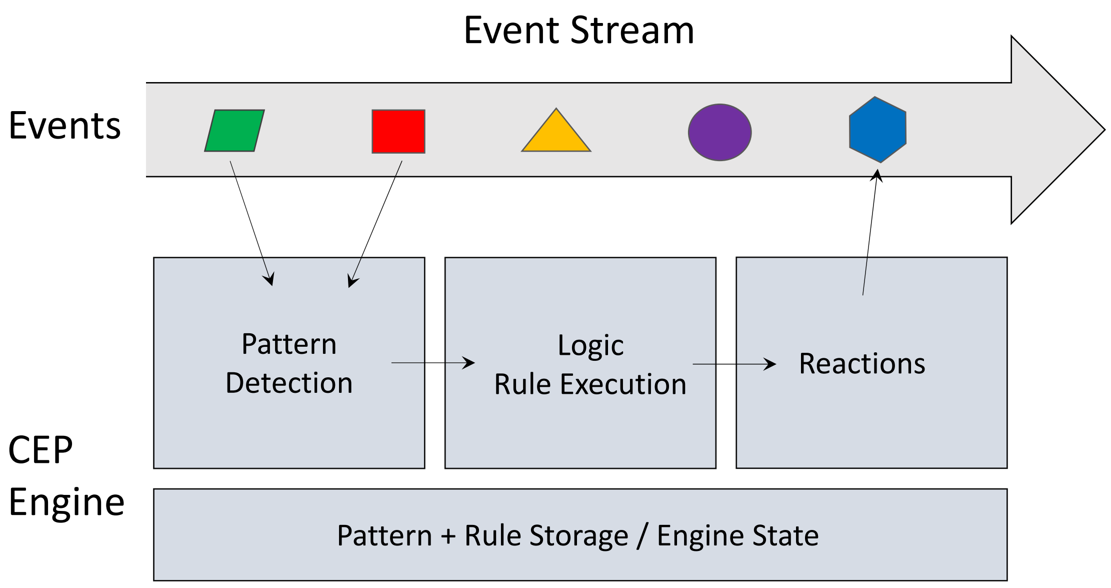


Figure 2.4: Complex Event Processing Architecture

CEP can be used in distributed systems to process events. It is also possible to use CEP as a diagnostic tool for the events in these systems [Luckham and Brian, 1998], which makes it possible for the user to dynamically adjust which events are relevant to him and which events should be included in the processing.

2.4.1 Event Processing Reference Architecture

Paschke and Vincent [2009] propose a reference architecture for event processing-based systems which consists of six layers (Figure 2.5). The use of a common reference architecture allows it to compare different architectures for EP-systems.

In this architecture, events are produced by Event Sources on the bottom layer. These events are then processed by Event Modellers, which use an Event Pattern Definition to route the events based on their type to the Event Processing Medium. The Event Processing Medium filters the events and processes them further, for example by translation or aggregation. It can detect and predict situations based on the given rules. This process allows it to build complex business events from simple atomic events, which can then be consumed by Event Consumers at the top layer of the reference architecture. Each detected event can be used to produce

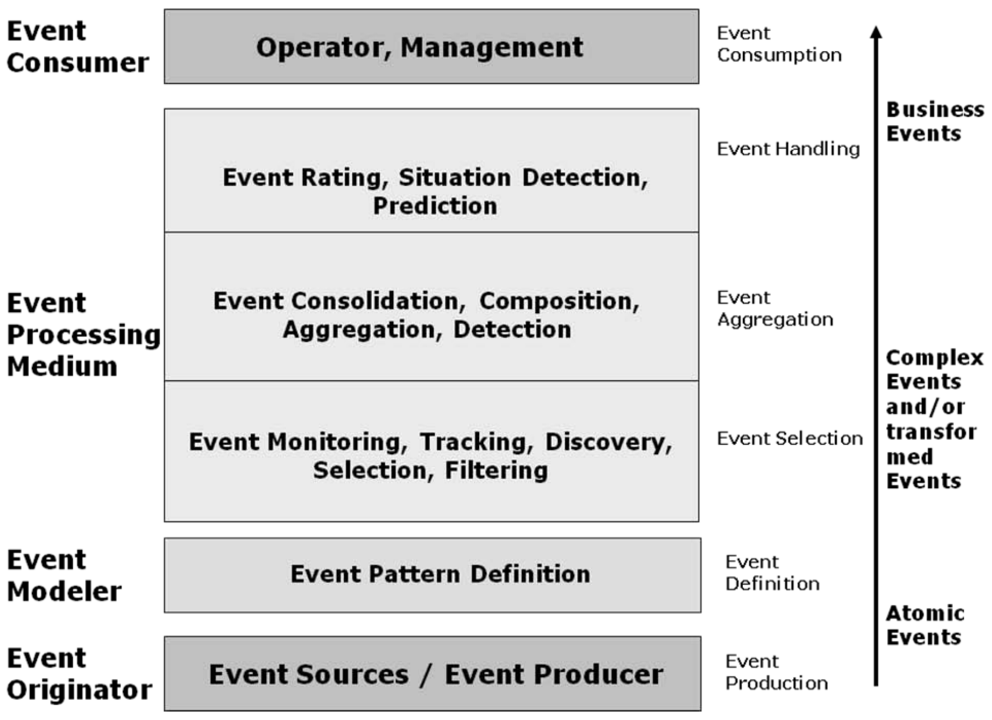


FIGURE 2.5: General EP Architecture Paschke and Vincent [2009]

new events that can be fed back into the systems. This creates an event processing circle and allows it to aggregate events into more complex events.

The terms for the components in this reference architecture are used throughout the document and are explained in the following sub-sections.

2.4.1.1 Event Producers

Event Producers, also often called called Event Originators, Event Emitters or Event Sources are system components, that send events to the system [Paschke and Vincent, 2009]. Events can be generated based on various sources, for example from message streams, processes, work flows, application events, or temporal data stores. Event Producers can also be wrappers for sensors, for instance a temperature sensor that publishes an event with the current room temperature every minute. These events can be simple, like a trigger from a switch, or complex, like the skeleton data from a 3D motion tracking sensor.

2.4.1.2 Event Modeller

The Event Modeller receives the events and matches them based on the Event Pattern Definition [Paschke and Vincent, 2009]. It is responsible for the forwarding of events from the Event Sources to the Event Processing Medium.

2.4.1.3 Event Processing Medium

An Event Processing Medium handles the selection, filtering, and classification of events and can aggregate events, which can produce more complex events [Paschke and Vincent, 2009]. The resulting events are then forwarded to the Event Consumers. Possible processing types are:

- Event Rating
- Situation Detection
- Prediction
- Event Consolidation
- Composition
- Aggregation
- Detection
- Event Monitoring
- Tracking
- Discovery
- Selection
- Filtering

2.4.1.4 Event Consumer

Event Consumer, also often called Event Sinks, subscribe to events, that match specific criteria [Paschke and Vincent, 2009]. Based on the received event, Event Consumers can trigger actions or act as Event Producers and create new events.

2.4.2 Integration of event-based and agent-based systems

The combination of event-based and agent-based approaches is not trivial because they follow two different paradigms. Agent-based systems consist of independent agents communicating via messages. The agents can act intelligently and independently. Event-based systems consist of event consumers and event producers interacting indirectly via an event bus using events or notifications. Since CEP engines are event-based and the smart systems studied here are agent-based, this is a special case of integrating an event-based system into an agent-based system.

According to Mariani and Omicini [2015] and Omicini et al. [2015] there are three steps to integrating multi-agent systems and event-based systems. These steps are:

- Recognising the **sources of events** - In the first step, all agents and the environment have to be valid event sources, and vice versa. This means that each agent can act as an event source, sending messages directly into the event processing. Furthermore, as event sources have to be valid agents, they inherit all agent features and become more expressive, because agents encapsulate control as well as management criteria.
- Defining the **boundary artefacts** - In the second step, boundary artefacts are defined to mediate between the event-based and agent-based worlds. A boundary artefact is responsible for the translation of events inside the agent-based system to the common event model. It can be used to transform multiple event definitions into one.

- Providing expressive **event-based coordination** models - The third step ensures that there are coordination components inside the system that can handle multiple event flows and mutual dependencies between agents and event-based components. This is important because, after an integration of event-based and agent-based systems, the complex dependencies between agents could emerge indirectly in the event-based part of the system.

This thesis does not consider EP architectures in general but only CEP architectures. Furthermore, this thesis focuses on multi-agent architectures with loose coupling, such as publish/subscribe communications. This decision is based on the results of the previous analysis, which found that both CEP and loosely coupled agent-based systems are typical for smart system architectures. The aim of this thesis is to support developers working with smart systems by integrating CEP into an agent-based architecture, not to show how event-based systems can be embedded into agent-based systems more generally. Nonetheless, the above steps are taken into account in the planned integration.

2.4.3 CEP in Smart Systems

CEP-systems are designed to be scalable and to handle a huge number of events with low latency, which matches well with the requirements of software in smart systems. This is because one of the challenges of smart systems, and especially context-aware systems, is the processing and storage of contextual information [Cook and Das, 2004]. A possibility to meet this challenge is to model the contextual information as events and process them through a CEP engine. The short-term storage of events and their aggregation can also be handled by the CEP engine.

There are many different implementations of CEP engines, thate have been used in distributed systems for a long time [Luckham and Brian, 1998]. Three such implementations are Esper, Siddhi, and Etalis. Esper [EsperTech, 2021] is a commercially supported open-source CEP engine that is implemented in Java and uses a SQL-standard compliant query language. It is used frequently in the literature to integrate CEP capabilities into smart systems and as a baseline for CEP implementations in the literature. The Siddhi event processing architecture [Suhothayan et al., 2011] combines stream processing aspects like multithreading and pipelining with the capabilities of a CEP engine, which has performance advantages over Esper. Etalis is a language for CEP, that is supposed to bridge the gap between event-driven and logic-based systems [Anicic et al., 2010]. It is a rule-based language with declarative semantics, temporal logic, and logic inference.

There are many possible applications of CEP in smart systems. For example, Augusto and Nugent [2004] argue that temporal reasoning based on contextual information can be used to recognise the activities of an occupant in a smart home environment, and Hallé et al. [2016] showcase that this can be done efficiently with CEP. It is also possible to use CEP to process sensor data and to detect errors in sensor networks, for example, to manage road traffic [Dunkel, 2009, Elchaama et al., 2017]. CEP is also often used in the field of IoT architectures for the pre-processing of context information [Chen et al., 2014, Ziehn, 2020].

Taylor and Leidinger [2011] argue that in heterogeneous sensor networks, there needs to be support for rapid specification of queries and experiments because it is challenging to derive measurable properties from sensor values or even to know what events of interest a sensor might produce over its lifetime. Therefore, a platform should include functions to find sensors, specify events, reuse measurement results, define actions when events are detected, and easily deploy these specifications at runtime. To implement all this, the context information and queries are modelled as an ontology. Queries are then automatically converted into a CEP query and processed by an existing central CEP engine, which processes all event streams in the system.

In a further development of this approach, Wang and Cao [2012] use a context-aware CEP architecture for IoT event processing agents. Ontology-based requests are converted into context-independent event stream tasks and processed by CEP. These agents form part of the event processing logic and handle the processing of events from the event producers to the event consumers. The agents take on different tasks, such as filtering, pattern detection, and transformation of events. CEP queries to the system are preprocessed and then executed with the help of one or more event processing agents. In this approach, there is no interaction with other agents in the IoT system. The results of the queries can be interpreted directly by the user or read by other components from an event cloud.

Cobeanu and Comnac [2011] use an Esper CEP engine for decision-making in agents for a traffic control application. The events generated by the agents are stored and read into an CEP engine, for example to detect traffic jams. The results of the evaluation in the CEP engine can then be used by the agents to adapt their behaviour accordingly.

Another way of integrating CEP into IoT systems was presented by Chen et al. [2014] and is based on a client server architecture. There can be several web-based clients in the system, each of which can start several servers that contain a CEP engine. Thus, this is a distributed CEP architecture with several engines. Each engine consists of multiple modules, including multiple event processing agents.

Akbar et al. [2015] presented a lightweight CEP architecture, called the Micro CEP Engine, for IoT applications that can run on hardware with limited resources. The core of the architecture is a central event processing engine that can process multiple event streams. Both normal CEP functions and event clustering based on machine learning methods are implemented.

It is possible to use several CEP engines in one system. Paraiso et al. [2012] present a distributed CEP engine that allows to process different event domains from different engines, building a CEP federation. This increases adaptability to different domains and scalability. The approach integrates the Etalis and Esper CEP engines, but the combination of several identical or different CEP engines increases the complexity of the system and creates a communication overhead between the engines. Different engines, different CEP dialects, or a new DSL must be used. The increased complexity makes it more difficult for developers to work with the system.

Zu et al. [2016] use a distributed data-driven CEP approach with Esper for a smart grid IoT application. The system uses a data distribution service. This is a middleware based on publish-subscribe messaging thats is connected to a distributed CEP engine to support the agents in the system in their decision-making.

Lachhab et al. [2016] integrate CEP into a context-aware service-oriented architecture to handle big data. It uses a central in-memory CEP engine to process the event streams. The CEP engine contains an event processing network consisting of several event-processing agents, that handle the event stream processing. The CEP agents are built with the Etalis engine to form a central CEP engine, and they interact with the system only via the output events. However, the data sources, such as CO2 sensors, are hardwired into the CEP engine. The CEP engine is therefore not flexible enough to be used for several use cases at the same time without adapting the architecture.

Meslin et al. [2018] use CEP to process sensor data in smart cities on several levels. The information is first pre-processed at the sensors before it is sent on. At the highest level, CEP enables queries that can be used for various tasks, such as to evaluate the delays of bus connections.

Parra-Ullauri et al. [2021] combine CEP with temporal graphs in an architecture for service monitoring. Temporal graphs allow for an efficient representation of the event history. Queries on these graphs can be used both for live monitoring and to investigate past system states, such as in the event of an error. The architecture combines these capabilities with the fast processing and short response times of CEP engines. The result is a central middleware that can work with temporal graphs and has an integrated CEP engine, Esper in this case, that processes event streams.

Table 2.3 offers a comparison of the publications analysed here. All these approaches have in common that the used CEP engine is not integrated seamlessly into the rest of the system. Either the CEP engine is used as a central component that processes all event streams and outputs the results, or several CEP engines

Author [Year]	Architecture / CEP Integration	Engine
Taylor and Leidinger [2011]	Central CEP server	Coral8
Wang and Cao [2012]	Central CEP server	Custom agent-based
Cobeanu and Comnac [2011]	Separated systems with API	Esper
Dunkel [2011]	Each agents have own CEP engine	Esper
Chen et al. [2014]	Distributed client-server based	Custom agent-based
Akbar et al. [2015]	Central CEP server	µCEP
Paraiso et al. [2012]	Distributed Heterogeneous CEP engines	Esper, Etalis, a.o.
Zu et al. [2016]	Data-centric publish-subscribe approach	Esper
Lachhab et al. [2016]	Single purpose in-memory CEP engine	Etalis
Meslin et al. [2018]	Large scale multi-layered distributed CEP	Esper
Parra-Ullauri et al. [2021]	Central CEP server with temporal graphs	Esper

Table 2.3: Comparison of other approaches to integrating CEP queries and event processing into a smart system.

are used, each of which processes specific event streams. Incorporating CEP into a smart systems architecture means mixing two different paradigms. On the one hand, smart systems consist of independently acting, often heterogeneous, components that communicate with each other. Therefore, an agent-based paradigm is often chosen for the architecture. At the same time, CEP engines promote descriptive definitions of the system's behaviour with rules or queries that are separate from the actual components of the system.

This thesis contributes to this research gap through the design of an architecture that allows a seamless integration of an CEP engine into an agent-based smart system in order to avoid the disadvantages identified and to keep the complexity of the system as low as possible.

2.5 Program Comprehension in Complex Distributed Systems

Program Comprehension is the task of understanding how a software system or part of it works [Maalej et al., 2014]. This includes the task of understanding the source code of a programme, which is one of the most time-consuming tasks during software development [Siegmund, 2016]. An understanding of the system is important to perform maintenance tasks correctly [Cornelissen et al., 2009].

To augment developers' comprehension, tools can automate manual tasks such as the generation of abstract system representations and knowledge verification [Lin et al., 2020]. They can identify components and their dependencies and help generate an abstract representation of the system [Siegmund, 2016].

In traditional software development, there is a strict separation between development-time and runtime and software engineering research is mostly focused on development-time [Baresi and Ghezzi, 2010] to make the development of software less error-prone and faster. The traditional assumption is that software is only changed during the development phase and is then unchangeable during runtime. If an error occurs during the runtime of the software, it must be fixed offline by a programmer. This strict separation is becoming increasingly softened [Baresi and Ghezzi, 2010]. Software develops quickly, and changes should often reach the user as quickly as possible. Adaptive software systems are needed that can adjust to changed environments at runtime [Baresi and Ghezzi, 2010].

Smart systems are an example of such adaptive systems. They adapt their function to changes in context and are partly subject to requirements for quick adaptability to the needs of the user [Bettini et al., 2010]. In addition, they often consist of subsystems and form a complex SoS [Fortino et al., 2021] that cannot be processed in an offline development environment, as is traditionally the case [Fang, 2021].

To be able to use development tools not only at development time but also during the runtime of a programme, they must be adapted or replaced by other tools. One challenge in processing software at runtime concerns discovery and learning [Baresi and Ghezzi, 2010]. In dynamic software systems, services can appear or disappear at any time, influencing the behaviour of the overall system. Tools are needed to make the current state of software visible to developers and other components at runtime. Another challenge is the verification or monitoring of the software at runtime [Baresi and Ghezzi, 2010]. Traditional methods such as unit tests, which are supposed to detect errors after a change during development, do not work here. Tools are needed that can find bugs due to changes at runtime or help developers track changes made to the code.

One possibility to implement this is rule-based verification of the runtime behaviour of software [Havelund, 2015]. The behaviour of the system is monitored at runtime and then analysed during or after the execution of the programme. The programmer formulates rules that must be followed during the execution of the programme [Havelund, 2015]. A simple example would be to define that an event A always has to occur before event B to ensure that this is the case at all times during the runtime of the programme.

If an error is reported during runtime by runtime verification or other sources, such as a user of the system, it is the task of a developer to find the cause of the error through debugging techniques in order to be able to fix it. Debugging complex distributed systems is an important but sometimes very difficult task, because tracing concurrent events and understanding the communication relationships between components can be challenging [Beschastnikh et al., 2016]. Beschastnikh et al. [2016] list seven approaches that can help developers debug distributed systems:

- **Testing** - Manually written tests can help check whether individual components or parts of the system show the expected behaviour. But there is no guarantee that all errors will be detected by tests.
- **Model checking** - Model checking is the systematic checking of a system description against a specification. The system is modelled with mathematical methods and then examined for expected properties. This procedure can be very time-consuming for larger systems and can only be used to a limited extent if the system is dependent on environmental influences [Beschastnikh et al., 2016]. For example, agent-based systems can be analysed with petri nets to find deviations or errors in the protocol [Poutakidis et al., 2002].
- **Theorem proving** - Theorem proving can be used to verify that a system behaves according to expectations. To do this, a developer must specify (e.g. by using special programming languages) why a programme must behave as expected, so that a theorem proving tool can check the specifications.

This can be very time-consuming and may not really be suitable for legacy systems, as they are often found in SoS.

- **Record and replay** - An execution of the system can be recorded in order to reproduce or simulate it later. In this way, processes can be reproduced afterwards, and errors can be localised.
- **Tracing** - Tracing refers to the follow-up of events or messages in the system between components. Messages are annotated with tracing information, which requires the compatibility of all message-processing components.
- **Log analysis** - If the system under investigation outputs log messages, these can then be analysed to find errors or to understand the functioning of the system. Large systems that log in detail can produce large amounts of data that can only be analysed by developers with great effort. Here, tools can help to automatically find anomalies in the log data and certain processes in the system.
- **Visualisation** - Distributed systems can become very complex due to the interconnectivity and dependencies between components. Visualisation tools can help developers understand relationships and processes by creating a graphical model of the software.

Model checking and theorem proving are not considered further in this thesis. Both are powerful tools for understanding relationships in systems and proving their functionality. However, this requires the formalisation and prior knowledge of the developers in this area. It is unclear how fast experimentation in a dynamic smart system would be feasible with such approaches. Furthermore, it has been shown that these formal methods are not suitable for all purposes, and that there are limits in multi-agent systems that require the use of other methods [Edmonds and Bryson, 2004].

An alternative approach to formal methods is reverse engineering. Reverse engineering is the practice of understanding hardware designs from one's own or others' finished products [Chikofsky and Cross, 1990]. Similarly, an existing software project can be analysed to produce design documents and thus create a higher level of abstraction to better understand the software [Chikofsky and Cross, 1990]. Despite the availability of support tools, this process is a predominantly manual task that is very time-consuming [Bosse et al., 2008].

To generate a higher abstraction of the agent system architecture, concepts such as goals and beliefs can be used [Lam and Barber, 2005]. Automated explanation generation can interactively generate relational graph interpretations of goals, beliefs, intentions, actions, events, and messages and compare them to the developers' current knowledge [Lam and Barber, 2005]. This can not only document the behaviour of an agent, it can also lead to an automated explanation of its behaviour. To gather runtime information with this method, the comprehension tool provides the developer with logging instructions that then must be added to the part of the source code that defines specific concepts of an agent. However, this requires a correspondingly uniform representation of the inner relationships of intelligent agents that is uniform in the system. In heterogeneous systems, however, this is usually not the case or can only be achieved with considerable effort.

Most reverse-engineering tools include visualisations to directly display generated models of the system [Stroulia and Systä, 2002]. Visual representations have been found to be very effective in communicating this complex information [Stroulia and Systä, 2002]. Visualisations of dynamic systems are often based on variations of direct graphs [Stroulia and Systä, 2002].

One example is ViVA [Lee et al., 2014], a visualisation and analysis tool for distributed event-based systems. It combines three techniques into one tool: runtime visualisation, message-log replay, and combined static and dynamic analysis. The runtime visualisation should enable developers to better assess the state of the running system batter than relying on the source code or diagrams from the documentation. The visualisation depicts all components, connectors, dependencies, and the event stream between the components. The tool can record the message stream and then play it back interactively under the control of a developer, which should support the identification of errors and their causes. Finally, Viva includes a combined static and dynamic analysis of control-flow paths, allowing the system to be analysed and traced at any time.

Another example of a visualisation tool is Shiviz [Beschastnikh et al., 2016]. Shiviz was developed to help analyse event chains to improve the debugging of distributed systems. The tool supports keyword and structured searches to enable the identification of events with specific attributes as well as events that are related to other events. With the latter, common patterns such as request-response or broadcast messages can be found and analysed. In addition, the tool supports the comparison of several executions and the clustering of similar executions of a part of a system, which helps to identify anomalies in the programme flow. Static analysis, in conjunction with formal verification methods, can be used to verify and validate processes. This can help find events that trigger other events and possible scheduling problems [Rabinovich et al., 2010].

There are similar approaches for agent-based systems. Flater [2001] use a case study to demonstrate that in cases of coordination problems between agents, it can be helpful to have a visualisation of the interaction between agents to localise problems and to then investigate them in a second step with formal methods. There are different views that can be used for visualisation to debug agent-based systems [van Liedekerke and Avouris, 1995]. Often, a distribution view that shows which components are running on which machines or an agent-based view is used, but an interaction-based view that shows the different relationships between agents in a graph can also be helpful. These relationships can be dependencies, communication, or other relationships such as cooperation and negotiated roles.

Another way to debug agents is analogous to conventional step-based debuggers for programmes. However, the problem with agents compared to conventional software components is that agents can often be mobile. This means they can change the computer they are running on at runtime and therefore debugging tools need to work remotely if they want to track these migrations [Osaki et al., 2015]. In addition, the execution environment should be considered when debugging, as agents can behave differently due to differences in the execution environment [Higashino et al., 2013].

In summary, visualisation and other debugging tools are an important part of facilitating programmers' work in distributed agent-based systems. It is important that both the agents and the communication between agents can be visualised and analysed. Since many systems in smart systems change dynamically and behave in accordance with on the context, it is necessary to carry out analyses at runtime to support understanding of the programme and troubleshooting. For this reason, visualisation, tracing, log analysis, and recording of messages are features provided in the architecture presented in this thesis to support program comprehension and debugging tasks.

2.6 Conclusion

During the preceding literature review, all research areas considered relevant for this thesis were analysed one by one. Literature on middleware for smart systems was discussed starting with a generic middleware architecture, and the literature was then evaluated for two architectural elements central to this work: agent-based design and publish/subscribe communication. This has shown that architectures and middleware for smart systems are active research topics, which also means that there is a need for research laboratories to investigate these topics to find new solutions for smart system software.

Several research gaps were identified through the analysis of the literature. Firstly, coupled with the lack of analysis of middleware requirements for smart systems laboratory environments, and no published architectures could be found that primarily addressed the challenges in these environments. Supporting developers in program comprehension and debugging tasks is at most one of many requirements in the comparable publications analysed, but this is rarely reflected in the evaluation of the architecture (see Section 2.3.4).

Secondly, comparable architectures for smart systems using publish/subscribe communication were analysed with performance measurements, simulations, and case studies (see Section 2.3.4). The first two methods are difficult to reproduce or compare with other architectures without the concrete implementation of the system or simulation tools. Moreover, these methods are more suitable for evaluating performance characteristics, such as message latency, and less for evaluating functionality or usability. The case studies carried out are more suitable for this, but in the publications analysed, they were not carried out according to a reproducible method, and several stakeholders were not involved in the research. This means that there is a lack of reproducible, structured evaluations of architectures for smart systems.

Finally, the review of literature dealing with the integration of CEP in smart systems failed to identify a publication that allows seamless integration into an agent-based system (see Section 2.4.3).

In the following, first the requirements for an smart system architecture are analysed (see Chapter 3). Then an architecture (see Chapter 4) and its evaluation (see Chapters 6, 7 and 8) are presented as a contribution to filling the identified research gaps.

Chapter 3

Requirement Analysis

This chapter analyses the requirements of a software architecture for smart systems. In particular, the requirements of laboratory environments for smart systems is addressed, including both requirements from the literature and requirements from experiences in the two research laboratories presented below.

3.1 Interdisciplinary Research Laboratories

The analyses and evaluations in this thesis were carried out over several years and mainly took place in two research laboratories, the Living Place Hamburg and the Creative Space for Technical Innovations (CSTI), which are presented in more detail in the following sub-sections.

3.1.1 Living Place Hamburg

The Living Place Hamburg is a smart home laboratory at the Hamburg University of Applied Sciences (HAW) [Livingplace]. The main part of the laboratory consists of a 140 m^2 loft-style apartment, including a kitchen area, sleeping, dining, and living areas, and a separate bathroom (see Figure 3.1). It is a fully functional apartment that can be used for experiments under real-life conditions. In

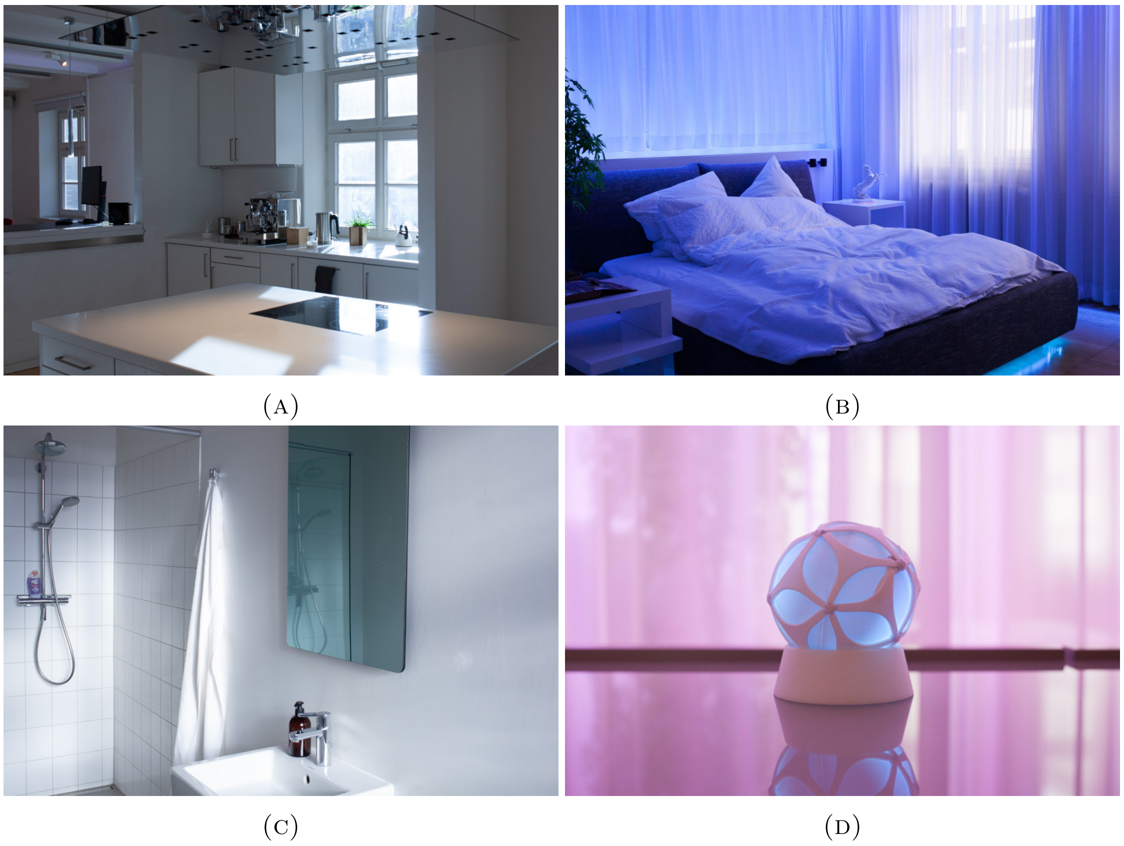


Figure 3.1: Living Place Hamburg. The kitchen (A), the bedroom (B), the bathroom (C) and a tangible object (D). [Broscheit, 2022]

addition, three office rooms and a control room are part of the Living Place. The control room contains the network technology and workstations for monitoring and evaluating experiments with the help of video and audio monitoring in the laboratory.

The Living Place was established in 2009 with the aim of conducting research in the fields of ubiquitous computing and smart homes. Initial projects included the development of position recognition and, based on this, the prediction of the movement patterns of residents [Jens Ellenberg et al., 2011]. In addition, sensors and actuators for the smart home were researched. These include the recognition of sleep phases with sensors in the bed, the recognition of sitting positions on the couch, a RGB light control, and window and blind actuators.

The software architecture in the Living Place is based on a publish/subscribe blackboard approach using ActiveMQ as a message broker [Jens Ellenberg et al., 2011].

Sensor data is published on the blackboard in communication groups and enriched by interpretation agents in several steps. Decisions are then made and actuators controlled based on the enriched contextual information. The software infrastructure of the Living Place has grown steadily over the years through further research and student theses, including projects such as an intelligent bathroom mirror, gesture and voice control of home automation, and a service for the automatic display of situation-dependent information on suitable displays near the user.

In 2020, the goals of the laboratory were redefined and expanded to give greater attention to people, with investigations into the effects that digital changes can have on residents. With these adapted goals, the software architecture was also changed and is now based on MQTT and other software. The experiments and analyses in this thesis are based on the Living Place before the changes in 2020.

3.1.2 Creative Space for Technical Innovations

The Creative Space for Technical Innovations (CSTI) is an interdisciplinary research laboratory that was established in 2016 at the Hamburg University of Applied Sciences (HAW) [CSTI]. The CSTI is a platform for interdisciplinary projects, collaborations, research studies, and student projects. It offers the technical and methodological prerequisites to support these projects.

The laboratory is designed for rapid prototyping of projects. In research environments, it is essential to enable researchers to quickly conduct experiments based on their research questions without limiting their creativity. Resnick [2007] proposed the kindergarten approach for learning, which is based on short imagine-create-play-share-reflect cycles and was developed in the MIT Lifelong Kindergarten Group, to improve creative thinking. The CSTI follows this idea and provides a space for creative work on projects.

The CSTI is composed of projects with four different main topics:

- **Machine Learning and Data Mining** - Machine learning can be used in a wide variety of areas, such as the recognition of objects, people, and gestures for interaction. Other use cases include smart recommendations for users, speech recognition, and more.

- **Interactive Virtual and Augmented Reality** - The CSTI researches interactive virtual and augmented reality topics in areas such as virtual teaching, visualisation of complex data, and intelligent environments.

- **Ubiquitous and Tangible Interaction** - Research in the field of ubiquitous computing and tangible interaction at the CSTI mainly includes environmental sensing and quantified self projects. Smart objects, tangibles, and ubiquitous computing devices are developed and analysed to study Human-Computer Interaction (HCI).

- **Science and Technology Studies** - In the field of science and technology studies, the CSTI's projects use interdisciplinary methodologies such as critical design, situated action, ethnography, and creative computing to investigate the relationship between digital technology and human adaptation.

All the main topics of the CSTI combined to create smart environments. Machine learning can be used to intelligently support people in their environment. Virtual and augmented reality, and ubiquitous and tangible interaction both deal with the interaction between humans and machines. Science and technology studies reflect these projects from an interdisciplinary perspective. Both the Living Place and the CSTI are part of the Research and Transfer Centre Smart Systems at the HAW. There are thematic overlaps between the labs, and the teams work closely together. The topic of ubiquitous and tangible interaction, in particular, is a common one between the two labs and is the reason the CSTI is also used as a prototyping lab for projects in the Living Place.

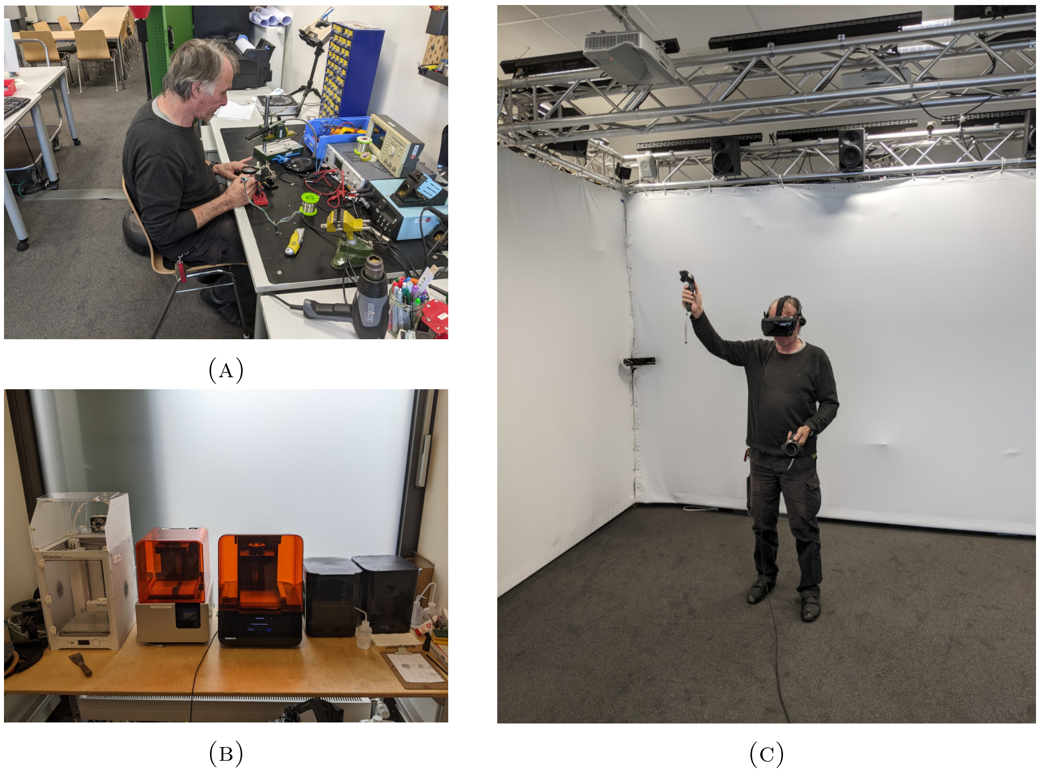


FIGURE 3.2: The CSTI with areas for microelectronics (A) and 3D printing (B) projects and a truss system with tracking systems for virtual and augmented reality experiments (C)

3.2 Requirements for Smart System Architectures

A number of requirements for software architectures for smart systems in general and middleware in particular emerge from the literature and previous experience in the presented laboratories. These requirements are presented and analysed in the following paragraphs, and they form the basis for both the design of the software architecture presented here and the subsequent evaluation.

There are several literature surveys that list requirements for and features of different smart systems. These refer to specific environments, such as context-aware systems [Henricksen et al., 2005, Bratskas et al., 2009, Fortino et al., 2014], IoT [Razzaque et al., 2016, Plattner et al., 2020], CPS [Shi et al., 2011], and SoS [Fortino et al., 2021]. The requirements largely coincide but in part have a different perspective on the requirements. Based on these surveys, the following requirements arise for the architecture presented here.

- **Support for heterogeneity** - Support for heterogeneity is one of the defining characteristics of architectures for smart systems. Heterogeneity is a challenge for smart environments [Fortino et al., 2014] as well as for the IoT [Savaglio et al., 2017, Plattner et al., 2020], CPS [Shi et al., 2011] and SoS [Fortino et al., 2021]. In these systems, it is common to have components developed in different programming languages and supporting different protocols [Henricksen et al., 2005]. There are also large differences in the capabilities of the individual components, which can range from small micro controllers, for example for sensors and actuators, to their own powerful complex systems, for example for machine learning tasks. In addition, potentially complex legacy systems have to be supported in some cases. Furthermore, it is often necessary to connect physical processes and hardware components to the system [Shi et al., 2011], which is especially the case in CPS.

- **Scalability and latency** - The middleware must be able to scale from small systems to large numbers of components [Henricksen et al., 2005, Razzaque et al., 2016, Plattner et al., 2020] because smart systems can become very large and also grow over time. At the same time, the message latencies must remain low to be able to implement time-critical functions such as user interaction.

- **Support for mobility** - A smart system must be able to handle and support mobile components. In smart environments, sensors and applications can be mobile [Henricksen et al., 2005, Bratskas et al., 2009] and may be located, for example, on the user's smart phone. In CPS, the dynamic reorganisation of components is one of the defining criteria. Therefore, the middleware must support appropriate protocols and implement a flexible discovery mechanism for components [Henricksen et al., 2005].

- **Tolerance for component failures** - The system should have a high fault tolerance to reliably perform its tasks [Henricksen et al., 2005, Bratskas et al., 2009, Shi et al., 2011, Plattner et al., 2020]. When working with sensors and other context-dependent components, it is to be expected that measurement errors and failures will occur, which the system must handle appropriately. In addition, components in a smart system are distributed across the network, so errors such as disconnections and temporary inaccessibility of components must be handled in such a way that the rest of the system can continue to run.

Henricksen et al. [2005] list two further requirements: *Traceability and Control* and *Ease of deployment and configuration*. These are further subdivided and specified in the following. This is necessary because the support of developers in smart systems is the focus of this thesis, and additional requirements have to be added. In addition, further requirements are added, that result on the one hand from the experiences in the research laboratories in which the architecture in this thesis was developed and on the other hand from the results of the studies carried out in this thesis (see Chapter 6). This results in the five additional requirements as follows:

- **Traceability and control**
  - **Support for debugging** - Even if the architecture includes a high degree of fault tolerance, any errors that occur must still be analysed. Unexpected error states can affect the availability of the system and even intercepted error states can have a negative impact and affect performance. To localise errors, it is helpful if the components and the messages between them are visible and can be inspected to support the debugging process [Henricksen et al., 2005]. This is especially important for systems that can only be analysed at runtime, which may be necessary, for example, if the system is highly context-dependent. Here, it must be ensured that system components can be inspected without affecting the system. It is also useful for debugging to have control over the system, such as to use or isolate specific agents and groups. This can be used to create specific scenarios, reproduce errors, and perform tests.

  - **Program Comprehension support** - Similar to the debugging approach, it is necessary to help developers understand the system. This is particularly necessary for new developers to become familiar with the system. At the same time it is also important for more experienced developers, because it is a challenge to keep track of complex dynamic systems that change depending on contextual information. It is important that the system make information openly available to developers to allow easy inspection in test settings and at runtime.

- **Development support**
  - **Support for fast experiments** - The possibility to conduct quick experiments is helpful for many steps of the development process. Manual and automatic tests can be helpful or even necessary to identify and reproduce errors to sufficiently exclude the possibility that they occur again. Experiments can also be used to test assumptions about the system and improve understanding of its behaviour at runtime. Furthermore, these experiments can also be used for research purposes in research laboratories or for decision-making in companies. There are often time constraints and a limited budget for answering a question, which makes it helpful to conduct experiments quickly. Here, frameworks for the development of new components, and tools for development support are helpful to accelerate the development process [Henricksen et al., 2005].
  - **Deployment and configuration** - Smart systems usually consist of a large number of components that communicate via the network and run distributed on different computers. Therefore, it is important that the architecture supports the deployment of components and facilitates their configuration. A simple configuration promotes the adaptability of the system to the needs of the users and the reusability of components, which can thus reduce the complexity of the system. A high degree of configurability of the components can also be useful to provide configuration options for end users [Henricksen et al., 2005], which can

be particularly useful for smart homes, where the tenants or owners are supposed to control the system.

- **Flexibility** - Architectures for smart systems must have a high degree of flexibility. This becomes clear when one considers the diverse use cases of these architectures. A few examples are smart homes, smart cities, smart power grids, medical devices, wearables and many more. With the increasing interconnectedness of these systems, it becomes more important that they be developed with compatible architectures. In addition, it is important in the research context of laboratory environments that these can be used universally to be able to react flexibly to upcoming projects and research projects. These may not yet be determined when the system is set up and will only be added over time, depending on project applications.

- **Support of communication between developers** - In a conversation between developers about a complex system, it is helpful or even necessary to have a common framework to think about processes in the system [Plate, 2010, Arnold and Wade, 2015]. Therefore, it is a further requirement for the architecture to support this. On the one hand, it is important that the architecture is consistent in its concepts. The use of several different paradigms for similar requirements can increase complexity and lead to confusion. On the other hand, uniform visualisations of the system can help develop common ideas about processes.

Another important requirement in the literature is *Privacy and Security*. This requirement is not considered for the architecture developed here because it is an architecture for development and laboratory environments and not for production systems. Therefore, this requirement is not included as part of the evaluation in the following work.

- **Privacy and Security** - Smart systems work with safety-critical systems and context information about users that needs to be protected. In smart environments, it is common to collect sensitive information about the user, analyse it, and then support the user depending on the situation [Cook, 2009]. This collected information must be protected from outside access according to the user's requirements [Henricksen et al., 2005]. Furthermore, actors and other sensitive components must be protected from manipulation by third parties. This can be particularly important depending on the type of system and its purpose. For example, systems like CPS have high security requirements as they are also used in the health sector [Shi et al., 2011, Plattner et al., 2020].

#	Requirement	Main Source
1	Support for heterogeneity	Literature
2	Scalability and latency	Literature
3	Support for mobility	Literature
4	Tolerance for component failures	Literature
5	Support for debugging	Expert Interviews
6	Program Comprehension support	Expert Interviews
7	Support for fast experiments	Expert Interviews
8	Deployment and configuration	Literature
9	Flexibility	Literature
10	Support of communication between developers	Expert Interviews

Table 3.1: List of all requirements for the architecture to be designed. The requirements are based on the literature and have been extended based on the results of the expert interviews.

3.3 Conclusion

In summary, there is no publication of requirements especially for smart systems research lab environments in the literature. This is a research gap to which this thesis contributes by supplementing the requirements from the literature of different smart systems with specific requirements based on the experiences from research lab environments. The list of all requirements and whether they are based on the literatures or are a result of the expert interviews is displayed in Table 3.1.

The listed requirements are used in the following chapter for the design of a software architecture for smart systems. The architecture is then evaluated with regard to these requirements. In the process, the requirements themselves are also examined. On the one hand, other environments are examined during the expert interviews to determine the transferability of the requirements determined here. On the other hand, scenarios are collected and evaluated through a survey of the stakeholders to concretise the requirements and determine their importance.

Chapter 4

System Design and Implementation

This chapter presents the system architecture that has been designed to meet the identified requirements. The aim of the architecture presented here is to support the development of software in smart systems in general and in research laboratories in particular. Therefore, the main goals of the architecture are the improvement of program comprehension and debugging tasks.

This chapter first provides an overview of all layers of this architecture, followed by a detailed description of each layer. Each layer is presented individually and is designed according to the requirements relevant to that layer. The evaluation of the entire architecture with expert interviews, latency and scalability measurements, as well as a scenario-based architecture evaluation is carried out in Chapters 6, 7 and 8.

4.1 Architecture Layer Outline

Based on the literature and the requirements identified (see Section 3.2), a middleware architecture is presented below, that was designed in particular to support developers during development, program comprehension and debugging. Since the focus here is on the interaction with the developers, the architecture is first examined from the perspective of this interaction. The architecture consists of three main layers, which are described and further subdivided in the following sections layer by layer. The bottom layer is the messaging layer, or the publish/subscribe layer. It provides the interfaces for communication with other agents and implements the basic functions of the system, such as monitoring the availability of agents. Based on this, the CEP layer is built. It contains a complete CEP engine based on the publish/subscribe middleware layer. Finally, the User Interaction Layer provides the interface for developers to interact with the system via visualisations of the system structure and CEP queries. Figure 4.1 presents an overview of the three main layers of the architecture.

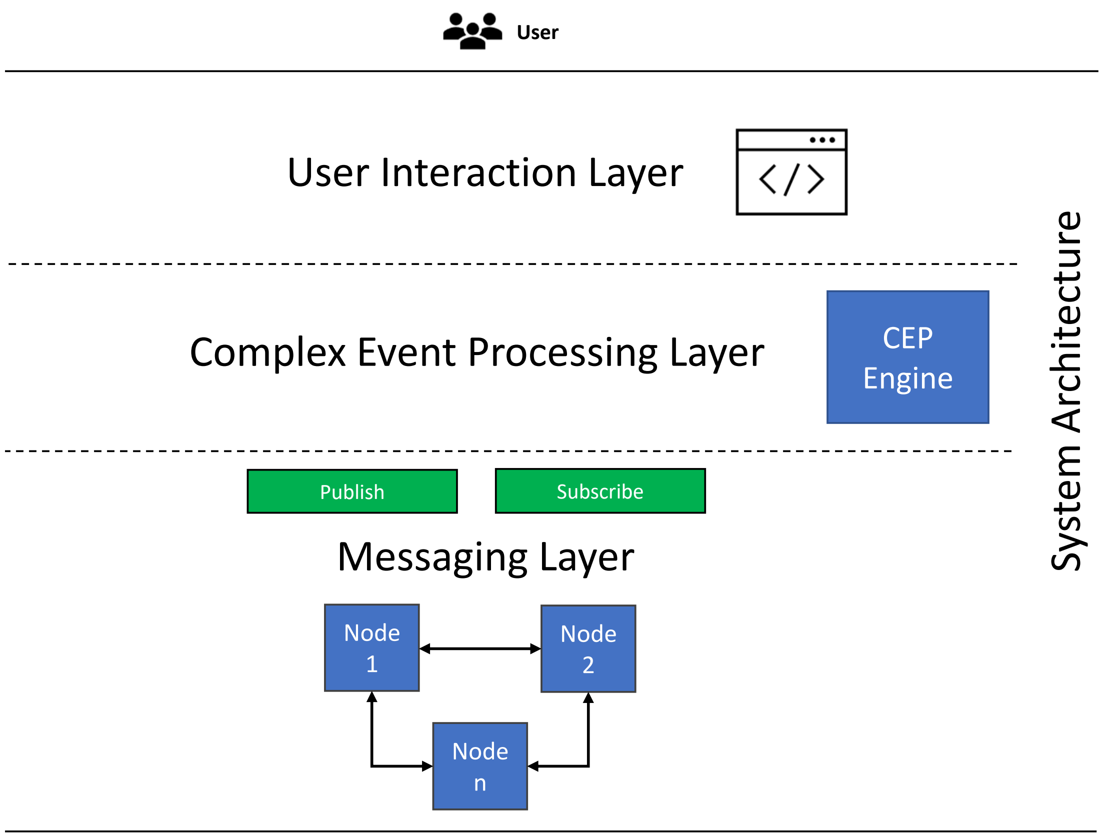


Figure 4.1: The three main layers of the architecture

Figure 4.2 offers an overview of the whole architecture, which is described in detail in the following chapters. In addition to the three layers already mentioned, the application layer is also shown here, in which all agents and user applications are located. This illustration also highlights the management and monitoring functions of the middleware for the individual layers. These functions are part of the middleware and are also implemented in agents. These include the monitoring and management of the nodes, agents, and CEP queries.

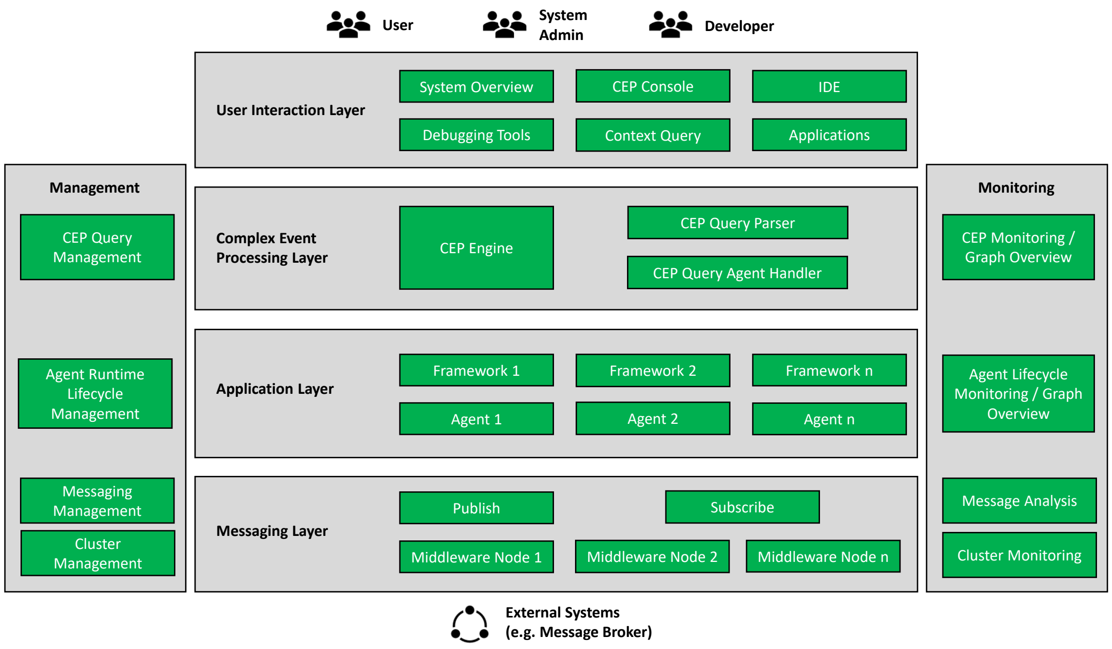


FIGURE 4.2: System architecture overview

The presented architecture can also be seen as middleware for context processing, based on the Generic Middleware Architecture by Henricksen et al. [2005], which is more in line with the representation of a middleware for smart environments. This describes the processing of context information and the control of actors starting from the user application and is shown in Figure 4.3

The right side includes the layers from the generic model of Henricksen et al. [2005] and the left side includes the architecture presented here. The bottom layer is identical in both architectures and it contains sensors and actuators. The only difference is that in the architecture presented here, programming toolkits can also be used for the integration of sensors and actuators. The two layers above are implemented with context processing agents and databases, as in Henricksen et al. [2005], as well as with a CEP engine. Developers can choose whether they write their own agents or implement the functionality via the CEP engine in the system. The CEP engine takes over both the processing of the context information and the storage of messages for defined periods of time. If data has to be persistent over

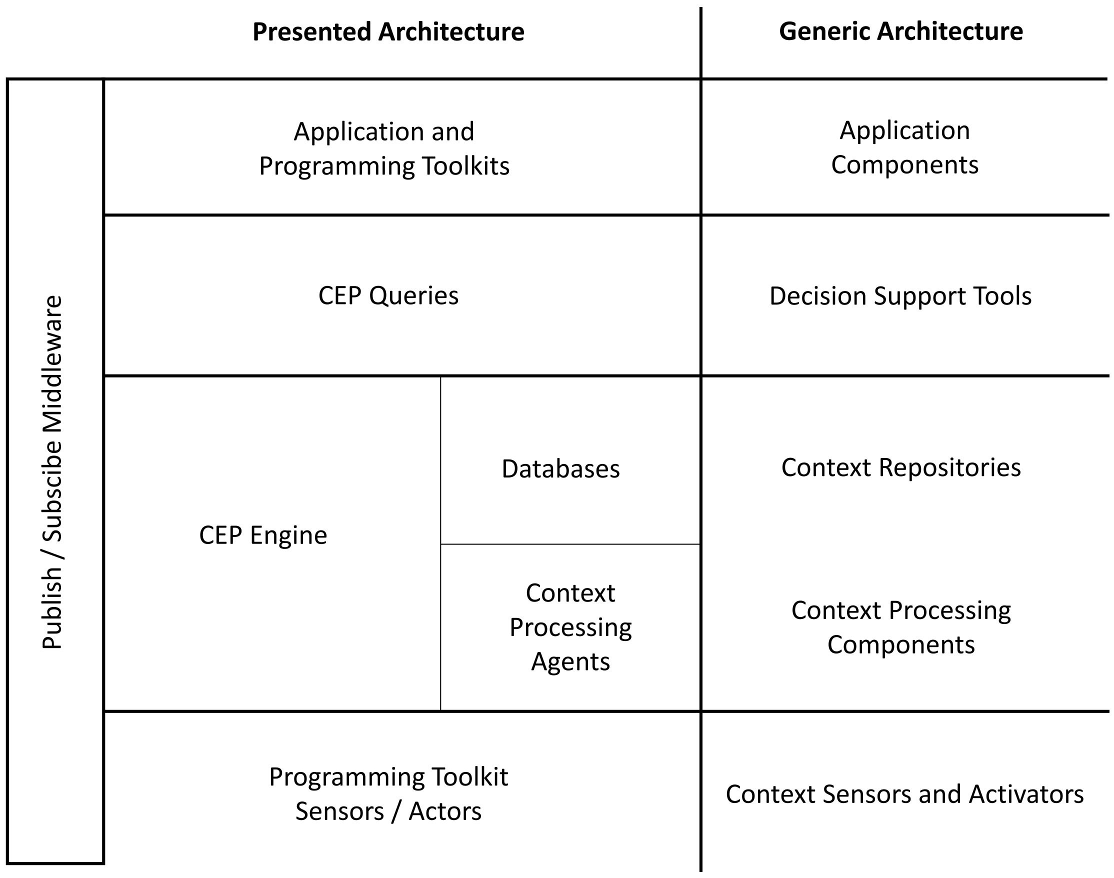

Presented Architecture
Generic Architecture
Publish / Subscibe Middleware
Application and Programming Toolkits
Application Components
CEP Queries
Decision Support Tools
CEP Engine
Databases
Context Repositories
Context Processing Agents
Context Processing Components
Programming Toolkit Sensors / Actors
Context Sensors and Activators

FIGURE 4.3: Architecture comparison to generic architecture

a longer period of time, it should be stored in a database. The decision support tools that help to make context-dependent decisions are defined in the architecture presented here with CEP queries and processed by the CEP engine. In addition, it is possible for agents to access the events directly or read data from the database. The top layer, in which application agents are located, is also identical in both models. The implementation of the agents is supported by programming toolkits.

4.2 Messaging Layer

The messaging layer forms the foundation for the architecture presented here. Its most important component is the middleware, which provides the interfaces for communication between agents and takes care of monitoring the agents and system components. A first version of the architecture of this layer was published in Eichler et al. [2017]. The main characteristics of the messaging layer are presented below. These include, above all, the middleware nodes, the message format, the programming toolkits, and the runtime environments. At the end of the section, the results of a latency and scalability test are presented.

As pointed out during the literature review in Section 2.3.4, there are already publish/subscribe systems used in smart system architectures. Commonly used examples include MQTT, but ZeroMQ and RabbitMQ are also used, as evidenced by the expert interviews in Chapter 6. To fulfil the requirements determined during the analysis, a custom publish/subscribe implementation is used here. To meet the identified requirements for program comprehension and debugging support, the middleware should be able to tell the developer which agents are running at any time and what communication links they have. To do this, it is necessary to monitor all agents and their subscriptions, which is not easy with the open-source solutions listed above. One reason for this is that many publish/subscribe implementations do not monitor subscriptions for performance reasons. Furthermore, to support heterogeneity, it should be possible to subscribe and publish messages without programme libraries using basic network protocols such as UDP or WebSockets. Supporting mobility across runtime environments also requires changes to the messaging systems to support changes to location at runtime with ongoing subscriptions without message loss. None of the existing implementations could implement all these requirements without requiring major adaptations or major efforts for agent developers. For this reason, the message layer presented below is based on a custom implementation.

4.2.1 Layer Architecture

The messaging layer can be subdivided into three sub-layers, as shown in Figure 4.4. The layer at the bottom, called the middleware node layer, is the foundation for the system's messaging. It is a distributed cluster consisting of multiple middleware nodes. The abstraction layer is the Publish/Subscribe Layer, which implements the publish/subscribe based group communication for all agents in the system. It handles the subscriptions from all agents and distributes messages

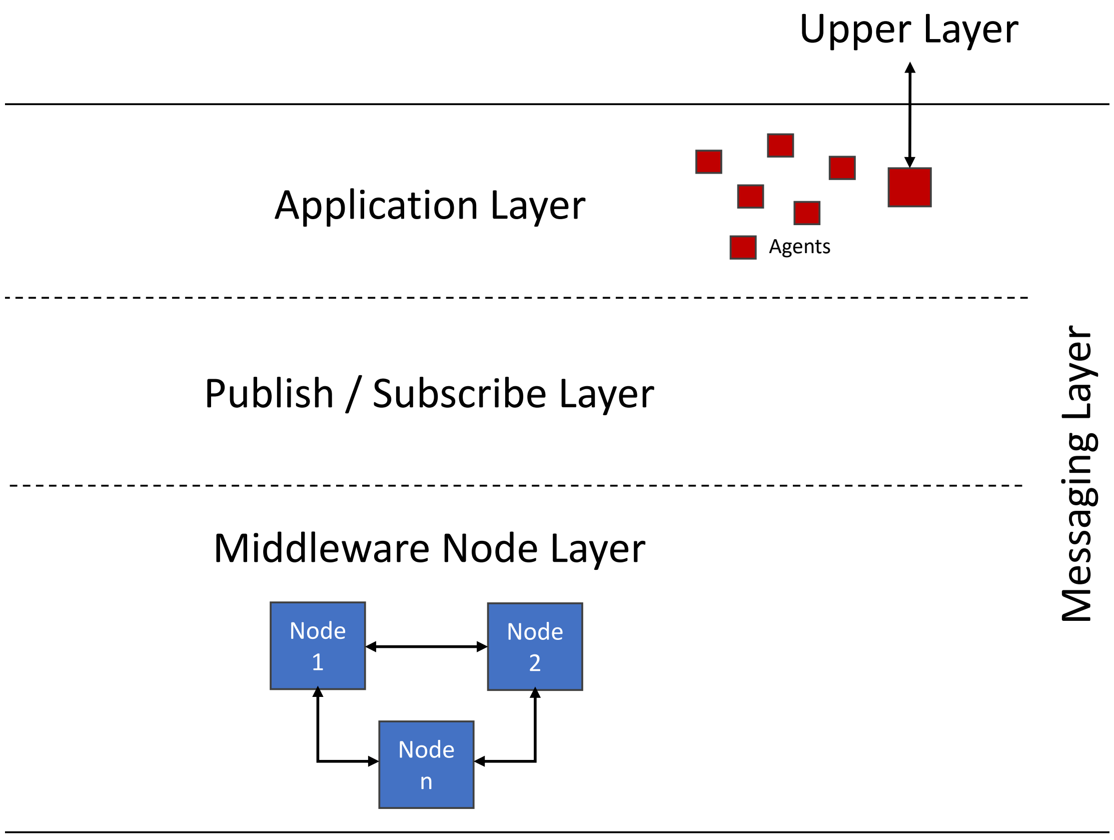


FIGURE 4.4: Structure of the messaging layer

accordingly. The application layer at the top contains all user applications. This includes all agents connected to the middleware that implement, for example, context handling and user interaction. The layers that are built on the messaging layer, such as the CEP layer, are implemented with the help of agents in the application layer.

Another representation of the messaging layer thats includes the communication paths of the components, is given in Figure 4.5. The cluster of middleware nodes maintains itself by sending regular heartbeat messages, to detect the reachability of each node. The heartbeat messages are sent over TCP between all middleware nodes. Cluster management tasks, like handling a non reachable node, are completed by a selected leader. Since the leader is negotiated by a majority vote between the nodes, the cluster is capable of acting as long as more than half of all middleware nodes are reachable. This ensures that at any given time, there is only one leader and therefore only one cluster. It is not possible that one or more

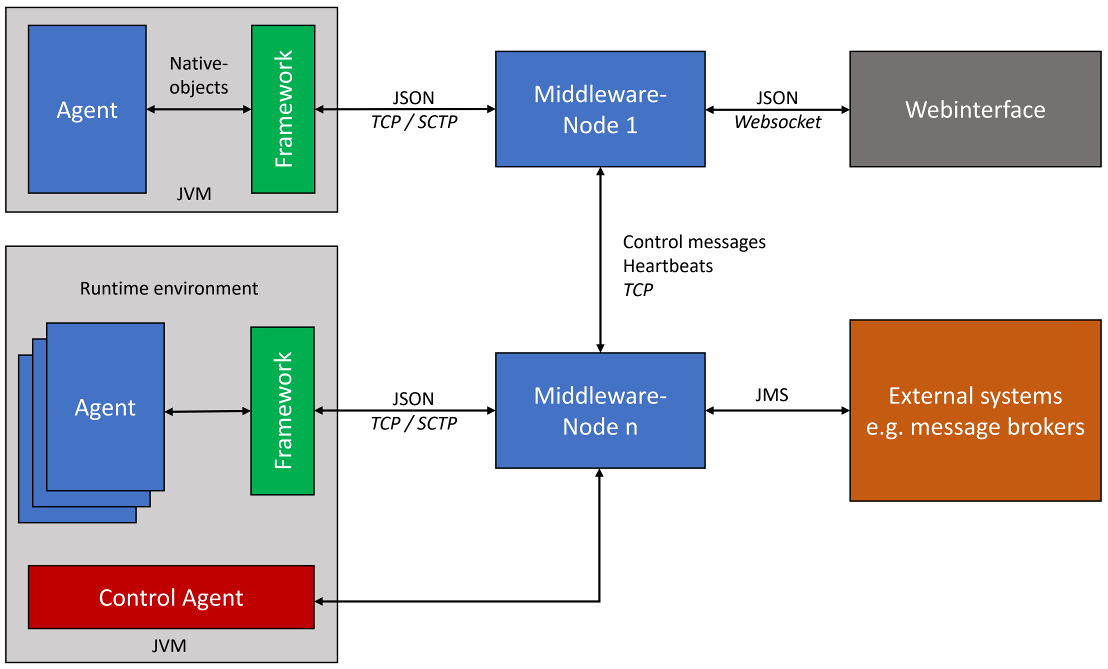


FIGURE 4.5: Messaging layer architecture

nodes that have temporarily lost their connection to the cluster decide to form a new cluster, because there can only be one group of nodes with more than half of the total number of nodes.

Each node can act as a connection point to agents by providing an Application Programming Interface (API) over TCP, SCTP, and WebSocket connections. This selection of protocols should make it possible for many different components and programming languages to communicate with the middleware. It is assumed that TCP connections are supported by the vast majority of programming languages. In addition, a simple connection between JavaScript modules and websites is possible via WebSockets. SCTP support is available in fewer programming languages, but is offered here because SCTP is very suitable for the communication interface with agents. In contrast to TCP, the SCTP protocol makes it possible to realise several message streams over one connection [Randall R. Stewart, 2007], which can improve performance. The middleware node cluster monitors all connected agents via their connection protocol and maintains a list of all connected components. Additionally, the platform can be monitored by the users via a web interface, which is also an agent implemented with web technologies that can be opened in a web browser. External messaging systems, like Java Messaging Services (JMS) or other middleware systems, can be connected to the system via the middleware API to allow transparent communication with other messaging systems.

The implementation of the middleware node layer is based on an Akka [Lightbend] cluster. Akka is a modern, high-performance and well-tested library, which is why it is used here for the implementation of the cluster management, the leader selection and the the TCP, SCTP, and WebSocket servers for client communication. The library allows an easy implementation of non-blocking servers and can help implement concurrent tasks, hereby meeting the requirements for many concurrent agents as clients and high message loads. By using standard network protocols and a Protobuf[1] API between agents and the middleware, no implementation details from Akka are passed to the clients. The agents can also be based on Akka, but it is also possible to connect to the middleware using other libraries or custom code.

Figure 4.5 shows the two recommended ways of connecting agents to the middleware. Firstly, an agent can be connected to the middleware via one of the frameworks provided (see Section 4.2.4). Here, an example agent is shown running on the Java Virtual Machine (JVM) and integrating the framework as a library. This allows the agent to communicate with native Java objects and leave the serialisation and deserialisation to the framework. Secondly, agents can be dynamically executed in runtime environments managed by the middleware (see Section 4.2.6). The agents in a runtime environment are managed by a control agent, which enables dynamic control of the agents in the system via an interface to the middleware. In addition, it is also possible for agents and other components to communicate directly with the middleware, as described above.

[1]Google Protobuf - https://developers.google.com/protocol-buffers - accessed 15.09.22

4.2.2 Messaging

Messaging is the most important functionality of this layer. To enable loosely coupled communication between the agents, a publish/subscribe system is implemented. The decision to use publish/subscribe for the communication is based on the literature evaluation in Section 2.3.3, which indicated that publish/subscribe communication is well suited for and often used in smart systems and especially IoT systems, such as MQTT or similar message protocols. The analysis of the requirements revealed that an architecture for smart system laboratory environments should support developers in understanding the system (see Section 3.2). In contrast to direct communication between agents, group communication, like publish/subscribe, offers the possibility of exchanging agents or letting other agents listen in without adapting the system. This loose coupling allows for easier insight into the communication and supports the important requirement that the system be open and assist developers with program comprehension and debugging tasks.

A topic-based system based on the formal modelling approach of Baldoni et al. [2003], is used in this architecture. Each message is explicitly sent to a group, also called a topic. A group is uniquely identified by a freely selectable string. Agents can subscribe to any group and will receive all messages sent to that group after a subscription until the subscription is terminated. Alternatively, a content-based publish/subscribe system could be used, which would allow subscribing not only to certain groups, but also to messages with certain properties sent to any group. A topic-based rather than a context-based system was chosen to keep the foundation of the system as simple as possible. The functionality to subscribe to messages with certain properties is possible via the functions of the CEP layer in the form of CEP queries. This functionality is not provided in the messaging layer such that each layer retains distinct functionalities.

There are no limitations or security mechanisms in the base system. Messages can be sent to arbitrary groups that can be subscribed to by all agents in the system. A group exists exactly as long as there is at least one subscriber. Messages to non-existing groups are discarded. Additional functionalities, such as access control and encryption, could be implemented for individual agents based on the publish and subscribe functions offered. Encryption and authentication are not offered as part of this messaging layer because the goal is to keep this layer as simple as possible. As mentioned in Section 3.2 during the requirement analysis, these security functions are not needed for laboratory environments.

All messages are delivered with an at-most-once semantic. There is therefore no guarantee that a message will be delivered to an agent, only that it will not be delivered twice. This semantic was chosen because it allows an implementation without performance constraints. In addition, guaranteed delivery of messages is not necessary in many of the environments for which this architecture is to be used. For example, if a sensor sends 60 measurements per second as messages to the system, it is not necessary that all of them arrive. A lost measurement can be replaced with the measurements received before or after it. Each message is given a unique ID consisting of the ID of the sending agent and a unique message ID generated by that agent. On the receiving side, it is checked that all messages are delivered to the agent's application logic at most once within a defined time interval. Higher guarantees for message delivery can be implemented on the basis of this semantic. For example, an exactly-once semantic could be achieved with confirmations of each message from the recipient and a re-send of the message if this confirmation is not received. However, this function would significantly increase the number of messages and would therefore burden the system. In addition, this function could lead to messages arriving late and the sequence of messages being changed. Without re-sending lost messages at the application layer, the used network protocols TCP, SCTP, and WebSockets can guarantee that messages from an agent to a group, when they arrive, are received in exactly the order in which they were sent, because the middleware nodes retain this guarantee.

4.2.3 Message Format

When selecting the message format for this architecture, three aspects had to be taken into account. Firstly, the format should follow a machine-readable structure that can be read with high performance. This is important for checking the validity of messages and for automatic processing of the message content. Secondly, the format should be human-readable, because the focus of this middleware is on supporting developers. It should be easy to read the network traffic and process it with simple text processing commands. Lastly, the format should be usable by many programming languages with as little effort as possible. Hence, it would be good if there were stable libraries for as many programming languages as possible that could parse the format. Performance is an important aspect, but it is not the main consideration in the selection process.

```json
{
    "type": "ControlLightRGB",
    "id": 177,
    "color": {
        "r": 100,
        "g": 0,
        "b": 0
    },
    "identifier": "sender",
    "options": ["debug", "timestamp"]
}
```

Listing 4.1: JSON message example

With the above criteria, several message formats come into question. These are mainly frequently used structured text formats, such as JSON and YAML. JSON was chosen because this format was already used for many other components in the laboratory environment in which the development of the messaging layer took place and it allowed easy integration of existing components and simple initial tests with the existing software.

Each message contains a unique *type* attribute, which allows the assignment to an interface definition of an agent and is intended to simplify parsing. All other attributes can be freely chosen by the developer in accordance with the agent's interface definition. Listing 4.1 presents an example of a JSON message to set a RGB light in a smart home with a specified ID to the colour red.

4.2.4 Programming Toolkit

For the development of new agents, a framework is provided that handles the connection to a middleware node, implements an auto-reconnect mechanism, and hides the serialisation and deserialisation of messages. This allows the developer to focus on the application logic and work directly with native classes and objects in the programming language, rather than, for example, worrying about parsing incoming messages and the associated error handling. The use of the framework is optional, but recommended for the implementation of agents with JVM languages and JavaScript. Alternatively, agents can communicate directly with the middleware via TCP or UDP messages or be connected via another communication interface and adapter agents. In this way, other programming languages that are not supported by the framework can be connected. There are multiple adapters for different programming languages available. Among others a native C library, which can be used by many other implementations over a foreign language interface.

The JVM framework follows an actor-based programming model. The implementation of the framework is also based on Akka [Lightbend], a JVM library for actor programming. Akka is very well documented, is actively developed, and is actively used in many projects and by many companies. The library is the most frequently used actor programming library for JVM languages and is therefore very well suited as a basis for the framework.

In the framework, an actor is a component that contains a mailbox and defines how to react to messages in the mailbox. The mailbox is implemented using a queue in which messages are placed in order of arrival. All messages to an actor are inserted into this mailbox and processed one after the other by this actor. The behaviour of an actor is defined in a receive method. It contains instructions for each of the messages expected by the programmer. Unexpected messages can be ignored, logged, or handled with own programme code. The processing of the messages in the mailboxes of the actors is carried out within a thread pool. This means that as many actors can work simultaneously as there are threads. If the thread pool is configured with at least as many threads as there are processor cores available, the available processor performance can be fully utilised because, if implemented correctly, the threads never block, for example to wait for IO.

In addition, Gradle is used as a build tool because it can be used for Java, Scala, and many other programming languages and is easily extensible. As part of the framework, the build tool was pre-configured to manage the dependencies of the framework libraries and to handle the publication of build artefacts, in this case the agents.

4.2.5 Interface Libraries

To define how the JSON messages being sent between the agents are structured, a simple message DSL is used, which can be seen in Listing 4.2. A Domain Specific Language (DSL) is a language that has been developed to fit a particular application domain. DSLs can have a higher expressive power for their specific domain and can be easier to use compared to generic programming languages [Mernik et al., 2005]. Here, a DSL is used to help developers define APIs.

With the help of the DSL, it is possible to determine how the messages of an interface between two or more agents are structured and which messages belong to it. The DSL is oriented around the possibilities and attribute types of JSON messages. It is possible to use strings, numbers, and booleans and to nest messages as desired. For example, Listing 4.2 defines an interface for controlling coloured lamps. With the message *UpdateLightSources*, several lamps can be set to the specified colours at the same time. The lamps are uniquely identified by an ID and it is possible to specify colours by their red, green, and blue values as well as a white value.

The DSL files can then be converted into class definitions or similar representations of a target language. This makes it possible to programme with automatically generated native objects in the target language. For example, in Java any message marked with the keyword *msg* would be converted into a class, and a *trait* would be converted into interfaces. The big advantage is that programmers can use type checking and auto-completion in the IDE if this is implemented in the language used. It also ensures that all messages sent in this way comply with the interface definition. The interface libraries are versioned to ensure that any changes to the interfaces are documented. This is necessary to detect whether two agents are also using the same version of an interface to guarantee that they both send and receive the same messages. An API generator, that generates the native class representation based on the DSL file is built into the Programming Toolkit and implemented for Java, Scala, and JavaScript. If another language is to be used, support can be added or self-built JSON messages can be used. In addition, the instructions for serialising and deserialising the messages are generated.

If the interface libraries are used, the framework transmits the expected interface name (as defined in the DSL) when an agent subscribes to a group automatically, otherwise, it has to be set manually by the agent's developer. This allows the middleware to check which messages are expected and whether the sender's library is compatible. It is also possible to enrich visualisations of agent communication with information from the interface definitions to help developers understand the dependencies between agents.

```
name LightControl
version 0.1
package de.hawhamburg.csti.actor.light

trait LightControlError

trait Color
msg RGB(r: Int, g: Int, b: Int) extends Color
msg RGBW(r: Int, g: Int, b: Int, w: Int) extends Color

msg LightSource(id: Int, color: Color)

msg UpdateLightSources(lightSources: Seq[LightSource])

msg ErrorUnknownLightSourceId(invalid: Seq[Int], requested: Seq[
    Int]) extends LightControlError
```

Listing 4.2: Light control API in the DSL of the framework

Furthermore, the framework contains functions for configuring agents. Configurations are stored in a JSON or YAML file, checked at the start, and then read in. The programmer can then access individual attributes of the configuration via methods in the framework and can adapt the behaviour of the agent accordingly. The special feature here is that this configuration can also be provided by the runtime environment and can thus be controlled by the middleware when new agents are started. This makes it possible to dynamically start agents with different configurations or to reconfigure parts of the system by restarting individual agents with a new configuration. The group names for communication via the middleware can also be defined in this configuration file. This makes it possible to programme in a reusable and customisable way. A simple example would be to write a logging agent that responds to messages in a configured group. This agent can be reused with different configuration files for different purposes. In addition, this further strengthens the loose coupling. If the communication groups can be easily adjusted when starting agents, it is possible to add more agents in a processing chain of agents to change or debug the behaviour.

In addition to the above-described functions, the framework offers support for the development of unit tests and integration tests. The framework's test kit allows agents to be started without a connection to the middleware, messages to be delivered directly to the agent, and expected messages to be easily defined. It also facilitates the handling of timeouts as well as the simulation of regular and delayed messages to test the behaviour in case of problems with the network.

It is important to note that the functions provided here to support developers come with performance penalties. For example, with JSON and text messages, serialisation and deserialisation may be slower than with binary optimised protocols, such as Protobuf[2]. Furthermore, keeping a list of all agents and checking their reachability produces an overhead. To ensure that the system meets the performance requirements, both the message latency and the scalability of the system will be tested as part of the evaluation to confirm that even high numbers of agents and messages can be handled (see Chapter 7).

4.2.6 Runtimes and Agent Migration

An agent runtime is provided so that agents can be dynamically executed by the system itself. The first implementations were based on the OSGi framework. OSGi supports component and service-oriented software development on the JVM [Tavares and Valente, 2008]. For this purpose, so-called bundles, or JVM software artefacts with additional metadata, can be used for tasks such as managing dependencies and the life cycle of the software. This approach is also already successfully used for context-aware applications, as in Gu et al. [2004] and Wu et al. [2007].

However, the use of OSGi comes with an overhead. All components must specify their dependencies and be tested with the OSGi runtime. It became apparent that the use of OSGi can complicate and slow down the development of components.

[2]Google Protobuf - https://developers.google.com/protocol-buffers - accessed 15.09.22

Since this contradicts one of the central requirements, the support for fast experiments in a research context, a separate runtime was developed that is limited to the most necessary features. This includes the controlled dynamic execution of components with a configuration that is passed on at the start. The implementation is based on a Java Class Loader that isolates the individual components from each other in to exclude conflicts in the dependencies and that also maintains complete control over the life cycle of the component. Agents can be started and stopped by the runtime via the middleware. In addition, the execution is monitored by the runtime and reported to the middleware. The management and monitoring of the agents in a runtime environment are implemented by a control agent that is directly connected to the middleware.

As soon as several runtimes are available, agents can be migrated between them. This fulfils the requirement for mobility in the system and allows agents to be restarted elsewhere in the event of a failure in one runtime. Another use case is to run an agent close to the user to reduce communication channels for interaction.

For an agent to be migrated without data loss, it must be stateless or keep its state in an external location. This can be achieved with an external database or implemented with event sourcing. Event sourcing systems use an event log to persist their state and decouple their communication [Lima et al., 2021]. All state changes are modelled and stored as events. In the case of an error, the event log can be read in and the state restored. To optimise this process, an agent can create snapshots of the state to summarise past events in the log. In this context, it is also possible to use the stored events for debugging, profiling, and anomaly detection [Lima et al., 2021]. Event sourcing is supported by the framework used, but its use is optional in order to avoid further hurdles for new developers and to keep development times short.

4.2.7 Integration with other messaging platforms

Connecting other messaging platforms can be useful to connect other parts of an existing system and to make information about the agents and their communication available to developers. Message brokers, such as MQTT, are often not designed to quickly look at the messages in a group or to demonstrate who has subscribed to those messages. In addition, the middleware can test the accessibility of agents from other platforms, if they have an interface that allows this. This would require, for example, a heartbeat message or a function to query the status of the agent.

Other systems can be connected to the middleware via adapters. In doing so, either all messages from the other system are transferred to one group or a group is created for each group in the source system.. All forwarded messages are provided with a proxy header, allowing the adapter to prevent loops and enabling debugging of the rest of the system. For example, software could be connected by reading incoming control commands in one group and publishing outgoing status messages in another, as with an actor. When connecting another message system, such as a message broker, all groups from the other system would be taken over, and all messages would be forwarded between the systems.

All groups created in this way can be given a prefix in their names to prevent collisions. This function can also be used to connect several middleware node clusters together, such as, to create a cross-lab network or to separate individual parts of a lab.

4.3 Complex Event Processing Layer

The Complex Event Processing Layer is built based on the Messaging Layer, which provides monitoring and messaging. The CEP layer enables the processing of messages as events with the help of SQL-like queries. On the one hand, the layer can be used to query and process context information and, on the other hand, it

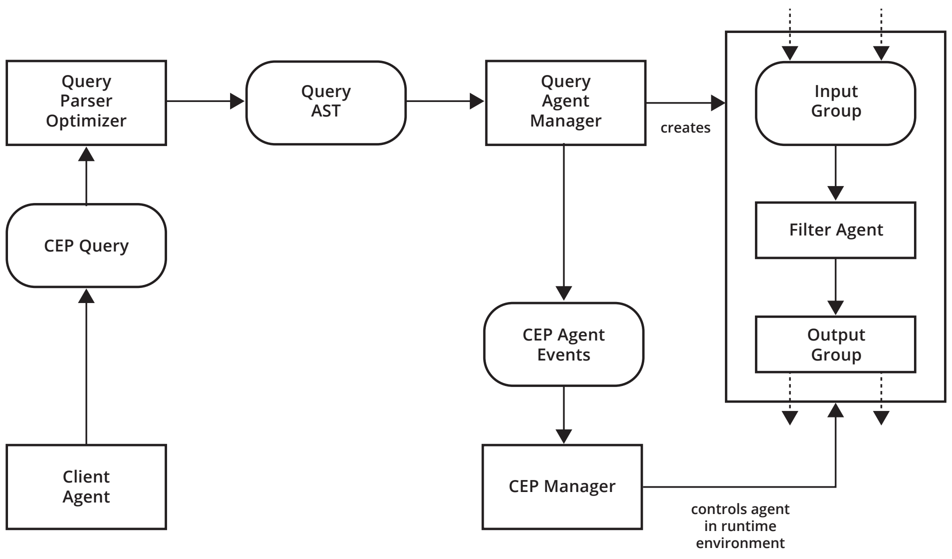


FIGURE 4.6: CEP layer components and interactions to create a new CEP query [Eichler et al., 2020].

can also be used to access information about the agents and the system provided by the middleware.

Instead of integrating an existing CEP engine into the system, the CEP functionalities are designed based on the design decisions of the messaging layer of the architecture presented here. This means that all components are agents and that the CEP engine exclusively processes the messages from the publish/subscribe system. In this way, the CEP functionalities are seamlessly integrated into the existing system. This seamless integration method and the architecture of the CEP layer were published in Eichler et al. [2020]. Since the goal is a CEP engine that is fully integrated into the agent-based system and that is based on the messaging layer, it has to be developed from scratch.

The CEP integration presented here uses the topic-based publish/subscribe service that allows messages to be sent to arbitrary groups. Furthermore, all messages in the system must follow a uniform structure that can be parsed. In this case, JSON is used, but other formats such as Extensible Markup Language (XML), YAML, and binary formats can be supported by additional parsers.

The CEP layer consists of three main components that are responsible for creating and managing agents for the execution of queries. The first component is the query parser and optimiser. This takes a query as a string, and transforms the query into a form that can be further processed and optimised. The second component is the Query Agent Manager, which creates all agents that implement CEP queries. The last main component is the CEP Manager, which manages all queries and monitors the execution of the queries. All these components are listed in Figure 4.6 and explained in detail the following sections.

4.3.1 Query Language

A query language similar to SQL is used for the CEP queries. The SQL queries are parsed and transferred into an Abstract Syntax Tree (AST), rendering the query language extensible and interchangeable. Thus, additional languages or features can be added without adapting the processing components.

Like SQL queries, all CEP queries consist of one or more clauses. The *SELECT* clause is at the beginning of the query and defines which attributes are present in the output messages. These messages are published in one or more groups that can be specified with the *INTO* keyword. The values for the attributes in the output messages can come from other messages, functions, or constants. Attributes in input messages can be referenced by specifying groups and attribute names. Constants can be specified with JSON values such as strings or numbers. In addition, there is a predefined set of functions for transformation, such as capitalize, for attribute generation, e.g. random numbers in a specified range, and for aggregation messages in time windows, such as min, max, and average.

The *WHERE* clause defines filters for the incoming messages. Here, free predicates with logical operators can be specified that refer to constants and message attributes. The *FROM* keyword specifies the groups from which messages are to be subscribed for the query. Further groups can be specified via *JOIN* clauses, and the mapping of messages from the groups can be defined with *ON* conditions.

Attributes and group names can be overwritten with the keyword *as*, which is helpful if, for example, the outgoing messages have a certain structure.

Listing 4.3 shows an example query that joins messages from the *first_input_group* and *second_input_group*, when they have an identical ID value. The WHERE clause filters all messages with an ID lower than or equal to 5 prior to the join. For each matching message pair according to the join predicate, the query publishes a single message. Each of these messages contains attributes with the following values: a constant value of 1, the ID of the matching messages, and *max_val* set to the value of the bigger *val1* variable of the two messages. All output messages are sent to the *output_group* as after by *INTO* keyword.

```
SELECT
   1 as constant,
   group_a.id as id,
   max(group_a.val1, group_b.val1)  as max_val
INTO output_group
FROM first_input_group as group_a
JOIN second_input_group as group_b
ON group_a.id == group_b.id
WHERE group_a.id > 5
```

Listing 4.3: Example CEP query.

4.3.2 Query Parser and Optimiser

CEP queries can be created from different sources. All agents can start new queries and manage existing queries via messages to the CEP engine. This can also be done by the provided frameworks to offer the programmer a native API in the programming language of the agent. In addition, queries can be started via the web interface of the middleware, which is especially useful for developers who want to query information about the system or create CEP queries interactively.

In the first step, all queries are checked and parsed by the query parser into an AST. This is a uniform tree-like structure for all queries that facilitates further processing and optimisation. Queries with syntax or semantic errors are returned to the requesting agent with a corresponding error message.

The AST is then passed to the query optimiser, which simplifies the query and optimises it for execution. In the process, unnecessary elements, such as tautologies from the filter clause, are removed from the queries. In addition, the AST is reordered so that as many filter elements as possible are at the beginning of the processing chain. This results in messages being filtered out as early as possible and thus no longer having to be forwarded to later processing steps. Further optimisation of the queries is possible by, for example, performing union operations or equality checks in an optimal order [Schultz-Møller et al., 2009] or by additional join query optimisations [Kolchinsky and Schuster, 2018]. However, this is not yet part of the implementation presented here.

After all optimisation steps have been completed, the AST is forwarded to the CEP query manager, which manages the agents for executing the queries. Further optimisations are carried out in the process. It is possible for an agent to take over several processing steps for a query. For example, a join clause can be connected with a filter step. In this and similar cases, the query manager tries to bundle as many related steps as possible into one agent, as this reduces the communication overhead and thus speeds up the execution of the queries.

Finally, all necessary agents for processing the query are started. If there is already an agent for a certain processing step, this is used instead of starting a new agent. This can happen if several queries contain the same filter clauses from the same groups. The responsible agent does not have more work due to additional queries; it receives all messages from the requested group, filters them, and forwards them to an output group that is subscribed to in the following processing steps. The reuse of agents is only possible for stateless processing steps, such as filters and transformations. Stateful steps, such as the aggregation of events in certain time windows, are always taken over by their own agents because otherwise the semantics of the query would change.

If necessary, individual processing steps are scaled by adding further agents. This is possible, for example, for filter operations and other stateless processing steps. The number of messages per agent is monitored by the CEP manager and as soon as the message load becomes too large, additional agents are started behind a load balancer, thus parallelising the processing. In addition, the agents are distributed to different runtime environments, so that the CPU and RAM load are distributed to different nodes.

Further optimisation of the implementation is planned for the future and is not the focus of this thesis. This includes further optimisation of the queries, for example by adapting the processing sequence [Schultz-Møller et al., 2009] and optimisations in the deployment and distribution of the processing agents [Cugola and Margara, 2013].

4.3.3 CEP Element Agents

Five different types of CEP elements are implemented to process the queries. Each step of the processing is taken over by an agent. Together, they are able to realise all the features of the presented query language. The elements used here are based on the reference model of [Paschke and Vincent, 2009] and equivalent components are also found in other CEP implementations such as Esper [EsperTech, 2021]. The difference of this implementation is the embedding in the agent-based system and the interface to the publish/subscribe layer.

The first type of CEP elements are sources. Sources generate events that can be further processed in the following steps. Generators can supply constant, random, or other algorithmically generated values, insert them into messages and publish them at specified intervals. Sources can also subscribe to messages from groups and forward them to the processing agents.

Elements such as apply and extract are used to modify the content of messages. Incoming messages are processed according to fixed rules and sent to the next agents in the processing chain. In this way, individual attributes can be extracted from messages or functions can be applied to attribute values.

With a join, messages from several input groups can be combined, which is needed to implement complex queries with several input groups.

Window elements group a set of messages together and send them on as one message. This is necessary for the aggregation functions, such as finding the message with the largest value in a certain time slot.

Finally, there are the sinks, which represent the end of the processing chain. In the architecture presented here, all messages resulting from the CEP queries are published in the specified output groups. Since this step does not change the messages, it is omitted in the implementation and taken over by the last processing agent.

The implementation of the CEP engine in the architecture presented here contains the following concrete agents that are used to execute all CEP requests. All agents are implemented using the presented framework based on Akka actors, permitting the processing of messages in a loop without much overhead and the forwarding of the results to the next agent. The necessary parameters for this work and the communication group for forwarding to the next agent are passed to the agent when it is created.

- **Generator** - A generator creates values based on constants and functions and inserts them into messages. This can be used, for example, for periodically scheduled messages. The generator is implemented as an agent that sends a single or multiple repeated messages to the output group.
- **Extract** - An extract-agent can extract one or more attributes of a JSON message based on its position. The extracted message parts are then packaged into a new message and forwarded. The extract-agent subscribes to the input group and tries to find the specified values in each received message. These values are then sent to the output group.

- **Apply** - Apply agents are used to execute functions on specific attribute values in messages. The parameters of the function are constants or values in the message. An example is the function *max(msg.a, msg.b)*, which forwards the larger value of the attributes msg.a and msg.b as the result. The Apply-Agent is given a function and uses it to generate values based on the specified inputs in the received messages. The execution of the function is handled in parallel.

- **Join** - A join-agent subscribes to two groups and tries to match the received messages according to the specified criteria. The output messages then contain a tuple with one element per message stream. Depending on the specified join type, only complete tuples are forwarded, or missing elements are replaced by a null value.

- **Sliding Window** - A sliding-window-agent collects messages from the input stream until a specified number of messages is reached and then forwards them together in an output message. With the help of an optional timeout, the waiting time for further messages can be limited.

- **Sliding Time Window** - An sliding-time-window-agent collects messages for a specified time period and forwards these messages together as one message. A timeout is also used for this.

4.3.4 Creation of Agent Graphs

The query agent manager is responsible for creating the agents needed to execute a CEP query. It decides, on the basis of the optimised AST which agents have to be created. For optimisation purposes, several nodes in the AST can be covered by one agent. The agents are started by messages to the control agents in runtime environments. The parameters are also written in the JSON messages and converted before an agent is started in a runtime environment. Normal communication groups with automatically generated names are used for communication between the individual processing agents. Only the output groups of the query use the group names specified by the user to allow interaction with the rest of the system. The communication groups are created by the middleware as soon as a subscription is requested by one of the created agents.

Figure 4.7 shows an agent graph created from the query in Listing 4.3. The given group names have been simplified for this illustration. Firstly, the two input groups are merged, applying the specified filter. The resulting messages are published in the intermediate group *joined_with_filter*. Then, the tuples from the two messages are filtered by an agent based on the *WHERE* clause. Finally, in the extract and apply stage, the values specified in the *SELECT* clause are extracted from the messages in the tuples, and the specified functions are executed. The messages created in this way are then published in the *output_group*.

Since only the input and output groups of the query interface with the rest of the system, the flow in between can be freely controlled by the CEP Agent Manager. For example, it would be possible to apply the filter from the *WHERE* clause directly in the join agent to speed up execution. However, this would reduce the traceability of the processing for developers and reduce the reusability of the individual agents. In addition, it would ensure that both processing steps must be executed in the same runtime environment and cannot be distributed. Since there is no optimal behaviour for all situations, the decision algorithm of the CEP Agent Manager can be adapted through configuration parameters.

By dividing the processing into several steps, the agent graph can also be retrieved via the middleware, as shown in Figure 4.7. The graph can be displayed for individual queries for debugging or as part of the overall system in an optionally filtered graph. The way of integrating the CEP functionalities presented here ensures that the queries are fully visible and traceable via the agent-based view of the system and the debugging tools of the middleware. For example, it is also possible to find out via these tools which query is responsible for the communication between two agents. This would not be possible if an external CEP engine were used.

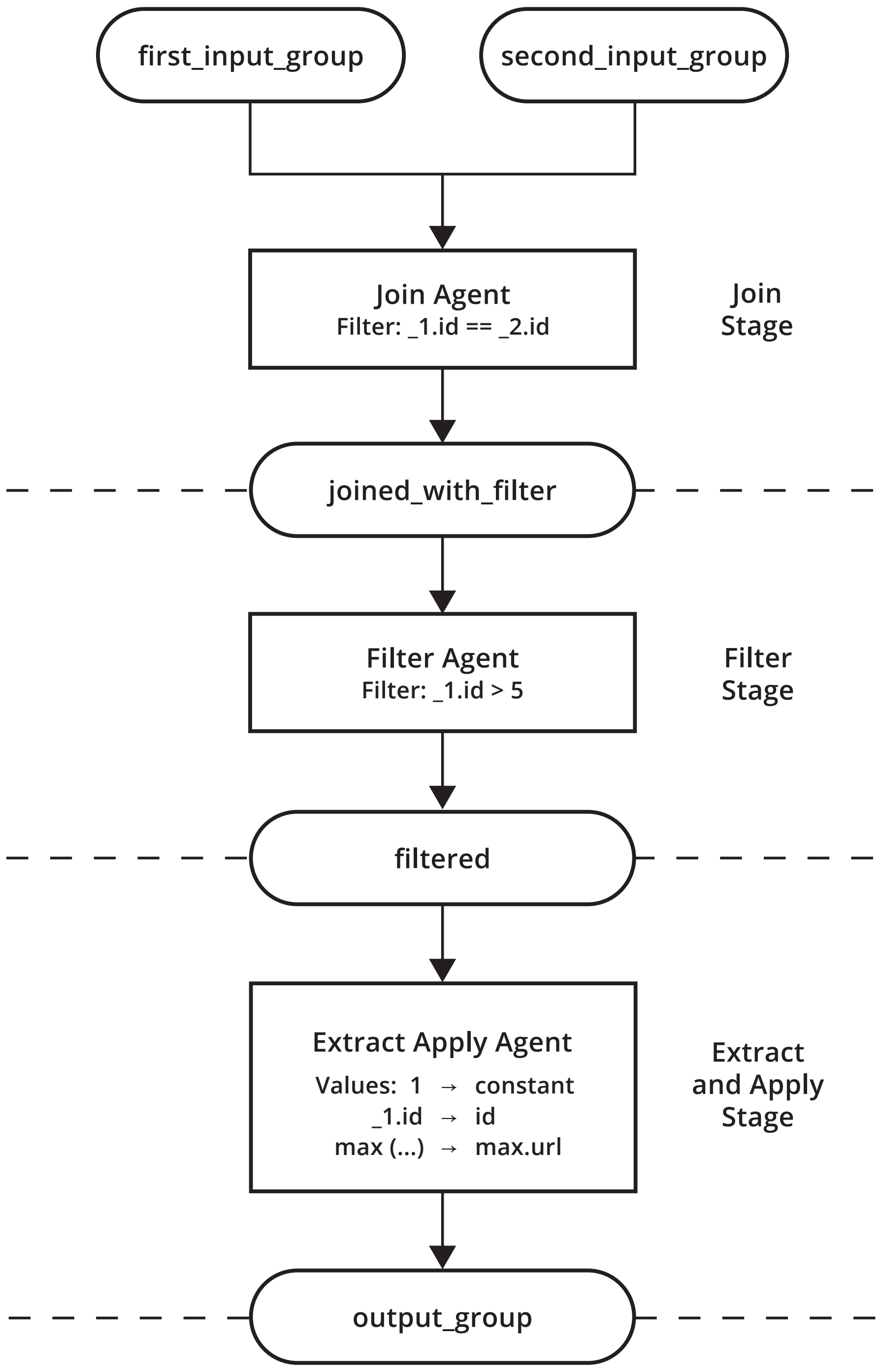


Figure 4.7: Graph of the example query in Listing 4.3 [Eichler et al., 2020].

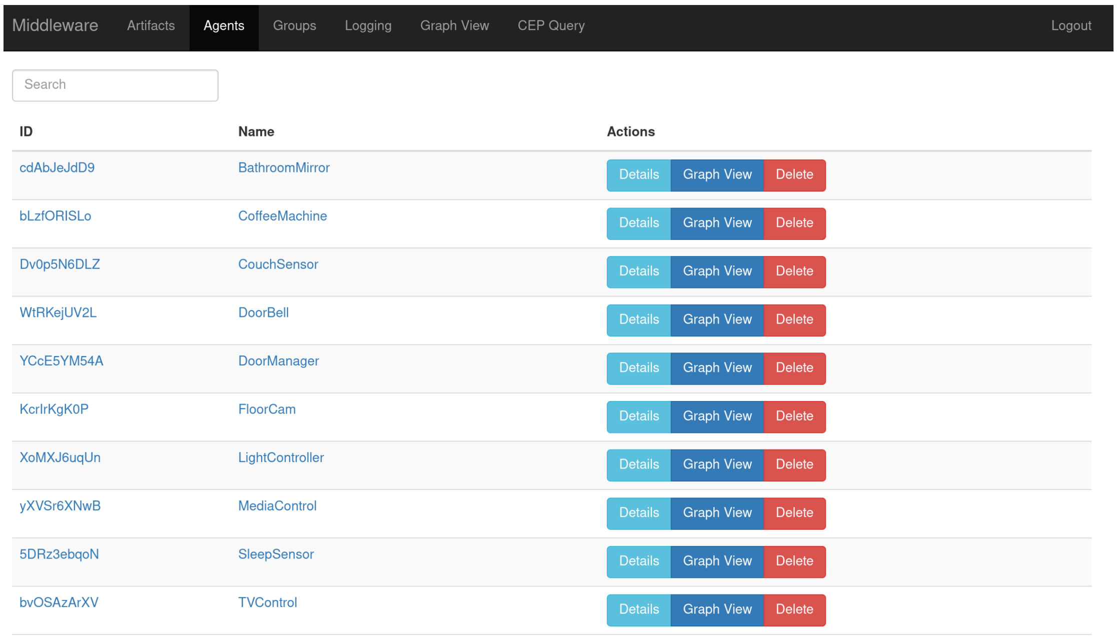
4.4 User Interaction Layer

The user interaction layer forms the interface between the layers already presented and the user. In this case, the developers who work with the system are seen as users. End users, such as someone who lives or works in a smart environment, are not the target group. Rather, developers are to be supported by the user interaction layer in their work with the system. This includes tasks such as program comprehension, debugging, the development and adaptation of components, and the execution of manual and automatic tests. Different user interfaces are offered for this purpose, which are explained in the following sub-sections along with the use cases.

4.4.1 Agent and Group Status Information

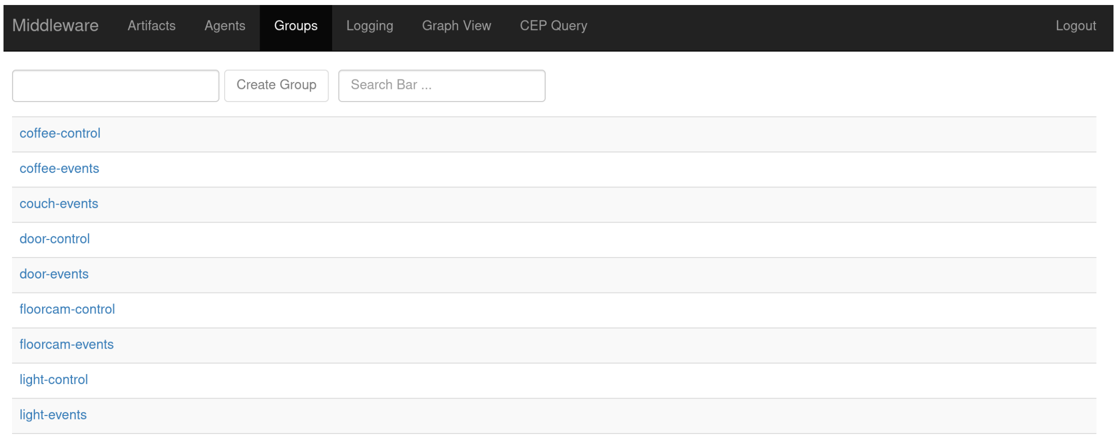
The messaging layer collects information about the agents, groups, and agents' subscriptions. This information is part of the registration of agents with the middleware and the subscription of groups via the middleware API, and it is automatically transmitted by the agent framework. The agents and groups are displayed as simple lists in the interface. For each agent further information can be retrieved, that was transmitted to the middleware. This includes links to the source code and documentation, dependencies and included interface libraries, and a short description of the function of the agent. An example of this is available in Figure 4.8.

Further information about the groups can also be displayed, including list of agents that have published messages to this group, a list of all subscribers, and an overview of the interface libraries used. Figure 4.9 provided an example of the group selection. In addition, when a group is selected or created, all new incoming messages are displayed live (Figure 4.10). For this purpose, the user interface subscribes to the group in the background and outputs the incoming messages chronologically in a readable format.



ID	Name	Actions
cdAbJeJdD9	BathroomMirror	Details Graph View Delete
bLzfORISLo	CoffeeMachine	Details Graph View Delete
Dv0p5N6DLZ	CouchSensor	Details Graph View Delete
WtRKejUV2L	DoorBell	Details Graph View Delete
YCcE5YM54A	DoorManager	Details Graph View Delete
KcrIrKgK0P	FloorCam	Details Graph View Delete
XoMXJ6uqUn	LightController	Details Graph View Delete
yXVSr6XNwB	MediaControl	Details Graph View Delete
5DRz3ebqoN	SleepSensor	Details Graph View Delete
bvOSAzArXV	TVControl	Details Graph View Delete

FIGURE 4.8: Searchable list of all active agents in the user interface.

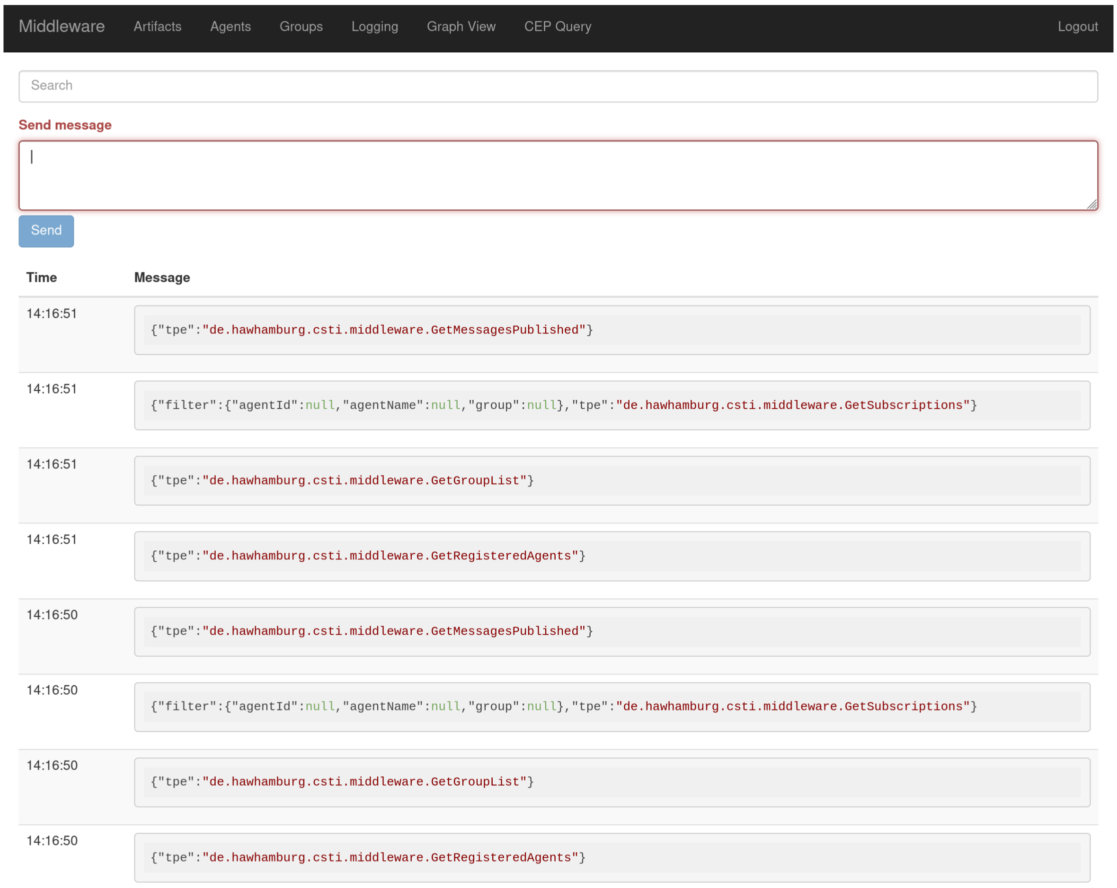


coffee-control
coffee-events
couch-events
door-control
door-events
floorcam-control
floorcam-events
light-control
light-events

FIGURE 4.9: Filterable list of all known groups in the user interface.

As the middleware monitors the availability of the registered agents, agents that lose their connection are automatically removed from the lists. The disconnection is then logged for the user. In addition to this information, the user interface displays log output from agents. The presented framework is preconfigured in such a way that all log output is automatically sent asynchronously to a specific middleware group for each agent. This allows developers to browse the log output in a central location and to quickly access the output during debugging.

In addition, information about the runtime environments and the agents executed in them can be displayed via the user interface of the middleware. The interface



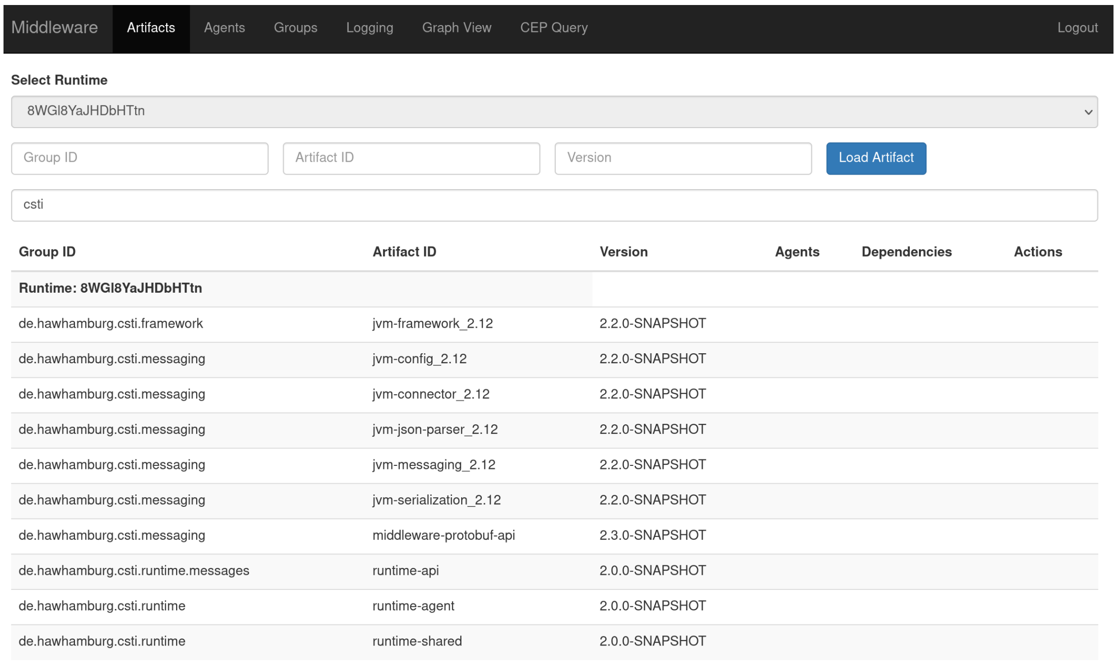
Time	Message
14:16:51	`{"tpe":"de.hawhamburg.csti.middleware.GetMessagesPublished"}`
14:16:51	`{"filter":{"agentId":null,"agentName":null,"group":null},"tpe":"de.hawhamburg.csti.middleware.GetSubscriptions"}`
14:16:51	`{"tpe":"de.hawhamburg.csti.middleware.GetGroupList"}`
14:16:51	`{"tpe":"de.hawhamburg.csti.middleware.GetRegisteredAgents"}`
14:16:50	`{"tpe":"de.hawhamburg.csti.middleware.GetMessagesPublished"}`
14:16:50	`{"filter":{"agentId":null,"agentName":null,"group":null},"tpe":"de.hawhamburg.csti.middleware.GetSubscriptions"}`
14:16:50	`{"tpe":"de.hawhamburg.csti.middleware.GetGroupList"}`
14:16:50	`{"tpe":"de.hawhamburg.csti.middleware.GetRegisteredAgents"}`

FIGURE 4.10: Sending and receiving messages from one group via the user interface.

view in Figure 4.11 shows a list of all software artefacts loaded in the selected runtime environment. If these artefacts contain agents, they are listed separately and can be controlled via additional buttons. In this way the user can start agents from artefacts, query their status and stop them again. The artefacts can be filtered via the input fields and further artefacts, including the dependencies, can be loaded dynamically.

4.4.2 Agent Communication Graph

To represent the communication structures between the agents, the user interface can visualise them in a graph. Agents and groups are represented as differently coloured nodes connected by directed edges, which represent the message flow. A subscription forms a directed edge from a group to the subscribing agent. If an



Select Runtime

8WGl8YaJHDbHTtn

Group ID | Artifact ID | Version

csti

Group ID	Artifact ID	Version	Agents	Dependencies	Actions
Runtime: 8WGl8YaJHDbHTtn					
de.hawhamburg.csti.framework	jvm-framework_2.12	2.2.0-SNAPSHOT			
de.hawhamburg.csti.messaging	jvm-config_2.12	2.2.0-SNAPSHOT			
de.hawhamburg.csti.messaging	jvm-connector_2.12	2.2.0-SNAPSHOT			
de.hawhamburg.csti.messaging	jvm-json-parser_2.12	2.2.0-SNAPSHOT			
de.hawhamburg.csti.messaging	jvm-messaging_2.12	2.2.0-SNAPSHOT			
de.hawhamburg.csti.messaging	jvm-serialization_2.12	2.2.0-SNAPSHOT			
de.hawhamburg.csti.messaging	middleware-protobuf-api	2.3.0-SNAPSHOT			
de.hawhamburg.csti.runtime.messages	runtime-api	2.0.0-SNAPSHOT			
de.hawhamburg.csti.runtime	runtime-agent	2.0.0-SNAPSHOT			
de.hawhamburg.csti.runtime	runtime-shared	2.0.0-SNAPSHOT			

FIGURE 4.11: Display of all artefacts in a runtime environment with search and interaction possibilities with the loaded agents.

agent publishes at least one message to a group, a directed edge is added from the agent to this group. Edges resulting from subscriptions are removed when the subscriber terminates or unsubscribes from the group. Edges indicating published messages are removed when no message were sent for a configurable time window, the sender terminates, or there are no subscribers left for the group to which the messages where sent. Figure 4.12 depicts an agent communication graph for the middleware user interface.

The thickness of the arrows indicates how many messages have been sent via a group within a configured time interval. The number of messages is weighted relative to the group with the most messages to determine the thickness of the arrow. In this way, it is immediately visible where there is currently the greatest message activity in the system.

The displayed graph can become unmanageable for larger systems. Figure 4.13 gives a more complex example with components from a smart home laboratory. The example contains various sensors, such as temperature, sitting position, and sleep detection sensors, which provide information about the home and the user.

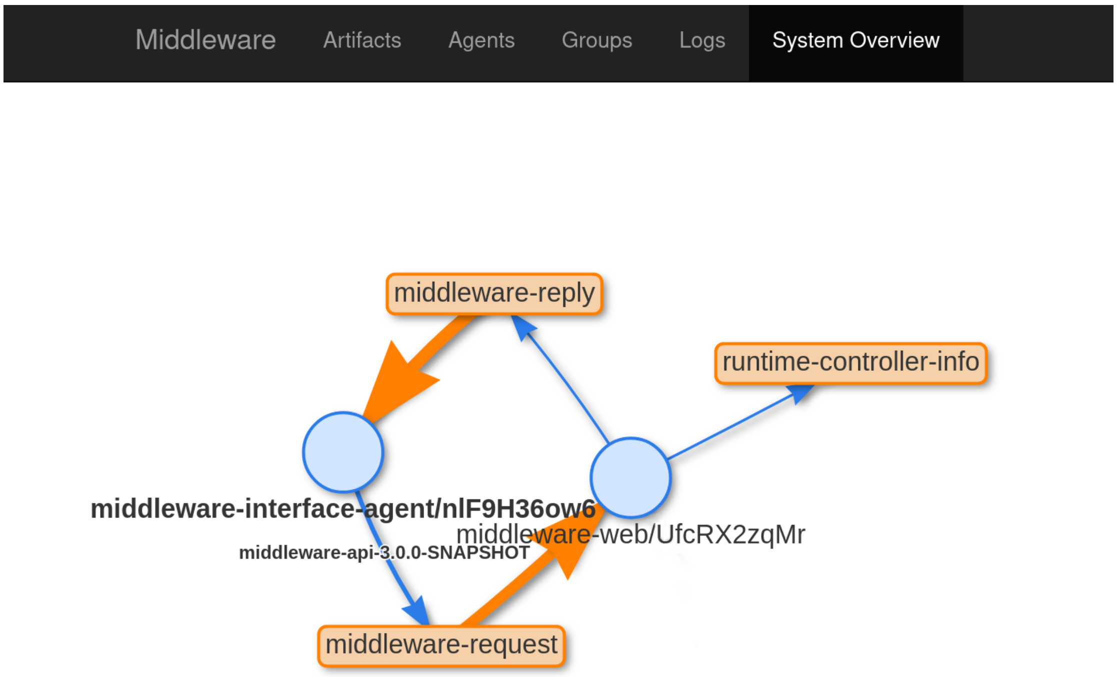


Figure 4.12: Agent group graph that shows the communication paths between the middleware interface and the middleware web interface

This information can be used to support a user's morning routine. Among other things, information can be displayed on various displays, such as a touch table, the television, and the bathroom mirror.

To reduce the complexity and size of the graph, it is possible to limit the graph to a certain part of the system (Figure 4.14). This is done by specifying an agent or group in the system and how far from this point the graph should be displayed. All outgoing and incoming paths in the graph up to this point are included if they are less than or equal to the maximum specified visibility range. This allows a developer to analyse certain parts of the system without having to deal with the complexity of the whole system. More complex queries and filters are possible via a later-introduced interface with CEP queries.

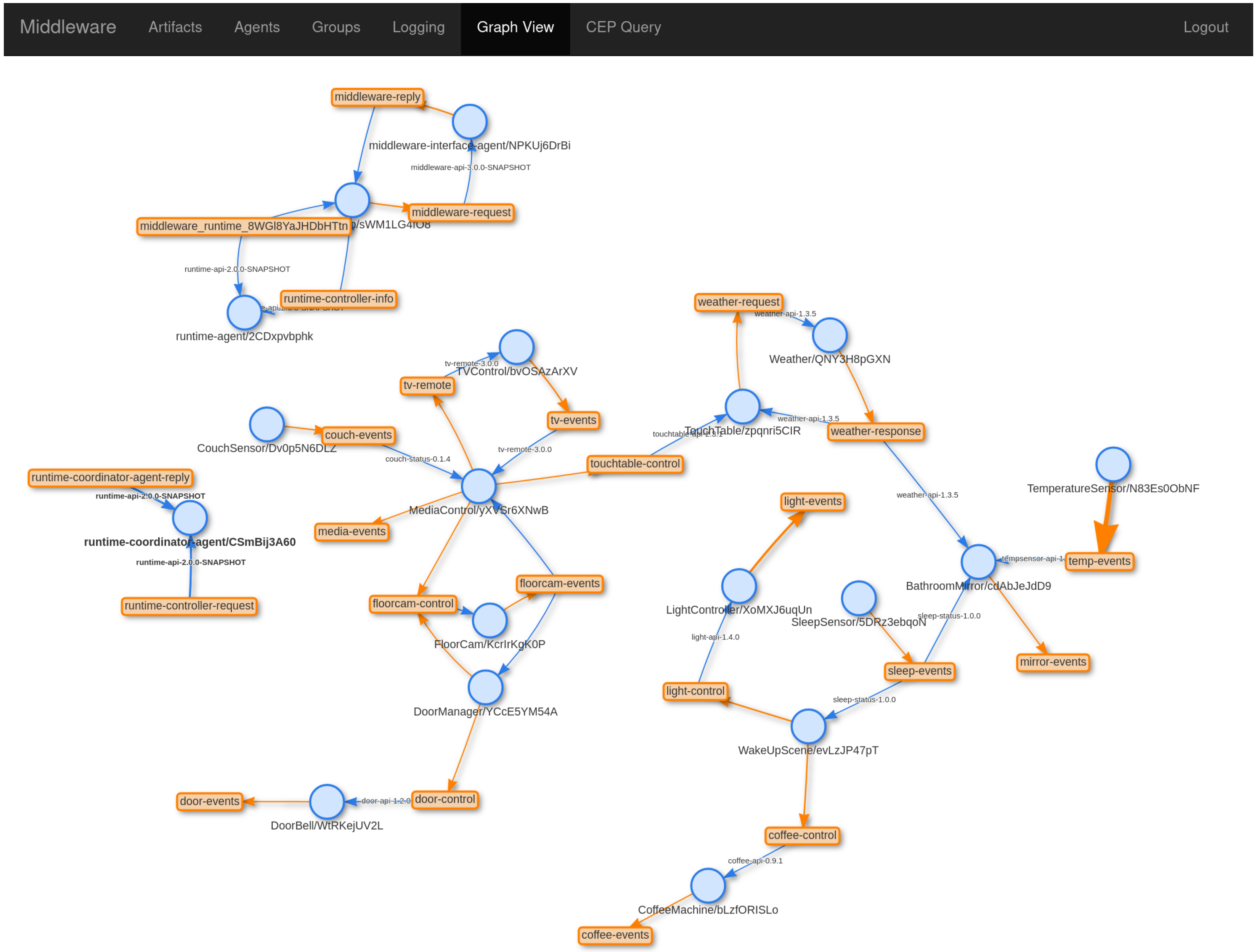
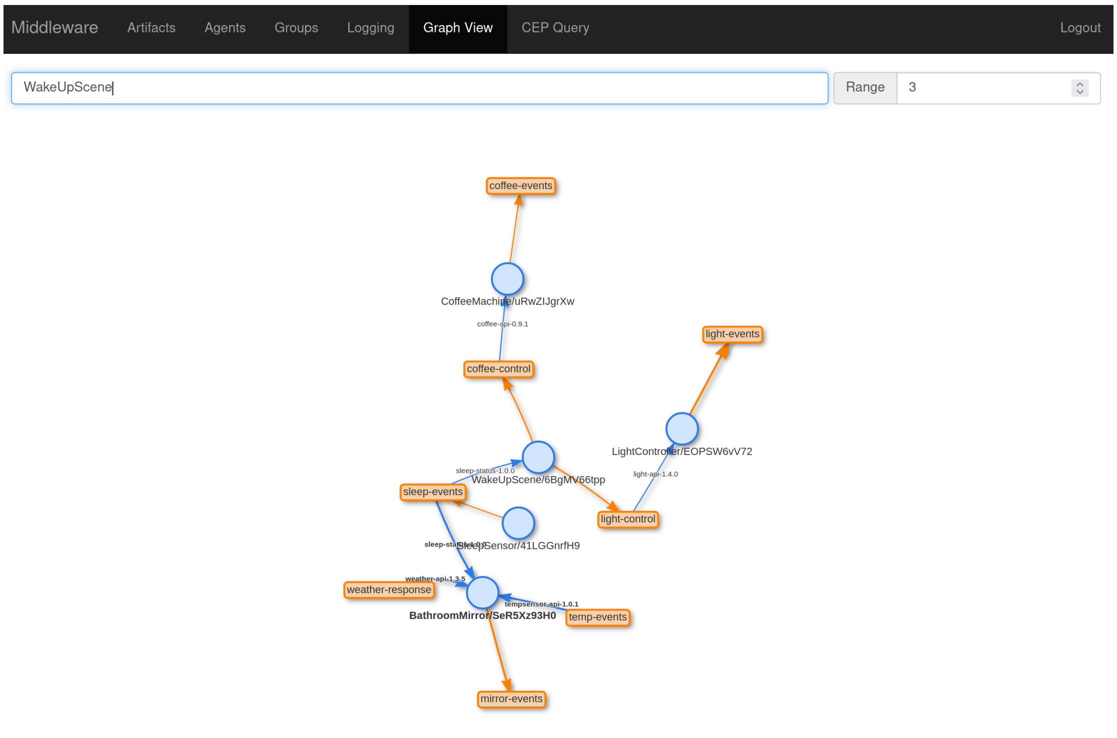


FIGURE 4.13: Example of a agent group graph based on components in a smart home environment



WakeUpScene

Range 3

FIGURE 4.14: Filtered agent group graph based on the graph in Figure 4.13 exhibiting only components that are in the range of three edges of the WakeUpScene agent

Changes in the system are updated directly in the graph display. As such, if the connection to an agent is broken and the middleware decides that this agent is no longer active, the corresponding node and its edges are removed from the graph. The display algorithm was chosen in such a way that the positions of the remaining nodes and edges change as little as possible during an update so that a user does not lose the overview in case of changes.

All nodes in the graph can be repositioned by the user via drag and drop and kept in this position as long as the page is displayed. This allows the user to organise individual parts of the graph appropriately or to read any hidden information if a graph cannot be displayed without overlaps.

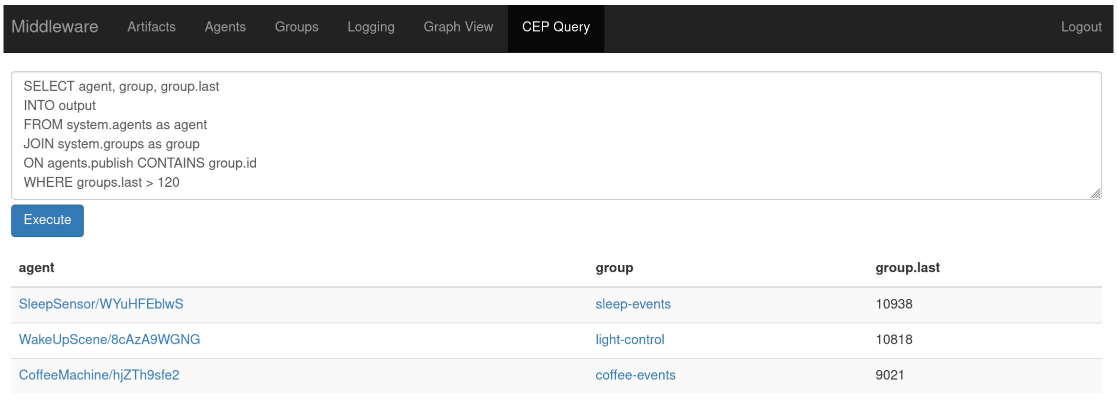
The agent communication graph has proven to be a powerful and versatile tool for interactively answering questions about the structure of the system and exploring the relationships between agents. These questions are important for debugging system components and can speed up the work of developers.

4.4.3 Complex Event Processing Queries

Thus far, only user interface functions provided by the messaging layer have been presented. By using the CEP layer, these functions can be made more extensible and accessible to improve developer support, which can be especially helpful for the developers because they are already familiar with the query syntax if they used it during the implementation of new components.

CEP queries can be used instead of a subscription when implementing agents, and they provide more control over message filtering. An additional advantage of using CEP queries is that they can be developed and tested live, directly in the middleware user interface. The query interface is presented in Figure 4.15. It allows ad hoc and interactive queries, and then uses them unchanged for the implementation of the agents. This helps in the development of components, because the feedback on whether a query works as expected is much faster than if an agent has to be compiled, and executed, and the outputs checked.

Simple agents can even be implemented entirely using CEP queries and started via the middleware interface. For example, an agent might be needed to filter the messages in a group, split them among several other groups, or reformat them to make them compatible with other existing components. Instead of implementing new agents or adapting existing components, CEP queries can be deployed to implement these functions. This could also be achieved with the integration of a existing CEP engine or a rule-based machine. The decisive advantage here is that simple agents are started when deploying CEP queries via the middleware, and they are therefore seamlessly integrated into the overall system and are transparent via the presented user interfaces and from the perspective of the rest of the system. From the outside, there is no difference between a function implemented directly via an agent or via a CEP query. This is important because otherwise all components and developers would have to potentially distinguish how agents were implemented when interacting with them. It would not be possible to generate an overall view of the flow of a request in the system without having to handle two different paradigms.



```
SELECT agent, group, group.last
INTO output
FROM system.agents as agent
JOIN system.groups as group
ON agents.publish CONTAINS group.id
WHERE groups.last > 120
```

agent	group	group.last
SleepSensor/WYuHFEblwS	sleep-events	10938
WakeUpScene/8cAzA9WGNG	light-control	10818
CoffeeMachine/hjZTh9sfe2	coffee-events	9021

FIGURE 4.15: CEP query based on data from Figure 4.13 listing all groups where agents have posted messages but which have not done so for more than two minutes.

These simple agents, implemented with CEP queries, can also be used to simulate sensor data or produce messages in the test, which can be useful when creating unit tests as well as during debugging. For example, the output of a temperature sensor could be replaced by a request that sends random values within a certain range or a fixed value at configured intervals. This allows testing that is quick and independent from the sensor. More complex sensors could be simulated by recording messages. With the help of a CEP query, it is easy to display all messages from an agent in a group. The messages can then be exported via the middleware user interface. Afterwards, it is possible to send these messages in a loop via another CEP query or an agent to allow controlled tests with this data.

All functions described so far are based on interaction with the underlying publish/subscribe system. To further expand these possibilities for interaction with the system, the CEP engine has been extended with additional functions. On the one hand this allows ad hoc queries to be made about the state of the system, and on the other hand, it permits the usw of CEP queries as a filter for the agent communication graph.

For this purpose, two virtual groups are created, with the names *agents* and *groups*. They contain the active agents and groups inside the system at any time. Analogous to the messages in a normal group, these special groups also contain entities with attributes, such as the name, the time of creation, and others. Hence, the query *SELECT agents.name FROM agents* returns a list of all active agents. As long as the query is active, the delivered results are adjusted for changes in the system. For example, additional agents appear dynamically in the results list, and terminated agents are removed. With these functions, it is possible to query status information about the system, such as to generate a list of all groups on which no messages have been sent in the last five minutes or a query that returns all agents that have sent error messages. The result can be enriched with information regarding which group these error messages were sent to and how long ago this occurred.

Queries about the state of the system can also be used for testing and verification. CEP queries can check that no error messages occur and display accumulated errors in test results. In addition, it is possible to formulate expected states for the system in CEP queries and check them. For example, it could be checked whether at least one agent is serving an interface to a certain group at all times. As soon as certain states are communicated via the message traffic, they can also be included in the queries. An example of this would be a leader election in a group. Using the messages and the status information about the agents, a query could determine whether there is a leader at any time after at least five minutes. This is also an example of how normal messages from agents can be used together with status information from the middleware, because they are made uniformly accessible here via CEP queries.

A complex system with many agents and groups would generate too many events and graphs that are too large to be easily understandable by developers, but the CEP queries can be used to filter the information. Events regarding the system topology can be filtered by the distance to a point of interest inside the graphs. Two different agent graphs can be used as a metric to provide a distance for this kind of filtering: the agent communication graph, which was explained earlier, and the agent creation graph. The latter graph connects an agents with all its child agents with the middleware as the creator of the first agents.

Since there is no difference between the virtual agents and groups and the normal groups from the publish/subscribe system, this opens even more possibilities. These include the combination of system information with messages from the system and the use of system information by agents. For example, a load balancer could be implemented using this information. An agent starts a query for a list, of all agents that implement a certain interface. The query additionally contains the group names of the subscriptions of these agents. Unreachable agents are automatically filtered out of the list and changes are sent to the requesting agent. This agent can then forward any incoming message on a configured group to one of the agents from the query results, thereby acting as a load balancer. This example can be seen in Figure 4.16. Another possible application would be an agent that queries the reachability and sending behaviour of another agent. The requesting agent would then be able to react to changes in the behaviour of the monitored agent and act accordingly. A possible action would be to replace the supposedly faulty agent or to inform the user.

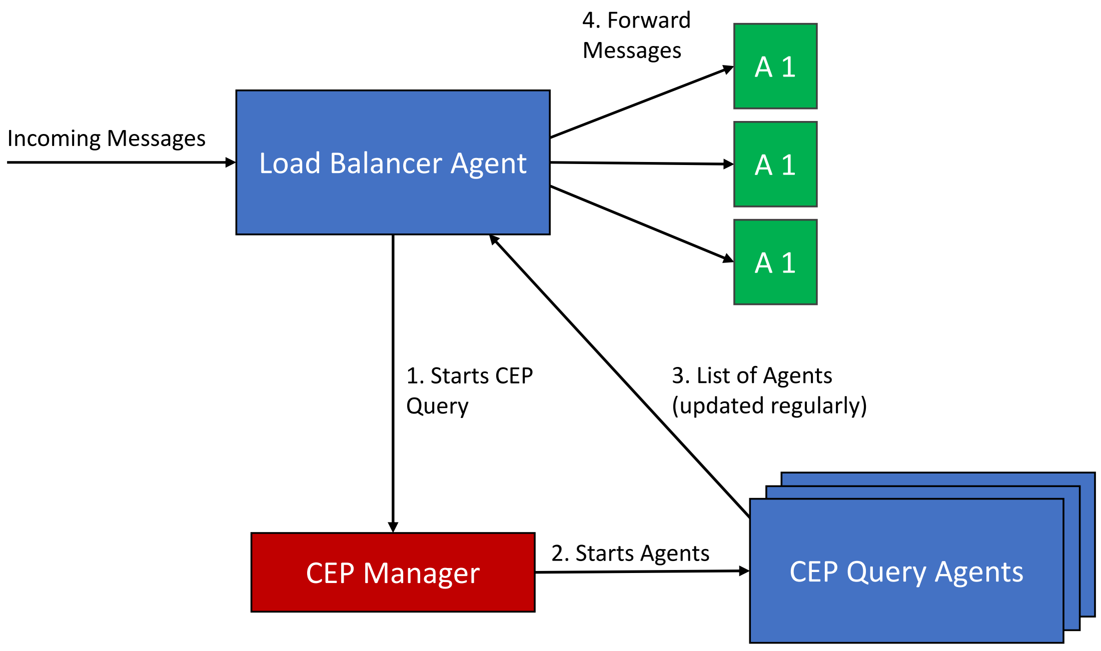


FIGURE 4.16: Example of an implementation of a dynamic load balancer with a CEP query, which provides an updated list of agents to which messages are forwarded.

4.4.4 Case Studies

This section presents two short case studies, as examples of how the interaction with the agent communication graph and CEP queries works. These case studies were published in Eichler et al. [2020] and both cases were used in multiple research projects in the CSTI.

A case study is a method of examining an empirical case in detail [Farrow et al., 2020] and is often used to analyse organisational, political, social, and related phenomena [Yin, 2010]. However, case studies are also used to evaluate methods, as in Clements et al. [2009] and are often utilised to demonstrate the suitability of software architectures (see Section 2.3.4).

The aim of the following case studies is to showcase different interaction possibilities with the user interaction layer to illustrate how the functionalities described here can help with program comprehension and debugging tasks. The evaluation of the presented architecture is done separately and is explained in Chapter 5.

4.4.4.1 Omnidirectional walking-in-place detection

The first application is an omnidirectional Walking-In-Place (WIP) tracking system, which can be used for Virtual Reality (VR) experiments. The first research publication using this system was published in Langbehn et al. [2015]. Subsequently, this installation was used for several student theses in the CSTI.

WIP is a method to enable the movement of users in a VR world. The user wears a head-mounted display and walks in place to move forward in the virtual world. The direction of the user's gaze can be used to control the direction of movement. There are various ways of recognising the user's steps, such as using acceleration sensors on the feet. In the experiment planned here, it was intended to test whether the immersion of the user increases when the speed of movement can be controlled by leaning forward. Therefore, it was decided to use a skeleton

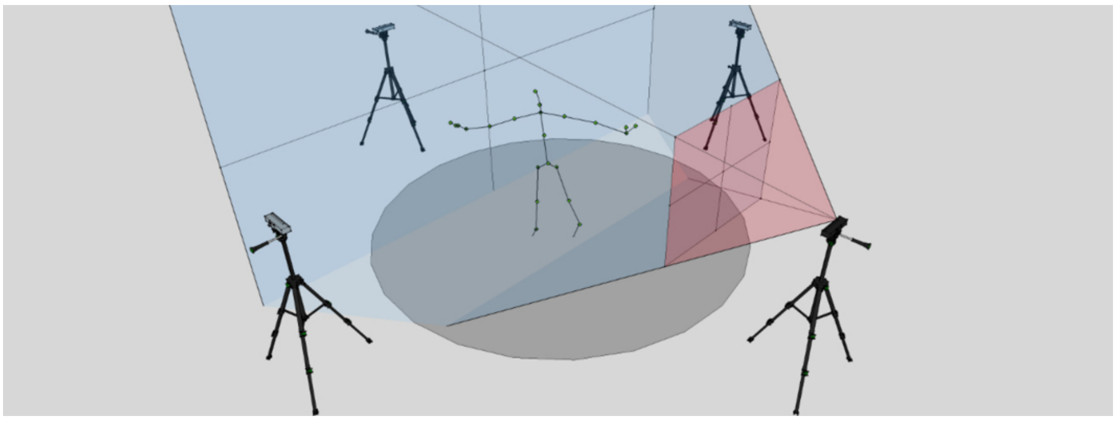


Figure 4.17: Omnidirectional tracking setup with four sensors [Langbehn et al., 2015]

detection sensor, such as the Microsoft Kinect 2[3], for both step detection and forward lean angle detection. Since the user should be able to rotate freely in the real world while interacting with the VR world, several sensors were used to track the user from different viewing directions. The measurement results were then transported in real time via the network, calculated and transferred to a common coordinate system. This was also intended to increase the measurement accuracy and reliability of the sensors.

Figure 4.18 displays the tracking system with 4 Sensors (k1–4). The sensor data is collected by a sensor fusion agent, and the output is then used by the VR application.

```
SELECT
   Vector3D(2.3, 3.4, 1.2) as head.position,
   Vector3D(2.3, 3.3, 1.1) as spine_mid.position,
   [...]
INTO skeleton_data
FROM src.periodic(1000)
```

Listing 4.4: CEP query to generate skeleton data to simulate a sensor (truncated)

In this system, a skeleton consists of 27 joints that are detected by each sensor every 30 milliseconds. The data collected in this way is difficult for humans to

[3]Kinect for Windows – https://developer.microsoft.com/windows/kinect, accessed 10.06.22

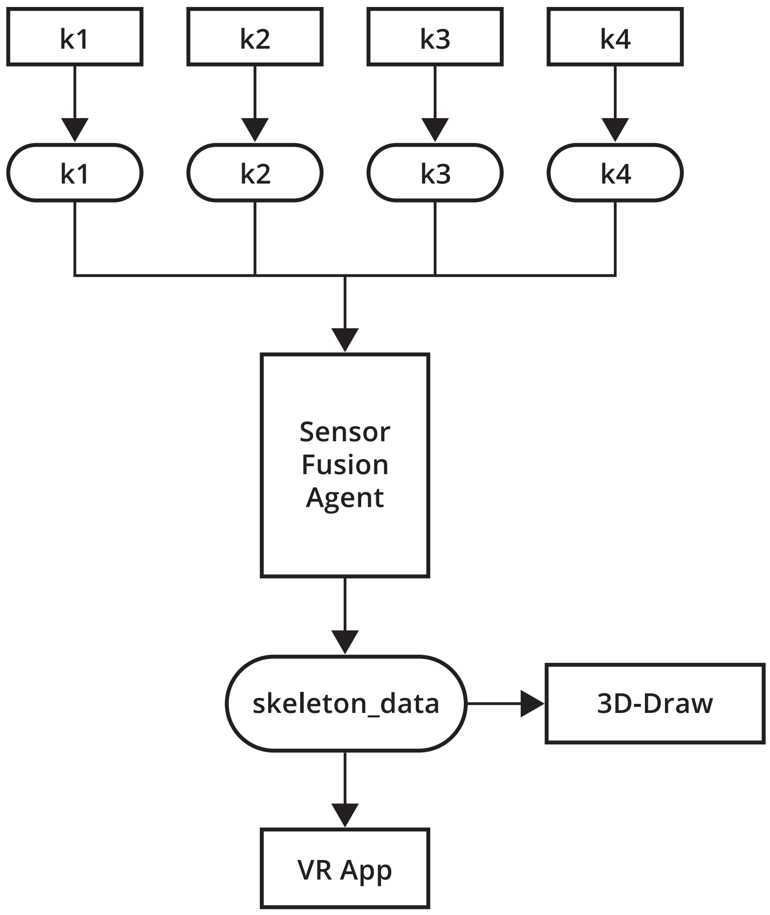


FIGURE 4.18: Overview of all agents and groups that are used to implement the omnidirectional walking-in-place detection [Eichler et al., 2020]

understand because there are a large number of three-dimensional vectors. To examine the data, a visualisation on a 3D canvas similar that in Figure 4.17 can help. Listing 4.5 shows a CEP query that extracts the relevant data for a visualisation from the sensor data and passes it to a drawing component. This procedure is possible for single, multiple, or merged sensor data and can help debug the sensors or the sensor fusion algorithm.

```
SELECT
   head.position,
   spine_mid.position,
   [...]
INTO draw_3d
FROM skeleton_data
```

Listing 4.5: CEP query to draw selected skeleton joints on an 3D canvas (truncated)

The WIP detection can be implemented using the sensor data as a CEP query based on the sensor data. The corresponding query is shown in Listing 4.6. For this purpose, the Fusion Agent calculates the distance between the floor plane and the user's foot bone for each update of the skeleton data. If this distance is greater than a predefined threshold, a message is sent to inform other agents that a step has been detected. Since the skeletons are updated much more frequently by the sensor than the position of the floor plane, the latter information is stored by the CEP engine, and the last state in the join is always used.

```
SELECT
   1 as step_detected,
   distance_point_floor(
      s.bone1.x, s.bone1.y, s.bone1.z,
      f.a, f.b, f.c, f.d
   ) as foot_height
INTO wip_events
FROM skeleton_data as s
JOIN floor_data.keep as f
WHERE foot_height > 3
```

Listing 4.6: CEP query to implement walking-in-place detection based on a skeleton sensor

If there is a problem with the sensors or the sensor fusion, the cause must be found. The agent group graph can help investigate such error cases. The graph in Figure 4.19 indicates that sensor k3 is not accessible because it is not displayed.

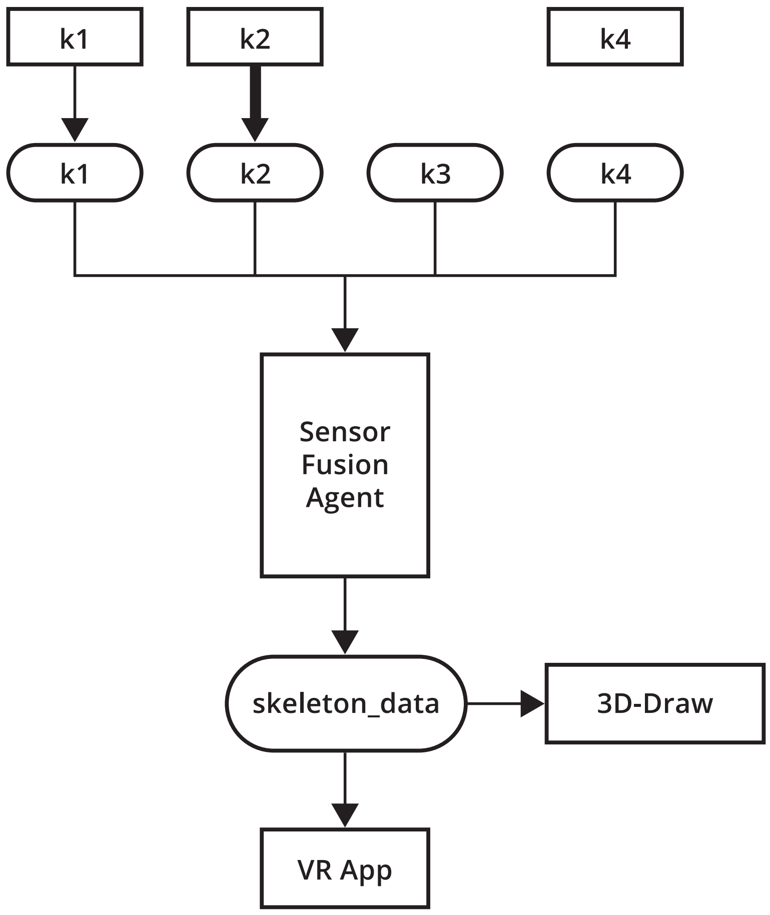


FIGURE 4.19: WIP agent graph with message throughput indicated by arrow width, which can help to debug the system [Eichler et al., 2020].

This could be a connection failure or a defect in the sensor. In addition, k4 does not send any messages, which indicates that there are also problems with the sensor. Since k4 is reachable, further information about the agent could be requested by the middleware. This could clarify whether the sensor has already delivered data and whether any error messages have been logged.

Other errors can also be detected in the graph. For example, the thickness of the arrows reveals that k2 sends more messages than k1, suggesting that k1 has problems with the recognition of the skeleton and therefore sends fewer data updates. The sensor fusion agent sends a similar number of messages as k1 because the fusion agent is waiting for messages from all available sensors to be added together. As a result the application will continue to run at a lower update rate.

When training new students in the research lab, the interfaces of the system are visible via the web interface of the middleware. In addition, the data can be displayed and visualised. Through the publish/subscribe communication, several projects can access the data simultaneously without affecting other parts of the system. Due to this modular implementation of tracking and sensor fusion, the application could be used in many different projects. Over time, this has resulted in a generic tracking framework that can process the position data from different sensors and play them out in a common coordination system. Among other things, gesture recognition for the control of different actuators and other interactions were implemented.

Because of the message-based implementation of WIP detection via the middleware, the project could be reused later and was used for further VR projects. One of the projects, which was published in Becker et al. [2019], involved the comparison of different forms of interaction for movement in impossible virtual worlds. In this case, the application allowed the user to walk on the walls and ceiling of a virtual room.

4.4.4.2 Processing environmental sensor data

The second application example is the processing of environmental sensor data. This is often implemented in smart homes to collect contextual information about the user's environment. For example, two sensors are used to measure the air quality. The first sensor is a carbon dioxide sensor (eCO2) and the second is a particulate matter sensor (PM10). The sensor data is to be processed further to be able to make statements about the air quality and this information can then be used to control an air filter system or to automatically air the smart home together with a window control. Both sensors measure every one to three minutes and publish the results as messages in the *sensor_data group*. Each message contains an identifier of the measurement (e.g., eCO2), a timestamp, the measured value, and the unit. The interpretation agent subscribing to the *sensor_data group* takes the readings and periodically publishes an assessment of the air quality from 0%

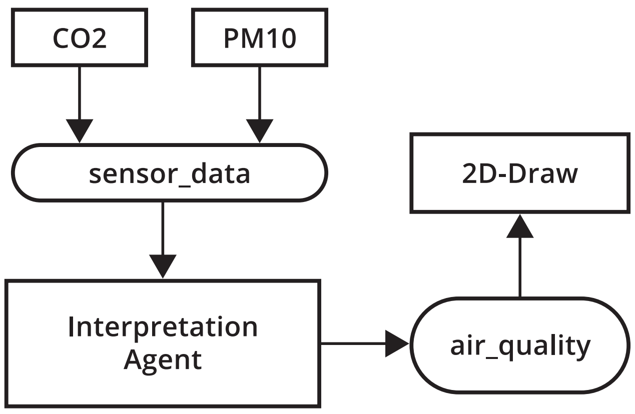


Figure 4.20: Overview of agents and groups to process air quality sensor measurements to automate ventilation in a smart home environment [Eichler et al., 2020]

to 100%. This value can then be used to control actuators in the home or for visualisations for the user.

To test the implementation of the interpretation agent independently of the sensors, CEP queries can be used to generate suitable test data. This is especially useful for sensors where the sensor data is rarely updated and cannot easily be quickly altered for testing, as is the case with the air quality sensors. The query in Listing 4.7 creates an agent that generates a message with random values every minute and publishes it to the specified group. The messages created in this way are identical to the messages from the sensor and are therefore well suited for testing. By adjusting the query, the interval or the values of the messages can be quickly changed to test further properties of the agent.

```
SELECT src.ec02
INTO sensor_data
FROM src.random.num(60000, 30, 60)
```

Listing 4.7: CEP query to simulate a sensor with random data in a specific range

The implementation of the interpretation or parts of it can also be implemented as aCEP query. The query in Listing 4.8 takes all sensor data and aggregates it within a five minute time window. The average of all sensor values within a valid range is then forwarded to the next integration step. Even if the processing of the sensor data is implemented with several CEP queries, agents, or a mixture of both, the complete path of the messages from the sensor via the interpretation to the actuator is visible via the visualisations of the middleware due to the seamless integration of the CEP engine. Any interruptions due to programming, configuration, or hardware errors would be directly visualised for the developers.

```
SELECT avg(sensor.ec02)
INTO filtered_sensor_data
FROM sensor_data.win.time(300000) as sensor
WHERE sensor.ec02 > 0 &&
      sensor.ec02 < 100
```

Listing 4.8: CEP query to aggregate all sensor data in a 300,000 millisecond (5 minute) time window by an average function. Values that are out of range are filtered out prior to the aggregation step

In the second step, during the implementation of the sensors, it is helpful to have a real-time display of the sensor data. This can be implemented with the query in Listing 4.9. It converts all sensor readings from the sensors into drawing instructions for a 2D display. The display then shows a real-time representation of the sensor values, which can be used to interpret the data and test the implementation. It is possible to filter the data with further queries to ensure that it is within the expected range or sent at the expected interval.

```
SELECT
   sensor_values as name,
   sensor.name as x,
   sensor.ec02 as y
INTO draw_2d
FROM sensor_data
```

Listing 4.9: CEP query to visualise sensor data

The system was expanded to include additional sensor nodes and sensor types in several interactions and research projects with students. In addition, various post-processing methods were implemented to detect and correct errors in sensor data.

4.5 Conclusion

This chapter gave an overview of the system design and implementation and then presented the three layers of the architecture.

The first layer, the messaging layer, implements publish/subscribe messaging and the basic functions of the agent-based system. When designing the architecture, particular attention was paid to the supporting program comprehension and debugging tasks, which is a special requirement for a software laboratory that is often neglected for other areas of application in the literature analysed (see Section 2.3.4). In addition, these requirements cannot be implemented, or can only be implemented to a limited extent, with the message brokers often used in practice, such as MQTT.

The second layer, the CEP layer, is mainly responsible for processing contextual information in the form of messages from the agents. It works entirely on the basis of the messaging layer and the publish/subscribe communication provided there, which consists of agents themselves.

The advantage of the type of integration presented here is that, in program comprehension and debugging tasks, developers can view all components of the system as agents. If a message from a sensor is processed through several steps in the system to trigger an actuator, all steps can be tracked regardless of whether the processing was implemented entirely by agents, by CEP queries, or a mixture of both. Thus, the advantage of the seamless integration used here is that the event-based and agent-based paradigms complement each other.

Messages and also system information about agents can be processed by CEP queries, which can also be used for development tasks. These functions are provided in the third layer, the user interaction layer, which is mainly responsible for the developer support requirements. The layer consists of several user interfaces that build on the interfaces of the messaging and CEP layers. The user interface supports developers in the development, testing, and deployment of agents and allows them to interactively explore the system at runtime. The latter supports debugging and program comprehension tasks in particular.

One of the most important features of the user interaction layer is the agent communication graph, which dynamically displays the current system status and can be filtered by the user as desired. For example, an agent and all agents that communicate directly with it can be displayed to find out which dependencies exist for this agent or to locate the source of faulty messages.

To illustrate that the functions of the user interaction layer presented here fit the use cases in a smart system, first interviews with experts (see Chapter 6) and then a scenario-based architecture evaluation (see Chapter 8) are conducted.

Chapter 5

Evaluation Methodologies

This chapter gives an overview of the research design choices in this thesis. Based on the research questions, the studies conducted in the following chapters to evaluate the architecture are presented. Then, in preparation for the evaluation, the literature on expert interviews as well as scenario-based architecture evaluation is analysed.

5.1 Methodological Outline

Based on the objectives and the research questions (see Section 1.3), the following section provides an overview of the further procedure in the analysis of the requirements for software architectures in smart systems and the evaluation of the architecture presented in this thesis.

The methodology of this work is predominantly based on Saunders [2019]. The *research onion* by Saunders [2019] is an overview of the different research design choices. It is organised in layers and goes from research philosophies on the outside in several steps to the concrete methods for data collection and analysis in the middle. This model is used here to present the research design in accordance with literature. Figure 5.1 depicts a research onion with the decisions made for this thesis.

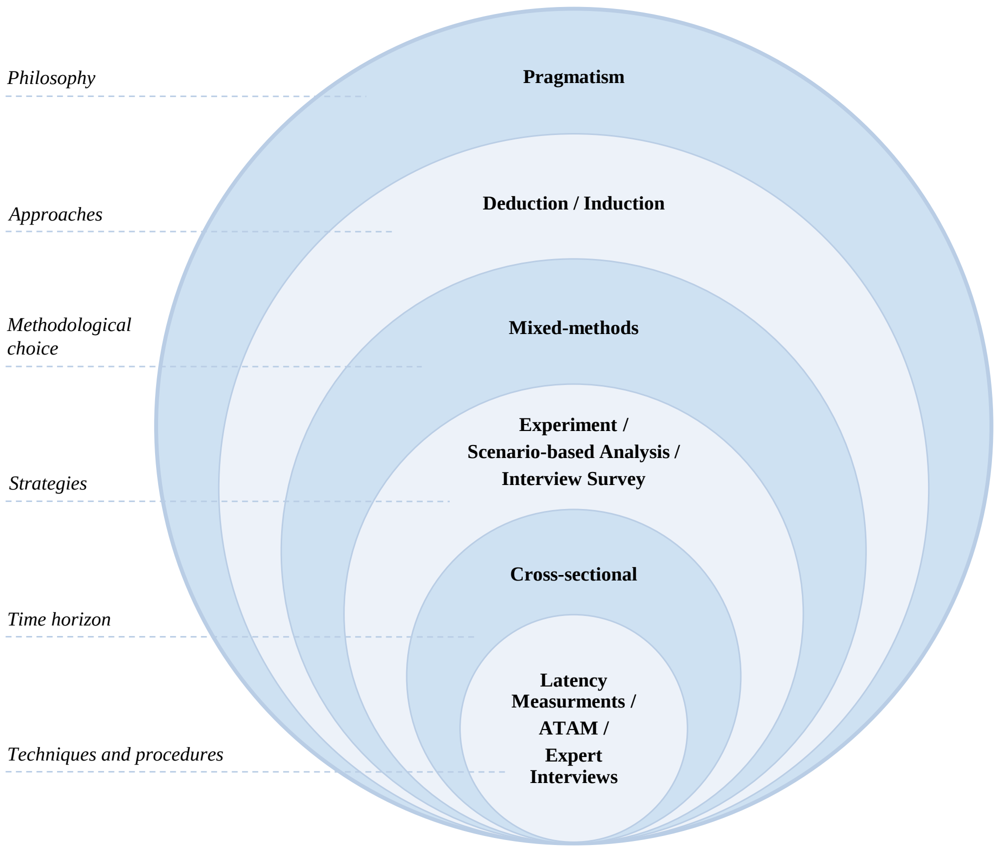

Philosophy
Pragmatism
Approaches
Deduction / Induction
Methodological choice
Mixed-methods
Strategies
Experiment / Scenario-based Analysis / Interview Survey
Time horizon
Cross-sectional
Techniques and procedures
Latency Measurments / ATAM / Expert Interviews

FIGURE 5.1: Research onion based on Saunders [2019] with the research design choices made in this thesis

Saunders [2019] describes five major research philosophies. They lay the groundwork for the decisions and assumptions made during a research study. Since the research questions Q1 to Q4 require very different approaches and methods, the philosophy that best fits this thesis is pragmatism. Pragmatism encourages choosing methods that best fit each research question, regardless of whether they are quantitative or qualitative. This is appropriate for this thesis because, for example, question Q2 is about message latency and scalability, which is a good case for quantitative methods, and Q4 considers whether CEP can support developers can therefore be analysed effectively with qualitative methods.

This thesis follows both an inductive and a deductive approach. The scalability of the middleware is inferred from an experiment confirming that adding more nodes increases the performance of the system. In contrast, the expert interviews

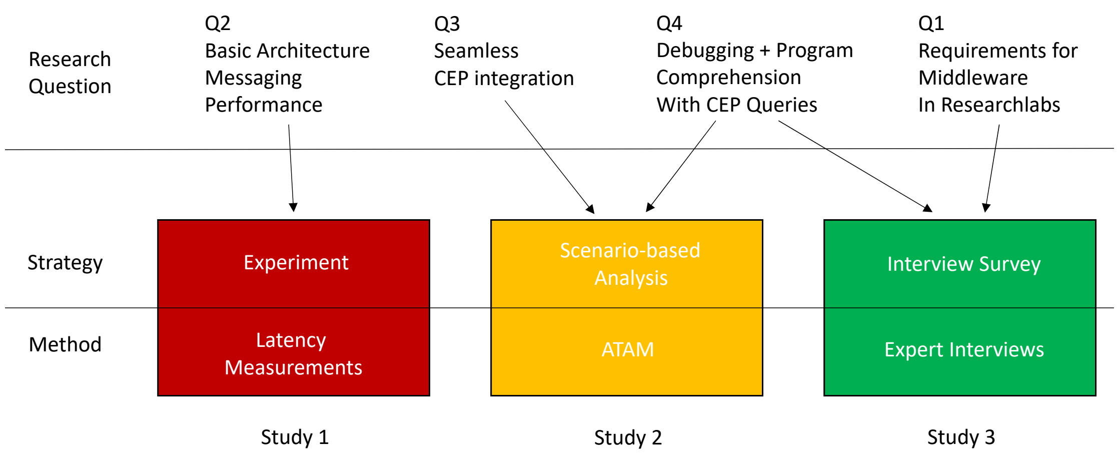

Research Question
Q2 Basic Architecture Messaging Performance
Q3 Seamless CEP integration
Q4 Debugging + Program Comprehension With CEP Queries
Q1 Requirements for Middleware In Researchlabs
Strategy
Method
Experiment
Latency Measurements
Scenario-based Analysis
ATAM
Interview Survey
Expert Interviews
Study 1
Study 2
Study 3

FIGURE 5.2: Methodological outline

are used to inductively infer the requirements of software architecture for smart system laboratories, based on the individual data collected in the interviews.

Since both quantitative and qualitative methods are used to answer the research questions, in line with the research philosophy of pragmatism, the methodology of this thesis follows a mixed methods approach. More precisely, since not all steps in this thesis use both quantitative and qualitative methods, this is partially integrated mixed method research. The different methods are each applied separately, and the results are combined in relation to the research questions.

Several strategies were used for the studies carried out here. Figure 5.2 displays the relationship between the research questions and the research strategies and methods chosen. The requirements for middleware in smart system labs (Q1) were determined with the help of an interview survey [Bhattacherjee, 2012] with expert interviews (see Seaction 5.2). In addition, the approach of using CEP queries to support program comprehension and debugging tasks (Q4) is evaluated.

Bhattacherjee [2012] distinguishes between two different survey types. Both follow a structured procedure and serve to collect data from individual persons to be able to evaluate them afterwards. In the case of questionnaire surveys, a standardised questionnaire is usually used to enable a quantitative evaluation even of large numbers of participants. Interview surveys, on the other hand, are usually analysed using qualitative methods and allow the interviewee to comment more freely and express their own opinions. Here, interview surveys were conducted to support an exploratory approach in which open questions were asked to determine the requirements for middleware in smart systems labs and to evaluate the developed approach to debugging with CEP queries.

The performance characteristics of the software architecture (Q2) presented here were evaluated with the help of multiple experiments. An experiment examines the dependence of an independent variable on another dependent variable [Saunders, 2019]. The independent variable is changed during the experiment, and the changes in the dependent variable are measured. In this thesis, the latency of messages depends on the number of nodes and the number of components and messages in the system is tested in two experiments.

Furthermore, the basic architecture and seamless integration of CEP were analysed by a scenario-based evaluation method [Babar and Gorton, 2004] (see Section 5.4). The presented architecture is analysed with the help of several scenarios to determine whether the previously determined requirements are fulfilled.

All studies in this paper are cross-sectional studies that examine contexts or events at a specific point in time. The scenario-based analysis of the architecture is based on concrete research projects that have a limited project duration, which is analysed here. A long-term study is not within the scope of this thesis. The latency and scalability measurements are also based on a given scenario to make the results as reproducible as possible. No long-term study is necessary here because message latency and scalability behave independently of the progress of a project or similar changes.

The individual data collection and analysis methods listed here are outlined in detail in the following sections. This includes the qualitative studies consisting of expert interviews [Meuser and Nagel, 2009] (see Section 5.2) and the evaluations of the scenario-based evaluation following the Architecture Trade-off Analysis Method (ATAM) [Kazman et al., 1998] (see Section 5.4). The quantitative study of the latency and scalability of the system is conducted in Chapter 7.

5.2 Expert Interviews

To analyse the requirements of software architectures in smart system research laboratories and to evaluate the transferability of the architecture presented, expert interviews were conducted.

Meuser and Nagel [1991] started the initial systematic debate of expert interviews in Germany, but the debate was only intensified 10 years later [Bogner et al., 2009]. Expert interviews should be open and based on an interview guideline, which structures the interviews [Meuser and Nagel, 2009]. The guideline is not a script that defines every question of the interview in a fixed order but a list of topics that are covered in each interview. According to Bogner et al. [2009] expert interviews are an efficient way to obtain good results quickly.

Whether someone is considered an expert for a research project depends on the exact research question. The expert status is in some way assigned by the researcher Meuser and Nagel [1991], depending on a specific research project. There are different definitions of experts. According to Meuser and Nagel [1991], someone can be considered an expert if they are is part of a field of action that represents the object of research. Experts can be people, who have responsibility for the design or control of a solution or have privileged access to information about people or decision-making processes [Meuser and Nagel, 1991]. This definition is used in this thesis. Often, experts are not to be found at the highest hierarchical level but two to three levels below, because this is where decisions are prepared and enforced [Meuser and Nagel, 1991]. This means that people who work with loosely coupled software, staff and students in research laboratories, are particularly suitable for this study.

The quality of the interviewees in expert interviews strongly influences the success of the study [Gläser and Laudel, 2009]. The experts are expected to understand what information the interviewer is asking for and to give as detailed and complete statements as possible. Researchers are considered experts in their field of research [Gläser and Laudel, 2009]. However, there are differences in quality, and it can be very difficult to judge whether a researcher is a good or a bad expert for an interview. Most studies ignore this problem and only some define the quality of researchers as experts using research performance indicators [Gläser and Laudel, 2009].

Since this study is primarily concerned with the practical experience of working with loosely coupled systems in a research context rather than research experience in the field, the work experience and not the academic performance of the interviewees is used here as an indicator.

Expert interviews were used in this thesis to get insights into two research questions. On the one hand for an exploratory analysis of the requirements for software architectures in the research context when working in smart systems and laboratories with similar requirements. This were achieved with the open questions and detailed structured analysis of the interviews. Using questionnaires and a quantitative analysis would make it more difficult to understand the connections and reasons behind statements during the analysis. On the other hand, the expert interviews were used to analyse use cases in other laboratory environments and limitations of the software architecture. The aim is to draw conclusions about the transferability of the architecture to similar environments. It is assumed that open questions, in contrast to a fixed questionnaire, can yield more detailed results, for example, by asking further questions during the interview to analyse topics in detail that were not expected in advance.

Qualitative research, in contrast to quantitative research, is concerned with meaning instead of general statements [Mason, 2010]. Therefore, the number of participants in a qualitative study is usually smaller than in a quantitative study, because the results are often not dependent on the frequency of individual data points [Mason, 2010]. In addition, the analysis of large sample sizes in the qualitative evaluation of interviews is very time-consuming or even impractical because of the extensive evaluation procedures. For example, to show that software is appropriate for a user group, one could survey a large number of users with a

Publication	Number of Participants	Field of Expertise
Lehner et al. [2010]	30 / 3 groups of 10	informatics / didactics of informatics / informatics teachers
Haselbock et al. [2018]	10	microservices
Koch [2019]	7	digital ecosystem-related industry
Hannibal [2021]	8	robotics
Beck et al. [2022]	5	business software architecture

TABLE 5.1: Comparison of sample sizes of computer science publications with expert interviews

questionnaire and statistically analyse it. Since the presented software architecture for smart systems and the CEP engine are special domain-specific software that is only to be used in very specific environments in the research context, there are probably not enough candidates who can be surveyed. In addition, a training period and prior knowledge are necessary to use the software, which makes it even more difficult to find participants.

The question of exactly how many interviews should be conducted is not easy to answer and depends on the circumstances. According to Mason [2010], the guiding principle for the selection of a sample size should be saturation, which is reached when no new findings are added with further data. In a study by Mason [2010], 560 PhD studies with interviews where evaluated with a median of 28 and a mean of 31 participants. The most common sample sizes in the analysed studies were 20 and 30. However, all types of qualitative interviews were considered in this study. If all participants must be experts regarding the research questions, it can be more difficult to reach these numbers. For expert interviews, the potential candidates may be very limited. Indeed, according to Baker and Edwards [2012] it can be very difficult to find more than 10 participants. For the studies in this thesis, it is difficult to find such a large number of participants because a considerable amount of prior knowledge, such as experience with loosely coupled systems and smart systems, is required.

Expert interviews have not only been used in the social sciences but also for various publications in computer science and other fields. Table 5.1 shows examples of computer science publications that conducted expert interviews and compares their sample sizes. All had 10 or fewer participants per field of expertise.

5.3 Qualitative Content Analysis

This section presents how the recordings of the expert interviews are analysed. The aim is to carry out the evaluation in a structured, reproducible way based on the literature and focused on the research questions.

According to Meuser and Nagel [2009] the evaluation of expert interviews generally follows six steps.

1. **Transcription** - All relevant sections of the recorded interviews are transcribed in the first step. The transcription of pauses (prosodic and paralinguistic) is optional, because they are not considered in the evaluation Meuser and Nagel [1991]. It is important that the transcription is complete in terms of content and reflects the content of the interview.

2. **Paraphrase** - The sequencing of the text based on thematically matching sections comprises the second step. Individual parts of the interview can be paraphrased to reduce the size of the material. Which parts should be transcribed and which paraphrased depends on the research questions.

3. **Coding** - In the coding step, the volume of the material is reduced by assigning codes to individual paraphrases or text sections. In this step, the terminology of the interviewee is used whenever possible. Hence, text sections can be understood independently of the sequence with one or more codes.

4. **Thematic comparison** - From this step on, different interviews are compared with each other instead of looking at individual interviews. Comparable text sections from the interviews are assigned the same codes, and redundancies are reduced.

5. **Sociological conceptualisation** - From this point on, the exact text or terminology of the interviewee is no longer considered, but rather terms and categories are formed. The terms and codes taken from the text are translated to make them comparable with other results and studies.

6. **Theoretical generalisation** - In the last step, theories are derived from the created categories by considering them in terms of their relations to each other

In the first step, *Transcription*, all interviews are transcribed for further analysis into text, with time-stamps for each question and answer. Afterwards, all texts are anonymised, with all names of persons, places, and institutions are replaced by place holders. In addition, all information that would allow for a clear identification of the interviewees is removed.

To analyse the collected data and evaluate it in relation to the research questions, a Qualitative Content Analysis (QCA) can be used [Schreier, 2012]. This method is used in this thesis to cover step 2 to 6 of the evaluation according to Meuser and Nagel [1991]. It is widely agreed that a QCA is structured Schreier [2012] in three aspects: Firstly all the material is sorted by the coding frame. Secondly, the QCA includes the same sequence of steps every time, independent of the material or the research questions. Finally, the goal of the coding process is to be consistent. To make QCA independent of personal understanding and interpretation of the data the consistency of QCA is checked throughout the whole process via multiple coding steps by different people or the same person at different times. A QCA can help describe the material with regard to certain previously defined criteria. It can not be used to describe the full meaning of the data in every aspect, wich is what distinguishes it from other qualitative methods for data analysis [Schreier, 2012].

The method thus helps to extract information from a large amount of data in a way that is targeted to the research questions instead of performing an analysis of all contained information.

According to Schreier [2012], a QCA consists of the following eight steps:

1. Deciding on the research questions
2. Selecting your material
3. Building a coding frame
4. Dividing your material into units of coding
5. Trying out your coding frame
6. Evaluating and modifying your coding frame
7. Main analysis
8. Interpreting and presenting your findings

Figure 5.3 shows the evaluation process based on a schematic from Kuckartz [2018] and the QCA steps from Schreier [2012].

5.3.1 Coding

In qualitative research, a large amount of data can accumulate quickly that cannot easily be processed automatically [Schreier, 2012]. For example, in the case of interviews conducted for this thesis with a length of 30 to 45 minutes and 5,000 to 6,000 words per interview transcript, there are approximately 4.5 hours of video and 38,500 words of transcript to evaluate. Even with a very specific research question, it would therefore be very difficult to exclude the possibility of overlooking something without a structured approach. In addition, the goal of the evaluation is to find new and possibly unforeseen results. To make this possible

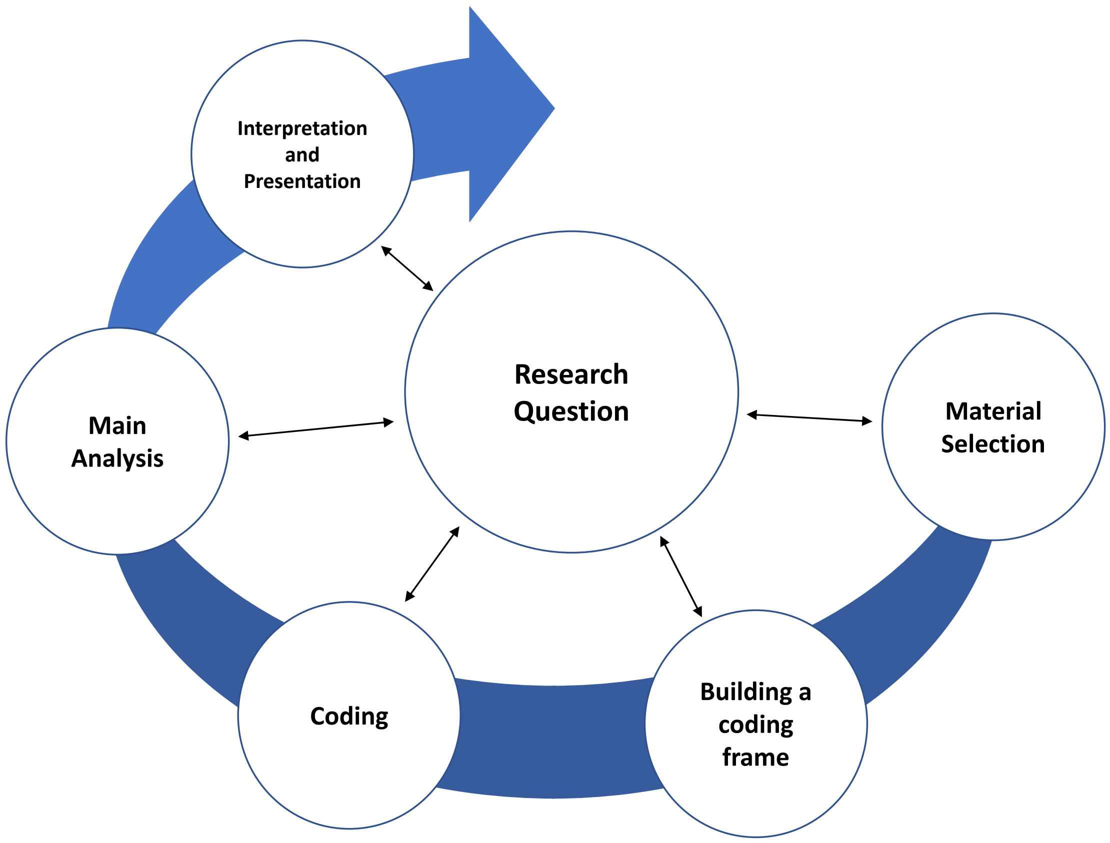


FIGURE 5.3: Main steps of the QCA procedure based on Kuckartz [2018] and Schreier [2012]

and ensure that the evaluation of the data is done in a structured way, a coding frame is used.

All interviews were be coded into coding units using the coding frame. The coding frame is the core of the evaluation. It is used intensively to classify, sort, and reduce the collected data. "[...] coding units are units that are distinguished for separate description, transcription, recording, or coding" [Krippendorff, 2004, page 99]. For example, a coding unit would be a statement during one of the interviews about a category that fits within the coding frame.

The coding frame consists of a list of main categories, subcategories, and their definitions. It is used to annotate the gathered data. The main categories of the coding frame are the topics that are the focus of the analysis. The identification of the main categories is carried out based on the research questions [Kuckartz, 2018]. Subcategories are used to further organise the main categories into aspects that are relevant to the study. For each main category, subcategories are formed to gradually narrow down all the units of coding assigned to the main category [Schreier, 2012].

The coding frame is developed according to the following requirements by Schreier [2012]:

1. **Validity** - The coding frame should be comprehensible and appropriate for the defined research questions.
2. **Reliability** - The coding frame should lead to the same results when used several times, even by other people.
3. **Unidimensionality** - Each category of the coding frame should represent only one aspect of the material. If multiple dimensions are grouped together into one category, they should be split up. or the coding frame has to be reorganised accordingly. For example problems with distributed software and solution strategies related to this study should not be part of the same category to allow an independent analysis.
4. **Mutual exclusiveness** - Subcategories in the coding frame should be mutually exclusive. This means that each unit of coding should fit at most one of the subcategories of the main category.
5. **Exhaustiveness** - All relevant units of coding should match at least one subcategory in the coding frame. If a relevant text section cannot be classified, a new category should be created for it be able to consider it in the evaluation.
6. **Saturation** - Each subcategory should be used at least once. Unused subcategories should be removed from the coding frame.

A good coding frame should implement the majority of these requirements, and the most important of which are validity and reliability [Schreier, 2012], which are evaluated with a coding test.

The process of coding with the coding frame is illustrated in Figure 5.4 based on Kuckartz [2018]. Firstly, all the material is coded with the main categories. Then, the subcategories are inductively generated from the material during a second coding step.

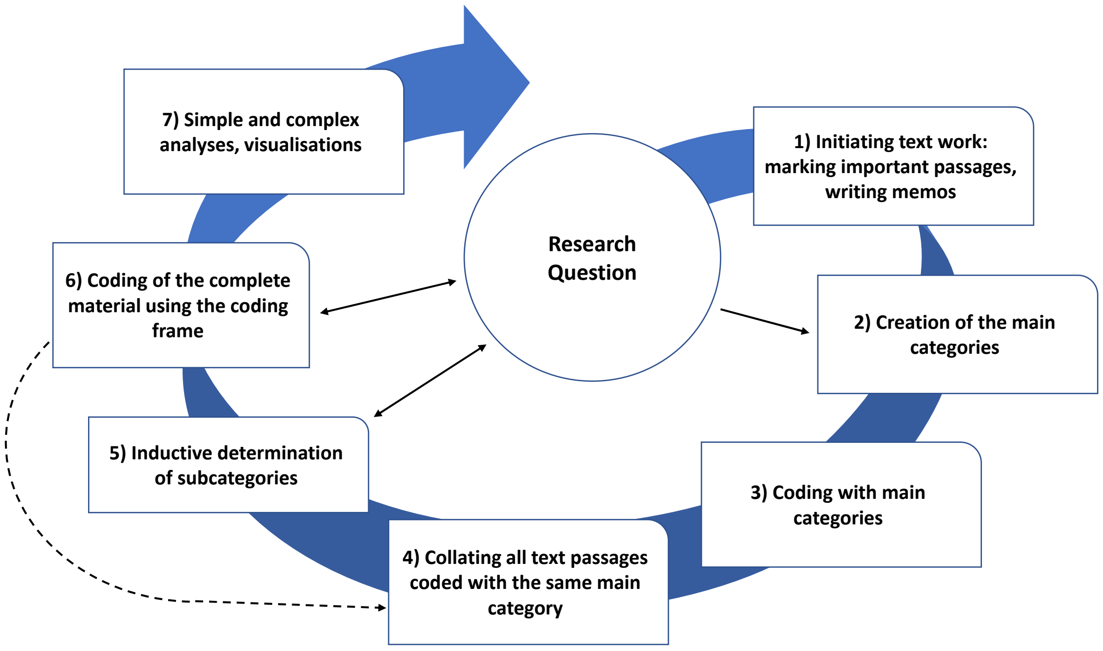

Research Question
1) Initiating text work: marking important passages, writing memos
2) Creation of the main categories
3) Coding with main categories
4) Collating all text passages coded with the same main category
5) Inductive determination of subcategories
6) Coding of the complete material using the coding frame
7) Simple and complex analyses, visualisations

FIGURE 5.4: QCA coding procedure based on Kuckartz [2018]

The analysis of the interviews with the help of a QCA and the presentation of the results is available in Chapter 6

5.4 Scenario-based Analysis Method

This section describes the procedure for evaluating the architecture in this thesis will be selected and described. Santos et al. [2022] surveyed research papers with an evaluation of SoS software architectures from 2006 to 2021, focusing on the last 10 years. Their study demonstrates that different evaluation approaches have been used to determine the quality of software architectures in this area, including experience-based, simulation-based and scenario-based evaluation, as well as mathematical modelling.

In an experience-based evaluation, the architecture is evaluated based on the knowledge of the software architecture according to the requirements [Patidar and Suman, 2015]. Compared to the other approaches, this has the disadvantage that it is less based on a well-defined and structured process and instead follows more subjective factors, such as the intuition and experience of the architect [Bosch and Molin, 1999]. However, since the opinion of experienced software architects has a value that should not be underestimated, this approach can serve as a basis or starting point for other evaluation methods [Bosch and Molin, 1999].

Alternatively, architectures can be evaluated by simulation or mathematical models. This approach is also suitable for agent-based architectures, as is often the case with smart systems. Often times, certain attributes of the architecture, such as performance or correctness, are evaluated by automatic simulation tools or mathematical proofs [Balsamo and Marzolla, 2003, Reussner et al., 2003].

Conversely, in scenario-based approaches the quality characteristics of an architecture are systematically evaluated based on scenarios.

Although simulation-based approaches for the evaluation of SoS have dominated in recent years, scenario-based approaches are better suited for the evaluation of trade-offs between quality attributes of an architecture [Santos et al., 2022]. Scenario-based approaches, in contrast to simulation-based and mathematical modelling, can investigate several quality attributes and their interactions simultaneously [Santos et al., 2022]. The architecture evaluated here was designed to facilitate development processes for complex systems. In addition to the performance of the system, usability aspects and functionalities related to the work of developers are particularly important. Several different properties and functions of the architecture must be included and set in relation to each other. An example of this is performance and the system's ability to provide debugging information. For this reason, a scenario-based approach was chosen for the following evaluation.

Different methods can be used for a scenario-based analysis, which is usually based on the following sequence [Babar and Gorton, 2004]: Firstly, the process is planned and the evaluation is prepared, secondly, the main approaches of the software architecture are presented, and thirdly, scenarios are identified that depend on desired quality attributes, and finally, the software architecture is evaluated based on these scenarios, and the results are interpreted and presented.

The central aspects of a scenario-based evaluation are the *scenarios*. A scenario consists of a brief description of the interaction of one or more stakeholders with the system [Clements et al., 2009]. Stakeholders can have different views of the system depending on their roles. From a user's perspective, a scenario could be a description of a use case; from a maintainer's perspective, it could be a description of a change to the system; and from a developer's perspective, it could be a design or enhancement of the system.

The description of a scenario consists of three parts [Clements et al., 2009]. The *stimulus* is the description of the action, or the trigger that causes a change in the system. The *environment* is a description of all relevant internal and external circumstances at the time of the change. These can be system states, system failures, errors, and other effects. This part can be omitted if there are no special circumstances. Finally, the *response* describes the reaction of the system to the stimulus. This can be certain expectations, such as *providing a response within 100 ms* or *outputting an error description*.

The scenarios can result from the requirements for an architecture but are described in more detail on the basis of a use case. For example, a common requirement is the fault-tolerance of the system. A corresponding scenario would additionally define which components are fault-tolerant, how many components may fail at the same time under which circumstances, and what the expected behaviour is when this occurs.

There are several different methods to perform a scenario-based architecture analysis [Babar and Gorton, 2004, Dobrica and Niemela, 2002].

In this study, ATAM was used because it is a mature and proven method for evaluating software architectures. Babar and Gorton [2004] presented a framework for the comparison of software architecture evaluation methods that it uses the maturity stage as one criterion for comparison. According to this, ATAM is one of the most mature methods and is in the refinement stage, which means that the method has been validated by results in credible research literature and is being continuously developed. Another indication of the maturity of ATAM is that ATAM and its predecessor, the Software Architecture Analysis Method (SAAM), unlike many other methods, have been used in different industry projects [Shanmugapriya and M. Suresh, 2012, Patidar and Suman, 2015]. These projects have been recorded and evaluated in books as extensive case studies [Clements et al., 2009].

ATAM is the only method that offers comprehensive process support [Babar et al., 2004]. Furthermore, only ATAM provides support for the production of reusable results, such as the identification of risks, scenarios, and quality attributes [Babar and Gorton, 2004]. According to Dobrica and Niemela [2002], the method to be used should be determined by the context of the problem. A method should be applicable early in the development process, support the evaluation of all quality attributes of the architecture to be evaluated, and be easy to apply. ATAM supports all these requirements [Dobrica and Niemela, 2002] and fits all the quality attributes evaluated in Chapter 8.

5.4.1 Architecture Tradeoff Analysis Method

In the following the ATAM will be discussed first and then the exact procedure used here will be described in detail before continuing with the evaluation and the presentation of the results. The procedure to evaluate the architecture, which is used in this thesis is based on the Architecture Trade-off Analysis Method (ATAM) [Kazman et al., 1998] and has been supplemented to take the circumstances of this study into account.

ATAM [Kazman et al., 1998] is a method for architecture evaluation that is based on the Software Architecture Analysis Method (SAAM) and it includes both social and technical aspects. The social aspects regulate the interaction between stakeholders and provide a framework for an exchange to transform the implicit assumptions of individuals into an explicit overall picture. The technical aspects regulate what data is collected and how it is analysed.

Instead of only evaluating whether a quality attribute is covered by an architecture, ATAM addresses multiple quality attributes and aims to determine how they are related to each other [Kazman et al., 1998]. For example, performance can negatively affect modifiability, and availability can affect safety. These dependencies are called tradeoffs and their identification is one of the main objectives of ATAM.

ATAM can be used during different stages of software development for the evaluation of one or the comparison of several software architectures [Santos et al., 2022]. In the early stages, a scenario-based evaluation can help to ensure that the architecture meets the requirements and is able to react to problems early. ATAM can also be used to analyse legacy systems, such as when major changes are planned or the architecture is to be compared with a successor to assess whether a change of architecture makes sense [Clements et al., 2009]. According to [Babar and Gorton, 2004], ATAM is most effective for evaluating the final version of a software architecture, as is the case in this study.

ATAM involves all major stakeholders, or all persons with a vested interest in the system [Babar and Gorton, 2004]. This includes software architects, designers, and end users. The evaluation procedure consists of four major phases [Kazman et al., 1998] and eight steps [Clements et al., 2009], that can involve different stakeholders. They conduct the following evaluation steps during one or more workshops.

- **Presentation** - The working group receives all necessary information about the further procedure and its contents through a series of presentations.

  1. **Present the ATAM** - First of all, the ATAM is presented to all stakeholders involved in the evaluation.

2. **Present the business drivers** - Subsequently, the business drivers of the development of a system and the evaluation are presented. This includes the presentation of the purpose of the system, constraints, business goals, and an introduction of the main stakeholders.
3. **Present the architecture** - Finally, the architecture to be evaluated is presented. Among other things, the structure of the architecture, the technical constraints, and relevant interfaces to other systems are discussed. Here, it is important that the architecture is presented in appropriate detail for evaluation because the results depend directly on the information given.

- **Investigation and Analysis** - The quality characteristics and requirements for the architecture are mapped to the architectural approaches.

4. **Identify the architectural approaches** - The first step in the analysis is to identify the architectural approaches or architectural styles. These describe, for example, important structures in the system, and how the system can grow, react to changes, or interact with other systems. This includes a description of the components involved, their structure, and their interaction. The list of approaches is created by the system architect and is used to support the following evaluation. The list of approaches is completed with all participants during the following steps.
5. **Generate the quality attribute utility tree** - The next step is to identify and order the quality goals of the architecture by importance. Firstly, the major goals of the architecture, such as performance, modifiability, or availability, are identified. These are then subdivided and described with scenarios. For example, performance could be subdivided into message latency and throughput. A concrete scenario for message latency could then be an expectation of certain maximum or average latencies for a component. The individual scenarios are then

pre-prioritised, according to their importance for the success of the system.

6. **Analyse the architectural approaches** - In this step, the quality attribute tree is used to examine the suitability of the architectural approaches for the prioritised scenarios. The goal is to determine whether the architecture to be examined fulfils the established requirements. The result is the assignment of the architectural approaches to the scenarios, analysis questions, and answers. In addition, risks, sensitivity points, and tradeoffs are identified.

- **Testing** - The preliminary results are checked against the requirements of all relevant stakeholders.

7. **Brainstorm and prioritise scenarios** - At the beginning of the testing phase, additional scenarios are first searched for, and then all existing scenarios are weighted. New scenarios are compared with the existing ones and added or merged if necessary. Only the architects and project leaders are involved in setting up the utility tree. Here, all stakeholders are consulted to expand the scenarios. With this approach, it is possible to determine if there are scenarios that were neglected when the architecture was set up.
8. **Analyse the architectural approaches** - Analysis of the architectural approaches for any newly added high-priority scenarios during the testing phase is carried out in the same way as in step 6.

- **Reporting** - Presentation of the results of the evaluation.

9. **Present the results** - Finally, the results are collected and presented. This includes the architectural approaches, the prioritised scenarios, the analysis questions and answers, as well as the utility tree.

5.4.2 Adjustments to the procedure

The procedure description for ATAM in Clements et al. [2009] is mainly laid out for an evaluation of an architecture in a business context. The motivation is to evaluate an architecture as early as possible in the decision-making process in order to save costs. The evaluation can be done before the implementation of the architecture as well as on the basis of existing systems, for example to compare legacy systems with a newer approach to decide whether a change is worthwhile. Scenario-based evaluation methods for software architectures are also frequently used in research. In particular, a survey by Santos et al. [2022] indicates that this also applies to SoS.

Since the architecture examined here is not in a business context, but in a research context, ATAM is adapted. The adjustments mainly concern the time frame of the implementation and the composition of the working group. All deviations from the method described in the literature are explained and justified in detail below.

According to Clements et al. [2009], an evaluation following the ATAM usually takes three days, plus preparation and follow-up. In contrast, this study spans a longer period of time and several phases, bacause the architecture presented here has been developed in multiple steps and has evolved over time. The basic architecture was developed first, and the CEP integration was added later, after intensive experience had been gained with the messaging layer. This has the advantage that the results of the evaluation incorporate the experiences and feedback of the entire development period and are not just a snapshot of the final results. In addition, the longer period gives stakeholders the opportunity to work intensively with the architecture and to implement their own projects based on this architecture. These advantages are not possible to this extent with the original method lasting a few days. It is assumed that the evaluation benefits from the longer investigation period, and ATAM specifies a shorter period to keep the necessary effort for investigations within the business context feasible.

The situation is similar with the participants and the procedure of the study. According to Clements et al. [2009], a four person evaluation team, the architect, the client, and the stakeholders are normally involved in an evaluation according to ATAM. In addition, the evaluation is carried out together with all stakeholders over a few days.

In this study there are different groups of participants with whom the architecture is discussed. The organisation and compilation of the results lie with the software architect and the author of this thesis. The evaluation includes experiences and results from several student courses at the university and discussions with users of the architecture in the two investigated laboratories. These include students writing their theses and their supervisors, lab managers, and lab staff.

Compared to an evaluation according to ATAM, more individuals are likely involved, but the examination was not carried out in the presence of all participants at once. These adjustments are justified by the longer study period and the nature of research projects in the laboratories mentioned. Bachelor's and master's theses, research projects, and university courses with students usually run for six month to one year. It would therefore only be possible to carry out this study at a fixed point in time if the number of participants were significantly reduced. Table 5.2 gives an overview of the professional roles of the stakeholders participating in the individual steps of the ATAM architecture evaluation with in this thesis.

Step	Number	Responsible Roles
Presentation	1-3	System Architect
Identification Architectural Approaches	4	System Architect
Quality Attribute Utility Tree	5	System Architect Laboratory Manager Student Supervisor
Analysis Architecture Approaches	6, 8	System Architect Laboratory Manager Laboratory Staff Student Supervisor Student
Brainstorm and Prioritize Scenarios	7	System Architect Laboratory Manager Laboratory Staff Student Supervisor Student
Reporting	9	System Architect

TABLE 5.2: Overview of the professional roles of the stakeholders involved in the analysis. It is important to note that in steps 7 and 8, the results of steps 4 and 5 are evaluated and completed if necessary.

Chapter 6

Expert Interviews

To further analyse the requirements for software architectures in smart systems laboratories and to start the evaluation of the program comprehension and debugging improvements, interviews were conducted with experts who have considerable experience in the development of software components for smart systems or in the maintenance of distributed, loosely coupled software systems in a laboratory context. The interviews pursue three objectives: Firstly, to analyse research environments and the team compositions the interviewees work in, secondly, to identify special requirements for software architectures in these environments, and thirdly, to analyse whether and how the software architecture presented here can be used in other smart system laboratory environments.

The use of expert interviews as opposed to other methods has two advantages. Firstly, it is possible to address all the above-mentioned objectives together through the interviews. For example, it would have been possible to perform an evaluation of the improvement of debugging by the architecture through a study with students measuring how long certain tasks take or how much code had to be adapted by them for certain tasks. However, this would not have provided any information on how the architecture could be applied to improve debugging at all. With the expert interviews, it is not only possible to ask about these aspects at the same time, but the results are also linked together during the evaluation. Secondly, this part of the evaluation had to be conducted under the contact restrictions of the

COVID-19 pandemic in 2020 and 2021. Expert interviews allow participants to be interviewed online without risk. Other methods would have required access to laboratory environment and meetings with participants in person. In addition, the necessary number of participants was reduced by using expert interviews. Normal interviews or a survey questionnaire would have required more participants, which did not exist due to the limitations mentioned above.

The following section details, the research questions for the interviews are specified and an interview guideline. Then, the analysis of the interviews via a Qualitative Content Analysis (QCA) is presented, and the evaluation is carried out, including the presentation of the coding system. Finally, the results of the QCA are illustrated.

6.1 Interview Research Questions

In the first step of preparation for the expert interviews, the research questions to be answered by the interviews have to be defined. These questions depend on the research questions of the thesis, which are already defined in the context of this work and have been shown in the introduction (s. Section 1.3).

Research question *Q1* was only partially answered by the literature review, which concluded with a listing of known requirements for smart system middleware, based on published information about smart systems. During the analysis of the requirements in Chapter 3, it was assumed that these requirements could be adopted for research environments, but should be extended by additional requirements. The additional requirements are mainly due to the need for increased flexibility in a research environment. The interviews aim to confirm this assumption and to concretise the requirements. Therefore, *QI1* addresses the software architectures in research laboratories. In addition to this QI2 investigates program comprehension and debugging tasks in the interviewees' research labs to further substantiate the analysed requirements.

- **QI1**: Which software architectures are used in research laboratories for loosely coupled systems, and which requirements are to be met by them?
- **QI2**: What do program comprehension and debugging tasks look like in research laboratories with loosely coupled software architectures?

Research question *Q2* is addressed in Chapter 7 with latency and scalability evaluations of the developed middleware and of the integrated CEP engine. Parts of research question *Q3* were answered on the one hand by the implementation of the architecture in Section 4.3 and on the other hand will be further evaluated with scenarios in Chapter 8.

Finally, research question *Q4* is covered with *QI3* to determine whether the approaches developed here are compatible with other laboratory environments.

- **QI3**: To what extent can the presented method support program comprehension and debugging tasks with the help of CEP in smart system research laboratories?

6.2 Interview Guideline

In the second step, the materials for the content analysis must be created and selected. In this case, this means planning and conducting the expert interviews.

As described above, an interview guide is used to pre-structure the interviews and ensure that no relevant topics are left out during the interview. In addition, the general procedure of the interviews is further specified, and the participants are selected.

6.2.1 Procedure and General Conditions

The guideline is used to ensure that the process of all interviews is as similar as possible and no topics are left out, but the concrete questions asked depend on the interviewee and the development of the interview itself to allow for the individual answers and opinions of the interviewees. There are therefore no predefined answers or other restrictions. The full interview guideline can be found in the Appendix (see A.1). The interview guideline was tested in advance with one test interview to adjust the timing and reduce the chance of misunderstandings.

The expert interviews are divided into three thematic groups, with QI1 to QI3 as lead questions and a short demonstration of the architecture of this thesis between Part 2 and Part 3:

- **Part 1** - Middleware and Software Development in Smart System Laboratory Environments
- **Part 2** - Program Comprehension and Debugging in Smart System Laboratory Environments
- **Demo** - A short demonstration of the developed middleware and the possibilities of interaction for program comprehension and debugging tasks.
- **Part 3** - Evaluation of Program Comprehension and Debugging with CEP

The interviews were conducted online with video and audio in order to interview people over long distances and eliminate any risk related to the COVID-19 pandemic. All interviews were recorded with video and audio with the consent of the interviewee and took approximately 30-45 minutes as planned. To ensure data privacy and security, the interviews took place on a private, self-hosted meeting server, and all recordings were stored directly on the hard drive of this server.

All interviewees volunteered to participate and received no compensation. Prior to the interview, they were sent a document explaining the procedure and highlighting the following general conditions. The informing of all participants and the collection of consent forms was done by email before arranging the interview appointments. The documents can be found in the Appendix in Section A.3. The information sheet, and the consent form were approved by the UWS ethics committee.

The interviewee could abort the interview at any time, without having to give a reason or worrying about disadvantages, but none of the participants made use of this possibility. Apart from a few minor technical problems and temporary problems with the audio quality, there were no interruptions during the interviews, and all interviews were finished. Thus, all interviews conducted were included in the evaluation.

All interviews began with a welcome, after which participants were asked to introduce themselves. The interview guideline was then followed, starting with questions about general experiences with loosely coupled systems. The second part then asked about procedures for identifying the root cause of software failures. After that, CEP was presented in a short talk as an approach to improving program comprehension and debugging processes. Finally, the participants were asked about the presented approach. At the end of the interview, all participants were given the opportunity to address open issues or to add further topics that they felt had been left out.

6.2.2 Middleware and CEP Interaction Demonstration

During the interviews, the approach presented for program comprehension and debugging with the help of CEP queries was demonstrated. To ensure that the presentation contained the same information for all participants, a presentation consisting of six slides was prepared. Firstly, a short introduction to the functioning of CEP was given if the participant had no related experience prior to the interview. A figure similar to Figure 2.4 and an example CEP query were used for this purpose. Afterwards, two figures depicting the developed interface were shown to demonstrate the user interface. The first figure includes a small graph of agents and groups and their interactions with messages, and the second figure shows a list of messages that were published inside of a group, including a textbox and button to interactively send messages to that group. During the presentation parts of the *Omnidirectional Walking-In-Place Detection* case study (see Section 4.4.4.1) were given as an example.

```
SELECT
   head.position,
   spine_mid.position,
   [...]
INTO draw_3d
FROM skeleton_data
```

Listing 6.1: CEP query to draw selected skeleton joints on an 3D canvas (truncated)

```
SELECT
   1 as step_detected,
   distance_point_floor(
      s.bone1.x, s.bone1.y,
    s.bone1.z,
      f.a, f.b, f.c, f.d
   ) as foot_height
INTO wip_events
FROM skeleton_data as s
JOIN floor_data.keep as f
WHERE foot_height > 3
```

Listing 6.2: CEP query to generate skeleton data to simulate a sensor (truncated)

```
SELECT
   agents.name,
   groups.name,
   groups.last
INTO output
FROM system.agents as agents
JOIN system.groups as groups
ON agents.publish CONTAINS groups.id
WHERE groups.last > 60
```

Listing 6.3: CEP query to select all agents that have not published a message in the last 60 seconds

Several examples of CEP queries were discussed and explained during the presentation. These examples are provided in Listings 6.1, 6.2 and 6.3.

The designed software architecture and the implementation of the seamless integration of CEP were not part of the demonstration to keep the focus on the of interaction with CEP queries and to keep the interview length manageable. To ensure that the interviews work well even with limited connection quality or on devices with a lower resolution, the text portion of the presentation was kept to a minimum. The slides only contain the example queries with a large font size to show the interaction with the CEP engine. The rest of the information was given during the presentation by the interviewer based on a prepared script and was based on experiences of the interviewees that were asked about beforehand. These examples were used to make it as easy as possible to follow the presentation.

6.3 Participants

For this study, seven interviews were conducted in total. All participants are listed with anonymised names in Table 6.1. The suitability of the participants as experts for these interviews was determined based on information provided by the interviewees and referrals. All participants had worked in research laboratories with loosely coupled software architectures for multiple years. Several research laboratories were contacted. and interview participants were sought through referrals. All the candidates who responded were male. At the time of the interviews, two participants were computer science professors, three were postdocs and two were students. Two interviewees reported their experiences in more than one software environment, and two other interviewees worked in the same research laboratory. In total, the interviews contain experiences from eight different environments with loosely coupled software. Six of these are research projects at universities, and two are projects in companies that had to be excluded later in the evaluation, because of an incompatible software architecture. The topics of the projects range from smart home and Ambient Assisted Living (AAL) to middleware and network topology research, self-driving cars, and micro-controller networks. Both for the collection of requirements for these environments and for the evaluation of the architecture presented in this paper, an attempt was made to find people with experience in as diverse environments as possible. This should provide indications about the transferability to other environments.

The language in which the interviews were conducted depended on the preference of the interviewee to encourage as much discussion as possible. However, due to the limitation of the language selection to German and English, some interviews were not performed in the mother tongue of the interviewees. In total, four interviews were conducted in German and three in English.

Interview	Name	Role	Language	Environments
1	Adam	Postdoc	German	E1
2	Bob	Postdoc	English	E2
3	Charlie	Postdoc	English	E3
4	Dave	Student	English	E4
5	Edward	Professor	German	E5, E7
6	Frank	Student	German	E5
7	Gabe	Professor	German	E6, E8

Table 6.1: List of the expert interview participants with anonymised names.

It is often difficult to find more than 10 experts for interviews [Baker and Edwards, 2012] and the number of seven interviews is comparable to other studies in the literature where 5-10 participants per field of expertise were interviewed (see Section 5.2). Since experts were interviewed here, the number of interviews required to derive statements from them is likely smaller since it is assumed that the experts have the information we are looking for and can reproduce it correctly. What matters is not the number of participants but the quality of the participants, the conduct of the interviews, and the systematic evaluation.

Interviews with experts, like other interviews, can fail. One possible failure would be if it emerges during the interview that the interviewee does not feel comfortable with the expert role [Meuser and Nagel, 1991] or if it turns out that they are not suitable as an expert according to the defined criteria after all. This was not the case in the interviews conducted here. All seven interviews were included in the evaluation, which reduces the necessary number of participants compared to studies where it is more difficult to identify experts.

6.4 Interview Evaluation

The next step after conducting the interviews is the evaluation of the results based on the video and audio recordings.

Steps 1 and 2 of the evaluation steps according to Schreier [2012] have already been dealt with. The research questions for the interviews were derived from the research questions of the thesis. The material for this QCA consists of the transcripts of the interview recordings and accompanying text material about the technologies mentioned during the interviews. In addition, this study looks at other documents that were mentioned during the interviews. This includes, among other things, documentation on software products used in laboratory environments. The remaining steps are described in the following sections.

6.4.1 Coding

The next steps require the construction of a coding frame and the coding of the entire material. The main categories and subcategories defined during the coding process are presented below.

6.4.1.1 Main Categories

Based on the interview questions, the following main categories were developed to analyse the transcribed recordings.

- **Part 1 - Middleware and Software Development in Smart System Laboratory Environments**
  - **Information about the interviewee** - Current and past roles of interviewees in the laboratories addressed and their experiences with loosely coupled systems. This information is mainly used to confirm the suitability of the interviewees as experts.

  - **Software laboratories** - Information on the addressed laboratory environments, work processes, and software architecture.
  - **Challenges due to the environment** - Challenges that arise from working in laboratory environments with loosely coupled software.

- **Part 2 - Program Comprehension and Debugging in Smart System Laboratory Environments**
  - **Challenges due to the software architecture** - Problems that arise when using loosely coupled software. These include, for example, problems with the delivery of messages or problems with the identification of individual faulty components.
  - **Solution approaches** - Strategies used to get to know the system or for debugging in case of error, and whether software is used to support this.

- **Part 3 - Evaluation of Program Comprehension and Debugging with CEP**
  - **Experience with CEP** - Interviewee's experience with CEP prior to the interview.
  - **Feedback** - Positive and negative feedback on approaches presented during the demo.
  - **Use cases** - Possible uses of the approaches presented that are addressed by the interviewee.
  - **Limitations** - Reasons an application of the presented approaches might not be possible or why it may only be possible to a limited extent in the addressed software environments.

6.4.1.2 Coding Process

In the coding process, the coding frame is used to divide the material into individual units that are assigned to categories [Schreier, 2012]. The software MAXQDA

2022 (VERBI Software [2019]) was used for the coding process. This software allows the creation of coding frames and applies them to texts without modifying them. In addition, the software can easily compile and evaluate text sections from selected categories.

The coding process is incremental [Schreier, 2012]. In the first step, all relevant text sections are coded into the main categories. All relevant statements are added to a main category one by one. If no suitable category exists, the coding frame must be adjusted accordingly. Text sections irrelevant for the evaluation are not assigned to any of the categories and thus do not hinder the further evaluation steps [Schreier, 2012]. Examples of irrelevant sections include the greeting at the beginning of the interview or questions about the operation of the meeting software used.

Suitable subcategories are subsequently created with the help of the units selected in this way. The text units are then completely or partially assigned to one or more of these subcategories. Afterwards, the coding for the main category is removed. This increases the overview and leads to the fact that no unit is omitted during this process. No information is lost in the process because assignment to the main category continues to exist indirectly through the subcategory.

6.4.1.3 Subcategories

In the following, the subcategories are presented for the three parts of the evaluation, which were defined during the coding process as described earlier. The full coding frame with all subcategories can be found in the Appendix in Section A.2. Only subcategories that are described further and contain several units of coding are further described in the following. Since some subcategories result directly from individual statements, they are omitted here and described later in the evaluation.

Part 1 of the interview was mainly about the experiences of the interviewee, especially about the software environments they have worked in. The information about the interviewee is divided into statements about their role and statements about their working experiences. This information is not part of the following evaluation, but was used to assess the expert role of the participants.

To further evaluate the mentioned software environments, important aspects of the architecture were created as subcategories. These include the message formats and the messaging software used as well as a subcategory added to the coding frame to identify how large and complex the environment is and how changes are made to the architecture.

- **Information about the interviewee**
  - Role
  - Experiences - Statements about the curriculum vitae of the participant
- **Software laboratories**
  - Goals - Aims of the project
  - Software Architecture
    - Messaging Format
    - Messaging API
    - Messaging Software
    - Complexity - Indications of complexity of the software environment, e.g. number of agents
    - Architecture Change Management - How are changes made to the system architecture?
    - Security - Security relevant parts of the architecture

[...]

In Part 2, challenges and solution approaches into working with loosely coupled systems in research projects are further analysed. For this purpose, the categories *Problem Fields* and *Approaches* were further subdivided according to the identified units of coding.

- **Problem Fields**
  - Complexity
  - Distributed Systems
  - Faulty Agents
  - Messaging
  - Documentation
  - Sensors
- **Approaches**
  - Message Validation
  - Debugging Strategies - Information on how debugging is performed
  - Debugging and Testing Software - Used debugging and testing scripts or software

[...]

Part 3 contains the evaluation of the approach presented during the interview. At the beginning, people were asked about their experiences with CEP. The answers were sorted into three categories according to the extent of experience in the coding frame. The *Feedback*, *Potential Uses Cases* and *Limitations* categories were further subdivided to further sort the responses. The subcategories are described in more detail in the further evaluation. They are small and contain only 1-3 units of coding each.

- **Experience with CEP**
  - A lot of experience - CEP used in own projects or in research.
  - Some experience - Heard from CEP but only superficial experience with the use
  - No experience - No knowledge of CEP

[...]

6.4.2 Evaluation of the Coding Frame

At this point, the coding frame is complete, and all relevant parts of the material have been assigned to categories. The coding frame presented in the previous chapter is the final version used for the evaluation. It contains all adjustments and improvements that were made to the coding frame during the evaluation based on the experience gained during the evaluation.

As recommended by Schreier [2012], the coding process was repeated for the complete material. Hence, after a period of several weeks, a second coding was performed. The results of the two runs were then compared. Some inaccuracies in the first version of the coding frame were noticed and are explained in the following paragraphs.

In the first version of the coding frame, there was an overlap of the *Challenges* categories in Part 1 and Part 2. To resolve this, the category names were adjusted and the descriptions were made more explicit. Now only challenges that have to do with the software environment are collected in the first part. Problems with processes are assigned to categories in Part 2.

In addition, an attempt was made to divide the classification of feedback into positive and negative feedback, but this was changed after the second coding run, because the assignment was not clear. For example, the comment that the presented system is not suitable for a purpose can have a negative effect in the first step. However, if the system was designed with this restriction, this feedback can also be perceived as positive. Therefore, the category *Feedback* was divided thematically into *Use Cases*, *Features* and *Limitations*. Further interpretation of the feedback in the individual subcategories took place during the later evaluation.

The coding frame was refined over several steps: during the first coding with the main categories, during the creation of the subcategories, and during the second coding. In addition, the coding frames were discussed with other researchers from the working group to gather feedback. However, no additional coding was performed by other researchers.

6.5 Results

In the following, the collected and processed results from the interviews are presented. The presentation of the results is based on the structure of the interview guide. First, the environments in which the interviewees worked are presented and the software architecture used there is discussed. Afterwards, the challenges that were addressed when working in these environments will be explained. Then the approach of using CEP for debugging is evaluated with the statements made during the interviews and finally a conclusion based on the presented research questions is drawn.

6.5.1 Software Environments

Table 6.2 lists all the environments that were addressed during the interviews. Environments E1 to E6 are research projects, and E7 and E8 are industrial projects. Environments E7 and E8 are not considered in the following because no message-based, loosely coupled software architectures was used in these projects. This only concerns a very small portion of the material because the excluded environments represent the previous work experiences of two interviewees and constitute a small part of the respective interviews.

Notably different environments were mentioned during the interviews. Because the interviews focused on how work was done in these labs rather than what the aim of the research projects was, the types of environments are only briefly described here. E1 and E5 are laboratory flats with the aim of researching systems to support people in their home. E1 started as an AAL laboratory and now focuses on assistance for everyone. E2 and E3 are mobile networking projects. E2 focuses on the networking itself, E3 on the visualisation of network topologies and E4 on self-driving cars. Finally, E6 is a research project about the development of a middleware for distributed agent-based systems.

Environment	ID	Environment type	Research Topic
Smart Home A	E1	Research laboratory	Smart Home, AAL
Mobile Networking	E2	Research project	Mobile Networking
Networking Visualisation	E3	Research project	Mobile Networking Topology
Self-Driving	E4	Research project	Self driving cars
Smart Home B	E5	Research laboratory	Smart Home
Agent Middleware	E6	Research project	Middleware for agent-based system
-	E7	Industry project	Microcontroller network
-	E8	Industry project	Facility control system

Table 6.2: List of environments mentioned during the interviews

One of the main characteristics of a loosely coupled, distributed system is the communication between the components. In all included architectures, this was achieved via one or more messaging systems. Table 6.3 gives an overview of the messaging systems and formats used in the different environments. In the smart home laboratories E1 and E5, mainly MQTT and Home Assistant are used. Both environments are heterogeneous and also use other software and protocols for individual systems, for example KNX in E1. E2 additionally uses Node-RED for graphical programming of processes in the laboratory. In E3 and E4, two other popular messaging systems, RabbitMQ and ZeroMQ, are used. Though very different technologies are used in each environment they are all based on the exchange of messages and, depending on their use, can lead to a loose coupling of the components.

There are also big differences in messaging formats. With JSON in E3, YAML in E4, and XML in E6, three formats frequently used in software development are already represented. In E1 and E5, there are few restrictions on the message format. In E1, a Representational State Transfer (REST) API is used. No other restrictions were mentioned during the interview. In E6 there are explicitly no restrictions, except that all messages must be formatted as strings.

IT security was not the primary focus in an of the environments. The systems are operated internally for research and are only accessible to a few users. Nevertheless, some security mechanisms were reported during the interviews, but these are mainly intended to support multi-user operation. For example, two separate messaging systems are operated in the E5 smart home laboratory. One for the production system and one for developing new components. Some message groups are then shared between the systems to allow, for example, the control of individual actuators in the laboratory from the test system without making the function of the production system completely dependent on the test operation. In E1, the REST interfaces are authenticated to protect them from unauthorised access.

Environment	Messaging Format	Messaging Software
E1	HTTP, KNX	MQTT, HomeAssistant, KNX
E2	Native	Open vSwitch
E3	JSON	RabbitMQ
E4	YAML	ZeroMQ
E5	Text	MQTT, HomeAssistant, Node-RED
E6	XML Schema, Java Code	Custom
E7	KNX	KNX
E8	Unknown	Custom

Table 6.3: Overview of the message formats and messaging software used in the software environment

6.5.2 Identified Challenges

This sub-section details the challenges identified in the interviews when working with loosely coupled systems in general and especially in the research context.

6.5.2.1 System Complexity

The interviewees were asked to estimate the number of components in the system, and further aspects were also explored that were assumed to have an influence on the complexity of the system. These include the characteristics of the distributed system, the handling of messages, and the use of sensor and context information. Furthermore, it was asked whether simultaneous use by several researchers was planned or was possible.

The size of the environments according to the number of components varies greatly. The largest number of components was given as 500 to 1,000 for E3. In E1 and

E5, an unspecified number of student projects run alongside the software platform. Between 5 and 10 of these projects were mentioned during the interviews. For the middleware in E6, a double-digit number of components was estimated. In all environments, it can be assumed that an overview of the system is not trivial, and it is likely that no team member knows all the components in detail. This is assumed because the number of components for some of the projects was very large, while in others it was stated that individual projects were carried out by students under the supervision of only parts of the team.

In some cases, complex technologies were mentioned that are used in the environments. In E1, AI planning tools are used to determine how a certain state in the environment can be produced automatically. In the mobile networking project E2 Bob said they work "[...] with OpenFlow and it is a very complex protocol [...]"' and "sometimes these kind of protocols [...] [are] very complex uh you need the uh for instance just to to be familiar with [...] Open vSwitch I needed uh three or four weeks."For the self driving cars in E4, they use a custom version of the Android operating system and Nvidia drivers with OpenCV in Python. The resulting technical problems took one person about six months to solve.

In some environments, sensor data is evaluated. In particular, in the smart home labs E1 and E5, this sensor data is used to control the behaviour of the system and adapt it to the needs of the user depending on the context. Sensors include, for example, the Microsoft Kinect, which can recognise people as skeletons to detect their position and movements, which can be used, for example, to control actors with gestures [Patsadu et al., 2012]. Other sensors mentioned in E1 are, for example, temperature sensors and cameras for facial recognition. E5 also has several cameras integrated in the lab that will be used in the future to interact with users at home. Sensor data is also used in the other environments. In the mobile networking projects E2 and E3, monitoring data is collected by the system.

Different solution strategies were indicated for operations with multiple simultaneous users. In E1, the smart home, the laboratory is reserved exclusively for one user if their work could affect others. This is also done in the mobile networking working group in E2. If components have to be used exclusively, they are reserved for the time period. Since the individual components are divided into virtual machines in the project, only the affected virtual machines and not the entire system must be reserved. In the smart home lab E5, the student groups work mostly separately. The lab has two different, partially separated messaging platforms that should allow work on the system without affecting the production system. According to Frank, there are also plans to duplicate other central system parts, such as the Node Red installation, for a test system.

In summary, all environments are complex software systems in which complex technologies and sensors are used. In some cases, multi-user operations or the exclusive use of system resources must be organised. This complexity is expected to have a negative impact on any program comprehension and debugging tasks.

6.5.2.2 Team Composition

Team sizes in the environments studied range from one researcher with some assistance to 18 researchers and students (see Table 6.4). Students work in all teams, sometimes during practical courses at the university, for theses, or as part of the paid staff. The teams in E1 and E5 are small and regularly supervise groups of students working on parts of the project. In E2, E3, and E6 the team sizes are between seven and 18 people and include some students working on the project. In E4, there is currently one researcher, with assistance from several other people including at least one student. Since all the teams are mainly composed up of students, PhD students, and postdocs, it can be assumed that the average time a person works in a team is between 0.5 and three years, depending on the degree. If students work on the projects as part of their normal course of study, they are part of the project for a maximum of one semester. In sum, all teams undergo frequent changes, which means that knowledge about the project has to be imparted to new members, and outgoing team members have to impart their knowledge to the rest of the team frequently.

During the interviews, training periods for new team members were reported to be between a few days and up to six month (see Table 6.5). As stated by Adam, training times can depend on the specific tasks of the new team member. There are tasks that do not require extensive knowledge about the structure of the overall system, such as the development of a stand-alone component that only accesses a few well-documented interfaces of the system. In contrast, the development of a component that builds on other parts of the system and offers its own interfaces can require significantly more familiarisation time. Furthermore, the training period depends on the scope of the planned sub-project and the previous knowledge of the new team member. It is assumed that the reported training times are this far apart for these reasons.

Given the frequent changes in the team and the non-negligible training periods, it is assumed that measures to support knowledge transfer can be helpful. One measure that was mentioned during the interviews is the creation of documentation, which will be discussed in the next section. Another factor is the complexity of the system and the way it can be interacted with. If there is insufficient documentation, reverse engineering must be done to understand how the system works. The time this process takes depends on the complexity of the system and how open it is designed to be.

Environment	Team Size	Team Composition
E1	x + student groups	Researchers, groups of students
E2	7	Researchers, students
E3	18	Researchers, students (8-9)
E4	1+	Researchers, students
E5	3 + student groups	Researchers, paid students, student groups
E6	7-8	Researchers, a few students over time

TABLE 6.4: List of team sizes and composition mentioned during the interviews

6.5.2.3 Documentation

All interviewees were asked about the documentation in their working environment because it is assumed that this has a significant impact on their work with

Environment	Training Period
E1	depends on task, partially quite high
E2	6 months
E3	-
E4	two weeks
E5	a few days
E6	multiple months

Table 6.5: List of estimated training periods for new team members

the software system, especially in teams with frequently changing members, as is common in the research context.

In the environments studied, different platforms and formats are used for documentation. In all environments except the middleware project E6, students are involved in the documentation or, as in the smart home E5, are the most important authors.

While analysing the interviews, it became apparent that in some environments, the interfaces, messages, and possibly the topics were documented, but documentation about the architecture or the interrelationships in the system was not mentioned. This is the case, for example, in the smart homes E1 and E5. In all environments except E6, documentation is kept informal and to a minimum. In the middleware project E6, standardised documentation formats such as arc42 and Unified Modelling Language (UML) were used.

Problems with the documentation were noted by several interviewees. In the self-driving car project E4, the documentation is a work in progress and the architecture is still changing. Edward said that "as with any good software system, the documentation is mediocre" (Edward, translated) in the smart home lab E5. Lack of documentation was cited as the biggest problem with highly fluctuating teams. Students reportedly write little documentation and consider little documentation sufficient. Subsequent generations of students would complain about the lack of documentation. Despite awareness of the problem, no solution has yet been found in E5.

Some of the interview results show gaps in the documentation. For example, only the REST APIs are documented in the smart home E1 other documentation is not present. Dave said the documentation was not finished and in the smart home E5, Edward mentioned the lack of documentation as a big problem.

Dave and Frank highlighted the use of open-source software because it provides good documentation in the cases mentioned. However, the self-created documentation was not mentioned positively at any point during the interviews. In summary, it can be said that the documentation could be improved in all environments and that the existing documentation is helpful in some places, but in some of the environments, gaps in the documentation are already causing notable problems.

Environment	Documentation Platform	Documentation Format
E1	Interactive Website	REST API
E2	Git Wiki / Repository	Text
E3	Git Wiki / Repository	Text
E4	Work in progress	-
E5	Mediawiki	Mediawiki
E6	Documents	arc42, UML

Table 6.6: Overview of the documentation platforms and formats in the working environments of the interviewees.

6.5.3 Debugging, Testing and Program Comprehension

All participants were asked which software problems occur most frequently in the projects and how they or others in the team deal with these problems. A number of typical problems in distributed systems were mentioned, such as unreachable components and lost messages. However, there were differences in where these problems were located in the projects. Adam and Dave stated that communication via the messaging system used in the projects was not a source of errors. Bob reported that the main source of errors in the mobile networking project E2 is the very complex protocols of the OpenFlow software used there. Edward, who works in the E5 smart home environment, stated that the most common error is miscoded messages. The second most common problem he reported was messages not arriving. This reflects problems on the part of both the sender and the receiver, as well as with the order of the messages. Frank also reported that the most common problem in E5 is undelivered messages. In his opinion, this is also due to network problems, such as disruptions to the WiFi connection. Gabe also mentioned addressing problems and misdirected messages that led to undefined behaviour when agents did not know how to deal with unknown messages. The most frequently mentioned method for finding errors in the system was to observe the communication between the components.

A check of messages on the side of the sender or in the communication layer was not part of the environments examined, except in the middleware project E6. There, the messages were initially specified by XML and later in Java code. In addition, in the mobile network visualisation project E3, components were used to check the content of the JSON messages sent. However, Charlie only mentioned the content of the JSON messages and not the structure of these messages. This became most obvious in the smart home E5, where there is no common message format but all messages are sent as unformatted strings, which makes a message format check nearly impossible.

When asked how to proceed in case of a software error, no concrete methods were described during the interviews. The most common answer was that the cause of the error was found based on the accumulated experience with the software system over time. For example, based on previous knowledge, it was assumed that an error in the light control component is probably to blame if the light in a Smart Home environment is no longer adjustable. In some cases, the option of being able to observe the message traffic between components via the message layer was mentioned.

But it was noticeable that in almost all environments tools were used to read messages between components. For this purpose, small tools or scripts were built in E3 and E4 that subscribe a group to the message system and output all messages. These messages can then be checked and evaluated manually or with other tools in the next step. According to Charlie and Dave, there is no component built into the system to do this automatically. A similar approach was also reported by Gabe in the middleware project E6. In contrast, according to Frank, there is no tool in smart home E5 to gain insight into communication.

Automatic tests were noted as another possibility to find errors in individual components or in the system. In particular unit tests were mentioned. However, there were major differences in how and whether testing is carried out. In the smart home lab E1, testing is not carried out on the entire system, but only on individual components where it seems necessary. This works similarly in the smart home lab E5, where only selected individual components are tested. According to Gabe, automatic testing is generally difficult in distributed systems, which is why very little automatic testing was used in the E6 middleware project, and instead log output and visualisation tools are used.

Visualisation tools are only used in the mobile networking project E4 and middleware project E6. In both projects, the visualisation or the platform itself is the part of research. In this context, the visualisation of network topologies was the research goal of E4. In E6, the visualisation was part of the middleware to be designed. In both cases, the visualisation includes the individual components and communication relationships and it can be used to gain an overview of the system. Dave stated during the interviews that the visualisation of the communication relationships was actively used in the E6 project to localise errors.

Overall, it became clear during the interviews that no standardised or documented steps for debugging in the event of a software error are used in the environments studied. Debugging is mainly based on prior knowledge and is partially supported by tools and scripts. This includes tools for message reading as well as tools for visualising communication paths between components. Furthermore, it became clear during the interviews that hardly any error prevention mechanisms are used„ including the checking of messages or the use of unit tests.

6.5.4 Evaluation of the CEP Approach

When evaluating the CEP debugging approach, the goal of the interviews is not a general evaluation by the interviewees, but to identify further use cases and limitations for the application of this approach, to assess transferability to and suitability for other environments. It is assumed that asking the experts directly to evaluate the approach in a one-to-one interview setting may not be answered honestly. But the experts should be able to provide a professional assessment of potential use cases in environments they are familiar with. The aim is to find out whether the experts generally consider the use of CEP for debugging in their working environments to be useful and where there would be any possible problems with the integration.

At the beginning of the presentation of the CEP debugging approach, all interviewees were asked for prior knowledge about CEP, regardless of the specific use case. This question was mainly asked in order to adapt the presentation to the interviewee and to keep it as short as possible. It was also interesting to see whether CEP was already being used in the environments and what goals were being targeted with it. As shown in Table 6.7 only one participant had extensive previous knowledge about CEP. Three participants had some experience or have heard of it prior to the interviews and the three remaining participants did not know the term. This means that 4 out of 7 participants where at least familiar with the term CEP and had a basic understanding of the concepts behind CEP. In all interviews, CEP was introduced during the short presentation. Previous knowledge was compared, if available, in order to build a common understanding.

CEP has not been used in any of the environments yet. Only in E1 CEP was used by a part the team in another research project as a rule-based machine.

CEP Experience	Participants	Percent
A lot	1	14.29
Some	3	42.86
None	3	42.86

Table 6.7: Participants' previous experience with CEP

6.5.4.1 Potential Use Cases

During the interviews, all participants were asked about the use cases the presented approach of debugging and analysing the state of the system with CEP queries could be used for in the systems they know. Table 6.8 presents an overview of all the potential use cases mentioned during the interviews.

According to Adam, the smart home Lab E1 has an ontology-based reasoning system that uses linear temporal logic to detect differences between the system described with a process language and the measured state of the system at runtime. This mechanism is used to automatically react to changes or errors in the system but, according to Adam, is not used in E1 to diagnose errors during development. The analysis is based on the synchronised state of the entire system, which is kept in a database. However, according to Adam, it is not possible to use this system with high-frequency sensor data, such as skeletal data from the Microsoft Kinect, because the ontology mechanism is much too slow for this. Furthermore, everything in the system that wants to communicate with each other and that is to be analysed must be covered by the semantic ontology representation. This is why, according to Adam, communication in E1 does not run via a broadcast mechanism.

CEP could be used here to make statements about the system components at runtime as well as during debugging by the staff. A CEP engine is also capable of using logical expressions and temporal logic to recognise relationships and could possibly implement similar functionality without the limitations mentioned. As shown above, CEP is applicable to the processing of distributed event streams and high-frequency sensor data. However, a comparison of the processing capabilities or usefulness of the two systems is not possible based on the interview.

During the interview with Bob and Gabe, it was suggested that the CEP approach could be used to create test settings for experiments. In addition, Bob said that CEP queries could be used to configure and synchronise system components to send specific messages. This is made possible by the seamless integration of the

CEP Engine, which allows CEP queries to access the current system state and send messages to agents for configuration. In addition, it is possible to start agents via CEP queries to create any test settings and, if necessary, to check again via CEP queries whether the setting is correct and the test is successful. Gabe added the example of a production line in which status can be measured via monitoring agents and evaluated via CEP queries to identify error states and detect failed components.

Charlie described the challenge of processing metrics from all agents simultaneously in the mobile networking visualisation project E3. A distributed and scalable CEP approach could help process the metrics without producing bottlenecks or negatively affecting the rest of the system. These metrics could then be used for the system itself as well as during development. Using CEP queries, it would be possible to control exactly at what point in time the evaluations of the metrics must be carried out. In addition, ad hoc analyses could be started with the help of CEP queries.

Dave envisioned many useful applications for the CEP approach presented. Among other things, he stated that it would be helpful for his work to be able to see which components are currently running and what they are currently doing in order to compare this with his expectations of system behaviour. He emphasises the possibility of dynamically analysing the systems' behaviour during runtime and reacting to it with defined rules. One example he gave was the dynamic automatic starting and stopping of processes based on the system state. The CEP approach could help keep track of the management of components in a loosely coupled system.

Edward and Frank, who work the smart home Lab E5, and Gabe, who worked on the middleware project E6, stated that the CEP approach could be used for error detection. According to Edward, to write unit tests, one needs to know in advance what errors are expected. This knowledge can be used with CEP queries to detect irregularities in the system, such as to check whether a component really only communicates with the components it is allowed to talk to. Especially in loosely coupled systems, it could be helpful to quickly detect components that send messages that are not appropriate for the situation or even necessary. One could set rules for the framework and intervals at which interactions are expected to recognise deviations. Frank explains this using the example of a sensor that regularly sends data in a certain form. With the help of a CEP query, one could check this behaviour and detect deviations from it.

Frank mentions that it would be helpful for debugging tasks to record data streams in order to be able to repeatedly examine processes from different perspectives in the event of an error. CEP queries can be used to temporarily store event streams and also to permanently store them in individual cases. For example, the last messages to an agent could be saved when an error occurs. However, saving the all messages in the system in order to examine any processes afterwards is not part of the presented approach and would have to be investigated further.

In summary, interviewees proposed several use cases, that could help at runtime, during debugging tasks, and when testing individual components or the entire system. This supports that, based on the idea of the approach, the interviewees see advantages to using it in the environments they know.

Use Case	Interview	Runtime	Debugging	Testing
Reasoning about system state	1 / Adam	x	x	x
Processing of sensor data	1 / Adam	x	x	x
Test settings	2 / Bob 7 / Gabe			x
Metrics processing	3 / Charlie	x	x	x
Dynamic system control	4 / Dave	x	x	x
Error detection	5 / Edward 6 / Frank 7 / Gabe	x	x	
Unit tests	5 / Edward			x
Event recording	5 / Edward	x	x	

TABLE 6.8: Identified use cases mentioned during the interviews

6.5.4.2 Possible Limitations

Limitation	Aspect	Interview
Increased practice time	Complexity	1 / Adam
Needs structured messages	Messaging	2 / Bob
Performance	Performance	3 / Charlie
Invalid messages	Messaging	5 / Edward
Messaging system compatibility	Messaging	5 / Edward

Table 6.9: Limitations mentioned during the interviews

One aspect of the evaluation was to further investigate possible limitations of the CEP approach. Multiple limitations were mentioned during the interviews, as listed in Table 6.9.

Adam pointed out during the interviews that complex technologies, such as ontology-based reasoning, can be a significant hurdle for students to work with, and therefore do not speed up development. Since the CEP approach also brings additional complexity into the system and the work with CEP queries has to be learned, there could be similar problems here. However, one reported advantage of the seamless integration of an CEP engine into an agent-based system, which was presented in this thesis, is that the complexity is not extended by an additional architecture paradigm and everything can still be considered an agent.

The CEP approach assumes that messages follow a schema or have a structure that can be processed automatically. Bob expressed concerns during the interview that this might not be possible for low level network messages, such as in mobile networking project E2. It is likely that this is due to the example chosen during the interviews. Here, JSON messages were used, which, unlike the messages used in project E2, are easily readable by humans. Further research could indicate whether network packets are also compatible with the CEP approach. However, since network packets also follow a fixed structure, for example, they probably contain the IP address of the sender and the IP address of the recipient, it can be assumed that they at least partially contain attributes that can be made accessible for a CEP query. However, this would only be possible with an extension of the presented CEP engine, as it can currently only process JSON and other text messages.

Charlie was concerned about the performance characteristics of the CEP approach. He noted that the system must be able to process great deal of information in real time and that there could be a bottleneck. These concerns are understandable given how important message latency is in the systems being addressed. The latency of messages and the scalability of the system, including the CEP engine, are investigated in Chapter 7, to demonstrate that the architecture presented is more than fast enough for its planned use in smart systems.

Edward noted during the interviews that coding errors in the messages cannot be handled by CEP queries. Because they have an incorrect structure, or flipped bits for example, the messages cannot be parsed and processed by the CEP engine. While this is correct, the CEP engine is still able to mark the unprocessable messages and make them available to the user. However, it is not possible to access individual attributes of the message when it is not parsable. Nonetheless, there are ways to deal with such errors. For example, as with the middleware presented in this paper, frameworks can be used to create and check messages before they are sent. In addition, such messages with an incorrect structure should also trigger errors for the recipients of the message, which can then be found again via CEP queries.

In addition, Edward observed that CEP queries can only work with messaging systems that provide the necessary information, such as a sender and receiver. According to Edward, this is not the case with a Controller Area Network (CAN) bus. CAN is often used for communication between microcontrollers, such as in the smart home lab E1. The messages sent through this would not be part of the publish/subscribe system and therefore would not be visible to the rest of the system or the CEP engine. It remains unanswered whether it is possible to provide an adapter for this and to encode missing information within the messages to make CAN compatible with the approach presented here.

In addition, according to Edward, CEP queries are only worth using for environments with sufficient complexity, because otherwise the additional effort, such as learning the query language, is not worth it. However, the identified use cases for this approach suggest that it is worthwhile to use it for systems of similar complexity to those examined here.

6.6 Conclusion

The expert interviews were conducted to provide insights into the following research questions.

- **QI1**: Which software architectures are used in research laboratories for loosely coupled systems and which requirements are to be met by them?
- **QI2**: What do program comprehension and debugging tasks look like in research laboratories with loosely coupled software architectures?
- **QI3**: To what extend can the presented method support program comprehension and debugging tasks with the help of CEP in smart system research laboratories?

Firstly, the software architectures used in the environments studied were analysed for *QI1*. In the evaluation of the expert interviews, particular attention was paid to the additional requirements for the software that resulted from the research context. The software environments addressed during the interviews were very different. Different message platforms and message formats were used. Overall, the results are consistent with the expected heterogeneity of software in this type of environment. This is not only noticeable in the comparison between the environments but is also evident within them. Especially in the two smart home labs, many different technologies are in use. In some cases, several protocols are even used for the communication layer within one environment in order to connect everything.

This is also reflected in the choice of software and protocols for communication. Message brokers such as MQTT, RabbitMQ, and ZeroMQ were used most frequently. These offer ready-made open source software libraries for connection with different programming languages and technologies. MQTT is frequently used for IoT applications, and can also be used for microcontrollers, for example. In the choice of message formats, text-based formats predominated. JSON, XML, and unformatted text were mentioned here. These results support the design decisions made for the software architecture of the middleware in this thesis because it is also based a on structured text format and, like the mentioned message broker, on a publish/subscribe approach.

During the interviews, there were no statements that contradicted the general requirements that emerged for middleware for loosely coupled systems in research laboratories during the analysis of this thesis. These include requirements such as scalability, message latencies, and flexibility. Agent management features were not mentioned during the interviews, except in the middleware project E6.

The challenges identified in research environments with loosely coupled software were confirmed by the interview results. The results of the interviews indicate that finding errors and getting to know the existing system can be particular challenges, which are further complicated by these system and environment characteristics. This leads to two important requirements for the architecture in these environments: *support for debugging* and *program comprehension support*.

The teams working on the projects are partly composed of undergraduate and graduate students, who are likely to be involved in the project for only a few semesters or a few years. In addition, the assumed issues with documentation and maintenance of student projects were confirmed. It is therefore even more important to support the transfer of knowledge and the introduction of new team members, which are part of the analysed requirements as *support for communication between developers* and *support for fast experiments*.

#	Requirement
5	Support for debugging
6	Program comprehension support
7	Support for fast experiments
10	Support of communication between developers

Table 6.10: List of requirements that were added based on the expert interviews, with the corresponding numbers from the requirement analysis in Table 3.1

As a result, the requirements for the architecture for laboratory environments were extended by the four requirements in Table 6.10 resulting from the expert interviews.

The experts gave many reasons for why the environments they work in can be very complex. These include the use of complex software, working with multiple platforms, and distributed applications over the network. The results of the expert interviews regarding *QI2* demonstrate, that program comprehension and debugging tasks can be very challenging. It can therefore be assumed that tools will be needed in these environments to support the work with these systems. Common problems mentioned in relation to the loosely coupled system were faulty components, messages not arriving, and formatting errors in messages. However, it is noticeable that in most environments no software is used to support these tasks, but individual developers help themselves with their own scripts to, for example, read messages. A visualisation of system states was only offered where it was explicitly the subject of research. No documented or standardised procedures were used in the project to search for errors. According to the experts, the procedure and the localisation of errors are mainly based on experience. Additionally, other methods to reduce software errors, like automatic testing or verification of components or messages, were not part of the majority of the projects addressed.

The complexity of the systems, the reported problems, and the lack of software support and documented procedures all indicate that it would be helpful to use a middleware designed to address this challenge during the development phase.

Regarding *QI3*, the results of the expert interviews indicate that the developed approach to supporting Program Comprehension and debugging tasks with CEP is suitable for the collected use cases in the development of software with loosely coupled systems. This includes reasoning about system state, processing of sensor data, dynamic control of the system, test setup preparations and unit tests, error detection, processing of metrics, and event recording. The use of the query language is similar to other approaches to do context processing and can also be used for this purpose. This reduces the necessary training time because the query language can be used equally for both tasks, further supporting the argument that it is worthwhile for these environments to use CEP for development tasks to support the use cases.

Several limitations were noted during the interviews. In addition to the initial requirement that it be a loosely coupled system, limitations include the need to work with structured messages and the need for information about the communication to be provided by the messaging platform. The former is already present in a large majority of the systems mentioned during the interviews. The use of structured messages is often necessary to enable the automatic processing of contextual information and to check messages for correctness. The latter, the messaging platform capabilities, are missing in some of the systems analysed. The use cases demonstrate here demonstrates that it can be helpful to use platforms that are able to provide information about the agents and their communication relationships. This is one reason why a separate middleware layer for the communication layer was built into the presented architecture. The architecture is designed to connect other messaging systems as easily as possible in order to use the CEP layer for as many other loosely coupled systems as possible. For this, however, either the message layer or the individual agents themselves must be able to provide information about the agents' status.

It was also noted that the system is expected to be slowed down by this approach, and the training time could be negatively affected by the additional complexity. The former is addressed in the following chapter with the evaluation of message latency and scalability to showcase that the system's performance is more than fast enough for the targeted environments. The training time required is an important aspect in deciding whether the CEP approach is worthwhile for a specific environment and team. But testing this is beyond the scope of this thesis. It is assumed here that the benefits demonstrated are worth the additional training time.

The implications of these findings for the research questions are discussed in the following conclusion of the thesis.

Chapter 7

Experimental Evaluation

Several experiments were conducted to evaluate the requirements for message latency and scalability of the system. The experimental set-up and the exact procedures are explained first, and then the results are presented and evaluated in relation to the requirements. The procedure for the evaluation through the latency measurements is based on Eichler et al. [2017], in which an earlier version of the messaging layer of the architecture presented here was tested.

7.1 Experimental Set-up

For the experimental setup, seven virtual machines with the same configuration were used, which were connected via a 10 gigabit host network. Each machine was assigned eight CPU cores and four gigabytes of RAM. The host system had an AMD EPYC 7542 32-core processor. The virtual machines were installed based on the same pre-built Centos 8 stream template. A script was then used to install JVM 11 and the software presented in this thesis.

After installation, the software was started via a script on all virtual machines involved in the measurement. First, the middleware was started on all machines, followed by the runtime environments, including an agent for managing the local runtime environment. As soon as all these runtime control agents are ready and

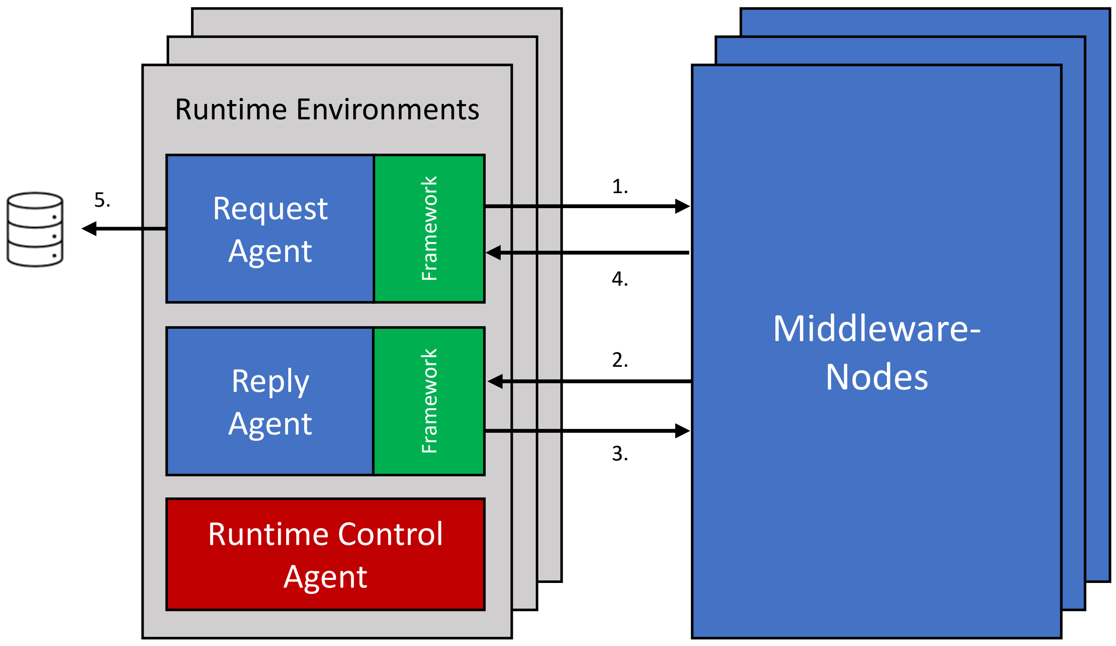


FIGURE 7.1: Experiment setup for the latency and scalability measurements

report that the connection to the middleware has been successfully established, the necessary number of agents are started. After about 10 seconds, the agents start sending test messages, and the measurement can be started. After each measurement, all runtime environments and middleware nodes are restarted to ensure that the individual measurements do not influence each other.

Each virtual machine runs a middleware node and a runtime environment. The middleware nodes form a cluster and thus allow the communication of agents within and between the runtime environments. All communication between the agents runs via the middleware. This experimental setup was chosen because it follows the recommendations of the architecture presented. All agents are executed in runtime environments, allowing the libraries to be shared in memory and many agents to run on one host. Each runtime environment is assigned to a middleware node. In this way, communication between the agents running inside the runtime environments and the middleware node have low latency because it runs over the local network.

Figure 7.1 offers a simplified representation of the experiment setup and the sequence of messages during an experiment. Two types of agents are used for the experiments: a request agent and a reply agent. Both agents are programmed in Scala and use the provided JVM framework for their implementation. The request agent sends messages with a current local timestamp at a configured interval. This message is serialised by the framework into a JSON message and published via the group communication of the middleware. Each request agent is assigned a reply agent that listens to the group in which the request agent sends. Therefore, each request sent is received, deserialised and processed by a reply agent. The reply agent copies the received timestamp into a new message and sends it back to the sender via another group. It does not contain any additional logic that could impact the performance of the experiment. The response is also serialised in JSON and deserialised again by the receiving agent, the request agent. The timestamp contained in the message is then compared with the current system time and the difference is written to a file locally in the runtime environment of the agent. A measured value therefore contains the complete round trip, two times via the framework through the network, two serialisations, and two deserialisations.

This procedure creates a file with all measured values per runtime environment for each experiment, which are merged for the evaluation of the latency values. The results of the evaluation of the measured values are presented below.

7.2 Messaging Layer Latency

Two experiments were conducted with the described setup to measure the message latencies in the system.

The first experiment tested the single node performance of the middleware. As the system is also designed for development and test environments, it is important that it can run on single workstations and servers.

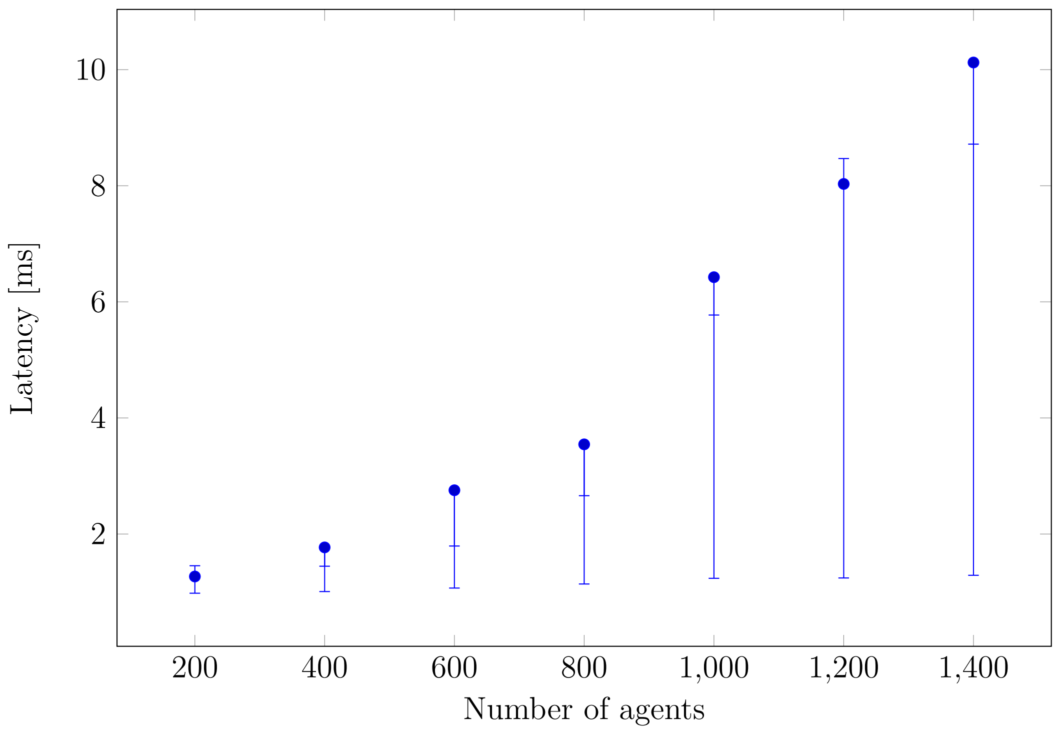


FIGURE 7.2: Latency measurements including the 10th and 90th percentiles with different numbers of agents and one middleware node

For the test, a server with 8 CPU cores and 4 gigabits of RAM, as described above, was used, which should provide a realistic representation of an average development computer. Figure 7.2 shows the latencies for different numbers of agents in the same runtime environment together with a single middleware node.

With 200 to 800 agents, the average message latency is less than 4 ms. Even with 1,400 agents, the latencies are still around 10 ms. As the number of agents and messages increases, so does the average message latency and the scatter of the measurements. In the figure, the 10th (at the bottom) and 90th percentile (at the top) are shown for each measured value. Since the middleware tries to deliver all messages as quickly as possible, under load many messages arrive with a latency comparable to that of a few agents, but some are delayed, which leads to the illustrated distribution.

In all tests, less than 1 gigabyte of Random-Access Memory (RAM) was used. After starting all agents at the beginning of the measurement, the amount of memory used by the processes was determined via the task manager of the operating system. Figure 7.3 provides an approximately linear progression of the memory requirements of the middleware node and the runtime environment with a simultaneous increase in the number of agents, groups, and messages per second. The required memory of approximately 500 to 800 MB should be available on current development systems. Since the application, tested here was programmed in Scala and runs on the JVM, there are fluctuations in the measured values. The JVM reserves memory, manages it for the application and releases freed memory areas via the garbage collector. The application itself has no direct control over the exact amount of reserved memory.

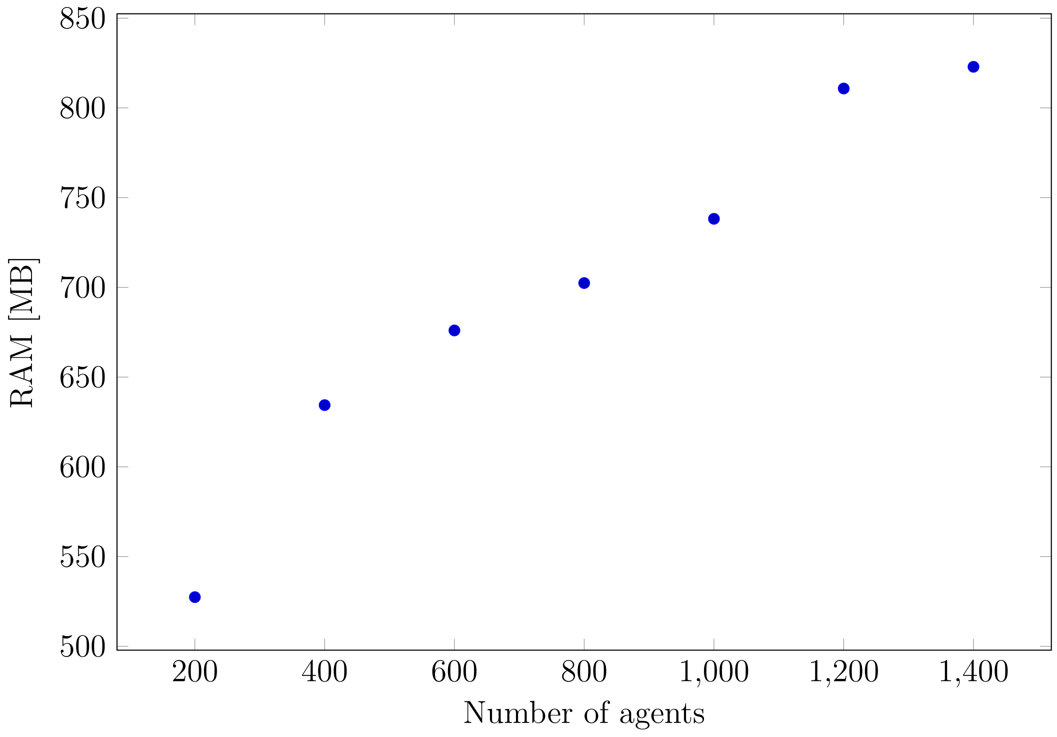


FIGURE 7.3: Main memory with different numbers of agents on one middleware node

The results from the expert interviews include research environments with up to 1,000 agents, and it can be assumed that the entire system does not have to be run on the development computer. For example, many hardware sensors and actuators will not be able to run on the development computer or will have to be simulated. Therefore, the implementation tested here is more than fast enough to run simulations during the development process of new agents and even to test the entire laboratory environment on a single computer if necessary.

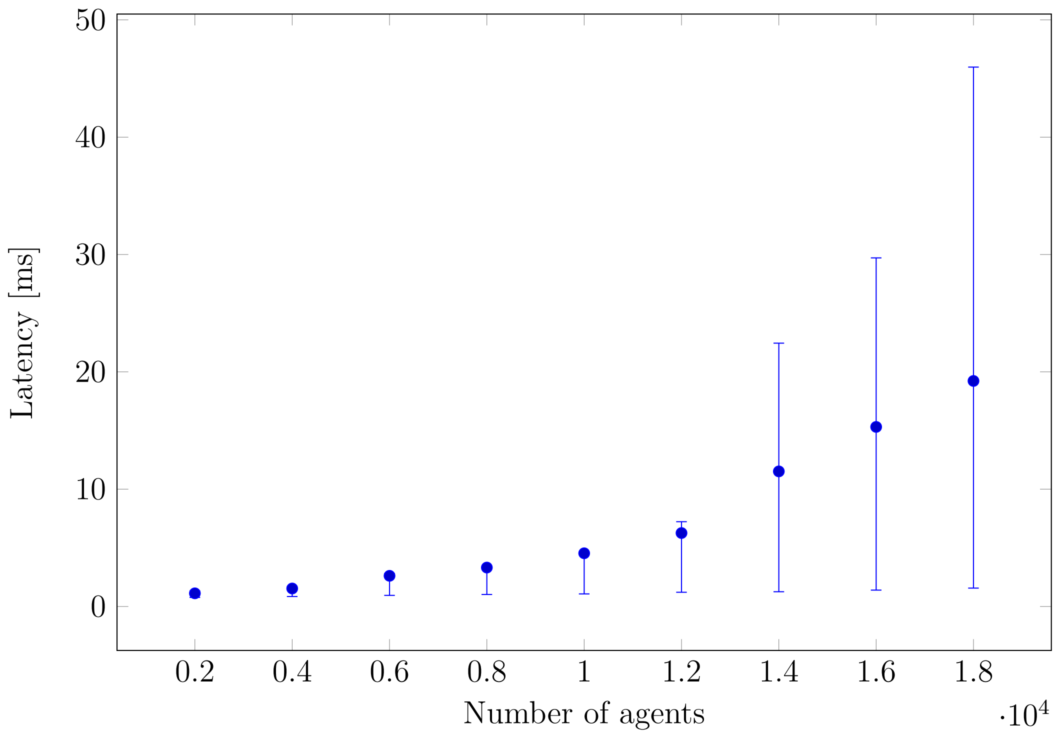


Figure 7.4: Latency measurements including the 10th and 90th percentiles with different numbers of agents and seven middleware nodes

In the second test, the system performance was tested with seven middleware nodes and seven runtime environments, which represents the recommended configuration of the installation in a laboratory environment. Due to the fault tolerance of the middleware, a failure of a total of three nodes can be tolerated in this configuration without impairing the functioning of the system.

Figure 7.4 presents the results of the measurements. With seven nodes, the average message latency is less than 6.5 ms for up to 12,000 agents and groups sending a total of 12,000 messages per second. With higher numbers of agents, the latency increases averaging around 20 ms for 18,000 agents. With increasing load in the system, the scatter of the measured values increases, and from 14,000 agents, groups, and messages per second, the number of messages that are slower than the average increases significantly. With a high number of messages, the CPU of the test machines is near maximum capacity, and delays can occur during message handling. It is assumed that this is caused, among other things, by the garbage collection of the JVM. Therefore, it is advisable to use more nodes in case of high CPU loads, which can also be done automatically, based on the average load of the individual nodes.

To interpret these results, it is important to consider that the numbers of agents tested here are extremely high for smart systems, especially for laboratory environments. The environments tested during this work and in the expert interviews consisted of about 100 agents, and the largest system had a maximum of 1,000 agents. Furthermore, in the tests conducted, one group was set up for each agent, and one message per second was sent. These message numbers should also have very high values on average for a smart system, since many components, such as sensors and actuators, communicate much less frequently.

7.3 Scalability

The variable requirements of the system in terms of the required number of agents and groups in a smart system should allow the system to scale with the requirements. This was tested with the next experiment. Latency measurements are made analogously to the previous tests, but this time have a fixed number of agents on a variable number of middleware nodes.

Figure 7.5 shows the average message latencies with 10,000 agents and the same number of groups and messages per second, using two to seven middleware nodes and an equal number of runtime environments. It can be seen that with two and three nodes the average latency is still very high, and 90th quantile is still at more than 500 to 600 ms. However, with four nodes, these values go down significantly, and at five nodes the average message latency is about 15 ms. At seven nodes, it decreases further to about 4 ms.

For this load on the system, an installation with five or seven nodes would therefore make sense in order to have a low latency and an error tolerance of two or three nodes.

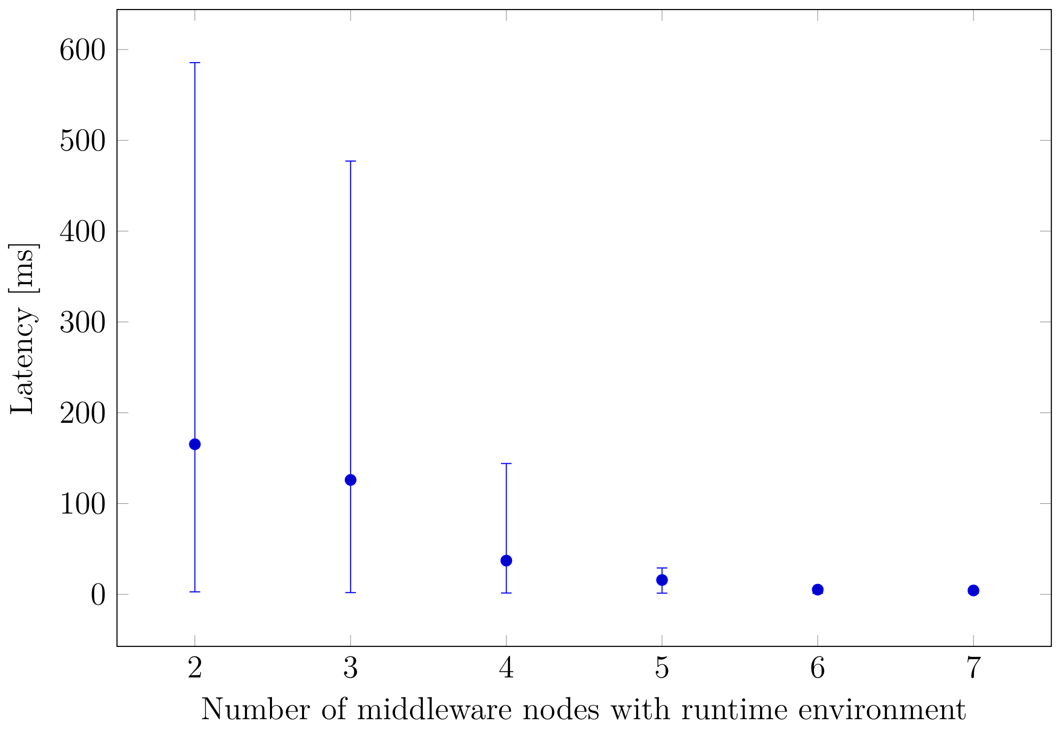


FIGURE 7.5: Latency measurements with 10,000 agents and different numbers of middleware nodes

For the test carried out here, very high numbers of agents were tested to measure the behaviour of the system under load. With the determined requirements of up to 1,000 agents in laboratory environments, a system with five or seven middleware nodes is more than sufficient, even with high message volumes per second. The expert interviews revealed that some systems have less than 100 agents, which could start with one node and increase the number of nodes if necessary or if failure tolerance is required. This test illustrates that the middleware can be scaled flexibly to the needs of the environments in which it is to be used and can be used for development on one computer as well as for testing system parts in small laboratories and in larger installations.

7.4 CEP Integration Overhead

Since all CEP queries are processed by agents communicating via the messaging layer, the latency of messages resulting from CEP queries depends on the performance of the messaging layer. Each step in the processing chain that results from a CEP query publishes the message, and the next agent then receives it through its subscription. Thus, when l_b is the latency of the base system and n is the number of processing steps, the latency of a message is $l_b * n + c$, where c is the sum of the overall processing time a message takes inside each agent in the processing chain.

The number of agents needed for a CEP query depends on the query and can be influenced by the CEP query manager. Depending on the configuration of the system, it is attempted to split the processing of messages into several agents to increase modularity and reuse of agents or to execute as much as possible in one agent to keep the latency as low as possible.

Regardless, the system creates only a few agents for each CEP query. Because of this, and because multiple queries share agents depending on the configuration, the number of agents remains in the hundreds, not thousands, when using CEP queries.

Multiple queries can share parts of the agent processing chain to reduce the number of components. The results of the previous experiments support that this can be easily processed by the system with low latencies.

7.5 Conclusion

The results of the latency and scalability experiments support that the messaging layer is more than sufficient for the targeted use cases and the architectures' identified requirements of low message latency and high scalability are met. This concerns both the number of agents and groups and the number of messages that the system can process. It must be emphasised that the design of this architecture focuses on supporting the development process and not on the performance of the system. It can be assumed that by omitting the registration and monitoring of agents and exchanging the message format, latencies could be significantly improved at the expense of these development support features.

User interaction within a smart environment can have different characteristics and thus different latency requirements. Arapakis et al. [2014] studied the impact of latency on user behaviour during web searches. They found that with response times of less than 500 ms, users were unable to distinguish whether the response was normal or artificially delayed. This would be comparable, for example, to a question about the current weather via an intelligent table in a smart home.

Claypool [2005] measured the effect of latency on performance in real-time strategy games, discovering that there was no significant effect on player performance at latencies of less than hundreds of milliseconds to several seconds. This use case could be compared to game-based approaches to intelligent surfaces.

Deber et al. [2015] research on the user perception of latency on touch surfaces demonstrates that the detectable threshold for dragging is between 11 ms, when the reaction is visible on the same screen that is touched, and 55 ms, when the reaction is visible in another place, and for tapping these times are 69 ms to 96 ms, respectively. Improvements in latency could be perceived in dragging from 8.3 ms, depending on the baseline latency. These latency values are comparable to the targeted latency times for interaction on touch surfaces, such as a smart bathroom mirror in a smart home. It can be assumed that, in order to reduce latency, dragging operations are processed locally in the application, and actions are triggered when they are released or remain at a certain position for a longer period of time. This means that only the touch interaction time is relevant for the messages over the network. Compared to the values measured here at approximately 3 to 5 ms with heavy utilisation, these use cases can be implemented without restrictions via the architecture presented.

An evaluation of the performance of different MQTT implementations by Bender et al. [2021] shows a message latency of about 20 ms to about 50 ms, depending on the implementation, in a measurement with 2,000 clients who each publish 20 messages on a topic and subscribe to it. Compared to the values measured here, this is faster when using one middleware node but is still in a similar order of magnitude. The nearest comparable measurement is 1,400 agents and groups, where the average message latency is 10 ms. In this measurement, the test VM is already close to its CPU limit, so a measurement with 2,000 agents will be slower. The higher message latency is to be expected and can be explained by the fact that the middleware does additional work, such as for monitoring the agents and counting of the messages.

It is difficult to make a good comparison of message latencies with similar publications that support developers in their work on smart systems. The latencies of the messages depend on the hardware used, the network technology, and the test setting. Of the four comparable works with performance measurements listed in Table 2.2, only Pahl and Liebald [2019] include latency and scalability measurements, with approximately 10 ms latency over the network for a single request. The scalability was tested only for up to 48 simultaneously running IoT services. The other publications only measure the latencies of certain elements of the architecture, such as the retrieval of data from the database or the rendering time of ontologies. Gu et al. [2005] and Elhabbash et al. [2020] measure the duration of requests to the ontology system. While this is not comparable to the latency of messages between two agents, it shows that for certain requests it is acceptable to wait several seconds for a response. For example, with 100 simultaneous requests, a requests lasts more than 25 seconds in Gu et al. [2005] and approximately 20 seconds in Elhabbash et al. [2020]. The overhead of the architecture presented here of a few milliseconds would therefore be negligible for such approaches.

The other requirements for the architecture identified in Chapter 3 are evaluated in the following chapter. Since these are requirements that do not concern performance in the sense of the speed of processing messages by the system but are mainly focused on usability, they are evaluated separately in a scenario-based evaluation and are not tested quantitatively via further experiments.

Chapter 8

Scenario-based Evaluation

In the following, a scenario-based architecture analysis is carried out as described in Chapter 5 to show that the software architecture presented here fulfils the identified requirements and is suitable for use in smart systems. For this purpose, the preparations and the procedure of the architecture analysis are described first, including the identification of architectural approaches and scenarios as a basis for the ATAM workshop. The results of the workshop are then presented and evaluated.

8.1 Architecture Analysis

The analysis of the architecture with the help of the scenarios starts with the identification of the architectural approaches and the selection of the scenarios. This is followed by the ATAM workshop, in which the actual evaluation takes place.

8.1.1 Architectural Approaches

In the following, the architectural approaches of the architecture presented here are listed and briefly explained. All aspects of the architecture addressed here are described in detail in Chapter 4.

- **A1 - Agent-based Design** - An agent-based architecture is used strictly for the whole system. All components have dedicated tasks and communicate with each other exclusively via messages. The only exception to this are the middleware nodes, which provide the communication interface for the other components.
- **A2 - Open Design** - The architecture is designed so that all components and their communication are easy for developers to inspect. Therefore, all messages are routed through the middleware's messaging layer. Furthermore, all messages are in a format that can be converted to JSON so that they can be easily read by humans. The interfaces to the middleware are based on open-source libraries and standardised protocols to facilitate adaptation to other systems and increase the system's integrability.
- **A3 - Publish/Subscribe Communication** - All communication between agents is sent via publish/subscribe. A direct message between two agents is not allowed. This ensures that all communication channels are visible to the middleware, and developers can view the exchanged messages at any time.
- **A4 - Open Middleware API** - The interface of the middleware for connecting agents is designed in such a way that it can be easily connected from as many programming languages as possible. To make this possible, there are three entry points. Firstly, the optimised interface is based on SCTP and Protobuf, which is used by the frameworks. In addition, there is the possibility to connect via TCP, UDP, and WebSockets with a simplified interface and JSON messages. It is assumed that every modern programming language supports at least one of these protocols and that JSON can be used everywhere as simple strings, if necessary. As part of the middleware API, all available middleware nodes are regularly transmitted to the agents to enable a failover should one of the middleware nodes fail.

- **A5 - Agent Framework** - For the simple connection of frequently used programming languages, a framework is provided that can be integrated into an agent project as a library and that can control the life cycle of the component as well as management of the communication with the middleware and serialisation of the messages. The main goal of the framework is to accelerate the development of new components and further reduce the complexity of connecting new agents.

- **A6 - API Libraries** - All agent interfaces are defined via a uniform description language. This makes it possible to check the validity of messages in groups. In addition, the middleware can specify at any time which messages are permitted where, which is helpful in understanding the system.

- **A7 - Runtime Environments** - Agents can be programmed, for example via the provided framework, in such a way that they can be operated in the runtime environments of the middleware. This enables full control of the agent lifecycle via a runtime management agent interface. Agents can migrate between runtime environments, which provides failure tolerance in case of an error. However, the agent must be either stateless (e.g., external database) or programmed in such a way that the state can be restored after the migration (e.g. via the event sourcing mechanism of the framework).

- **A8 - Integration of CEP** - The architecture contains a fully functional, seamlessly integrated CEP engine consisting entirely of agents. This engine allows descriptive queries to be executed by developers and by agents that provide dynamically updating results. All messages can be queried, as well as the state of the system as maintained by the middleware.

- **A9 - Distributed Middleware Nodes** - The middleware messaging layer is provided by the middleware nodes. Several middleware nodes can be operated in a cluster. The nodes monitor each other and automatically disconnect unreachable nodes from the cluster. Decisions about unreachability are determined by a chosen leader, which is automatically determined by the cluster. This allows the processing of messages from connected agents to be shared between the middleware nodes and provides fault tolerance. In case of failure of one of the middleware nodes, the agents affected automatically connect to one of the remaining nodes. The system remains operational as long as more than half of the original nodes are running.

- **A10 - Layer Design** - The middleware consists of three layers that build on each other. The publish/subscribe communication is provided by the lowest layer, the messaging layer. Based on this, the CEP layer processes the CEP queries from agents and users and makes the system status information of the lower layer accessible. The top layer is the interaction layer, which contains the interface for the developer. Here, the status information of the messaging layer is further processed and prepared for display to the user. In addition, the control of the user over the runtime environments and ad hoc CEP queries is managed in this layer.

- **A11 - Monitoring and Logging** - All agents are registered with the middleware during the initial connection. The middleware regularly checks the availability of agents and manages the subscriptions of all agents. At the request of an agent or by a user request, messages can be stored in one or more groups for a specified time interval. For this task, another agent is started by the middleware, which subscribes to the specified groups and saves all received messages ordered by their time stamp. These messages can then be replayed at a later time, for example for testing purposes.

8.1.2 Quality Attribute Utility Tree

As usual, according to the ATAM, the scenarios are compiled in two steps [Clements et al., 2009]. In the first step, the quality attribute utility tree is created. This contains all scenarios that result from the requirements for the architecture determined so far. Since the architecture presented here has already been in use for several years at the time of writing, the experiences from various tests are included. This includes the experiences from operation in the two research laboratories, the Living Place Hamburg and the CSTI, the use of the architecture in several research projects, and experiences from the use of the architecture for a total of five university courses with students.

8.1.3 Scenario Prioritisation

Questionnaires are used to collect further scenarios and to subsequently evaluate the importance of these scenarios. In the first step, all available stakeholders are asked to rate the scenarios from the Quality Attribute Utility Tree according to their importance. They are also asked about other scenarios that they think are missing. The newly collected scenarios are then evaluated by the participants of the ATAM in a second questionnaire.

The first questionnaire asked about the importance of the 15 scenarios from the quality attribute utility tree. The complete questionnaire can be found in Section B.1. All participants were asked to describe the professional role they predominantly had at the time they worked at Living Place Hamburg or CSTI and could select multiple answers. This was useful, for example, for students or PhD students who were also employed as staff. The questionnaire was sent to all current and former staff and students of Living Place Hamburg and CSTI who were available. The prerequisite for participation was that they have experience working with the software architecture in the labs, for example in the context of a project, a thesis, student supervision, or working on the infrastructure of the labs. When specifying the role in the questionnaire, there were several questions because it was not clearly defined what time it was and what employment as a staff member meant in connection with the work on the architecture. It was ensured that only people who worked with the system as such were counted as employees. A parallel job in another project was not sufficient.

A total of 18 people were surveyed. All but one of the questionnaires were completed in full. One of the questionnaires was missing an evaluation of one of the scenarios. Figure 8.1 shows the distribution of roles. Twelve of the participants were students, five were staff, three were PhD students and three were professors.

Quality Attribute	Refinement	Scenario
Performance	Message Latency	Developers can use the system to implement user interaction with reasonable message latencies.
	Scalability	The performance of the system scales with the addition of more middleware nodes up to at least 1,000 agents.
Availability	Node Failures	System functions are available as long as more than half of all middleware nodes are available.
	Agent Failures	Uncontrolled sending of messages from an agent does not affect the functioning of the system.
Modifiability	Add Technologies	Developers can connect additional technologies, like other messaging systems or previously unsupported but compatible protocols, to the middleware within a day.
	Descriptive Programming	Developers can create simple full-featured agents via a descriptive language.
	Change Messaging Format	Developers can add or change message formats within a day.
Variability	Heterogeneity	Developers can connect new components written in commonly used programming languages (such as Java, C, and JavaScript) to the middleware in a matter of hours.
Modularity	Reusability of system components	Administrators can exchange system components and use them independently of each other.
Functionality	Context Processing	The system offers developers the possibility to subscribe to, edit, filter and forward context information from agents via queries,
	Program Comprehension	The system can show developers the current communication graph, filtered on demand, with all agents and groups.
	Debugging	Developers can search for specific agents and messages with specific properties via queries and thus localise errors in the system.
	Adhoc Queries	Developers can query the system state and messages of agents ad hoc via queries to interactively learn about the structure of a running system.
	Test Settings	Developers can generate test settings and test data with the help of the system without having to write their own components.
Conceptual integrity	Agent-based	Developers can view all components of the system as agents with dedicated tasks.

TABLE 8.1: Quality attribute utility tree

The comparatively high number of students can be explained by the composition of the lab teams. It is common in such environments in the research sector that there are several students for every staff member and professor.

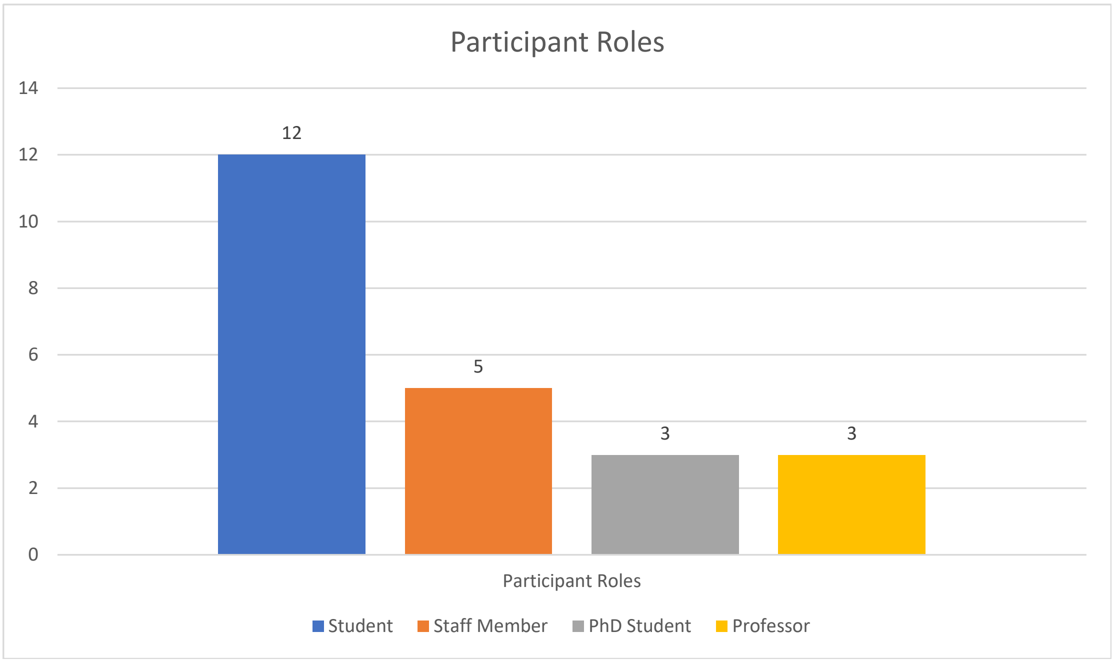


FIGURE 8.1: Distribution of professional roles among the participants of the first questionnaire

The evaluation of the 15 scenarios can be found in Table 8.2. Overall, all scenarios were considered at least moderately important on average. This indicates that the selected scenarios for the applications in the research laboratory are suitable for the target groups surveyed. Only one scenario was rated as very important, with a score of 4.5 (rounded up). This was the scenario of integrating new components written with different programming languages into the system. This result supports existing literature, according to which support for heterogeneity is an important factor for all forms of smart systems.

Most scenarios were rated as important. Among the higher-rated features are the search for faulty components, the possibility to process context information with CEP queries, and the system's ability to tolerate single faulty agents. In addition, message latencies suitable for user interaction and the ability to exchange message formats quickly were rated as important. Interestingly, the possibility of simply searching for errors was rated higher than both performance-relevant scenarios.

Two scenarios were assessed as only moderately important the scalability of the system to up to 1,000 agents and the possibility of agents with CEP queries. Regarding scalability, based on feedback after the questionnaires and during the workshop, it is assumed that 1,000 agents was perceived as very high as many students and staff are normally confronted with only a few components at a time.

When analysing the questionnaires, it became apparent that the different stakeholders voted differently depending on their roles. To illustrate this, the voting results of two groups were compared. The first group comprised all the students, and the second group comprised the all staff and professors. The differences in the average voting results are outlined in Figure 8.2. The group of staff and professors rated all scenarios in the *functionality* category higher than the group of students. The differences range from about half a level of importance to almost a whole level of importance. In contrast, students rated the performance scenarios as more than half level higher. This difference is even higher for modification scenario 7 (change of message format) and conceptual integrity. Here, the group of students rated the scenarios almost one importance level higher.

It could be assumed that, in comparison, staff and professors value the functionalities of the architecture that are designed to improve debugging and support program comprehension more because they potentially work with more team members, for example when supervising student projects, and are potentially part of the team for a longer period of time. In contrast, students in these environments are predominantly involved in the practical work on the system, where the performance and the simple modification of components are important.

This cannot be definitively deduced on the basis of the data, but it does indicate that it could be interesting to do another survey that focuses on such differences. This could also be done independently of the architecture and could therefore include other laboratory environments to achieve a higher number of participants.

	Category	**Refinement**	**Scenario**	**Eval**
8	Variability	Heterogeneity	Developers can connect new components written in commonly used programming languages (such as Java, C, and JavaScript) to the middleware in a matter of hours.	4,50
12	Functionality	System entity search	Developers can search for specific agents and messages with specific properties via queries and thus localise errors in the system.	4,33
4	Availability	Agent failure tolerance	Uncontrolled sending of messages from an agent does not affect the functioning of the system.	4,22
1	Performance	Message latency	Developers can use the system to implement user interaction with reasonable message latencies.	4,17
10	Functionality	Context processing	The system offers developers the possibility to subscribe to, edit, filter and forward messages from agents via queries.	4,17
7	Modifiability	Change message format	Developers can add or change message formats within a day.	4,06
9	Modularity	Reusability of system components	Administrators can exchange system components and use them independently of each other.	3,83
3	Availability	Node failure tolerance	System functions are available as long as more than half of all of the middleware nodes are available.	3,78
5	Modifiability	Add new technologies	Developers can connect additional technologies, like other messaging systems or previously unsupported but compatible protocols, to the middleware within a day.	3,72
14	Functionality	Test settings	Developers can generate test settings and test data with a descriptive language without having to write their own components.	3,72
15	Conceptual integrity	Agent-based	Developers can think about all components of the system as agents with dedicated tasks.	3,72
11	Functionality	System status visualisation	The system can show developers the current communication graph, filtered on demand, with all agents and groups.	3,67
13	Functionality	Ad-hoc queries	Developers can query the system state and messages of agents ad hoc via queries to interactively learn about the structure of a running system.	3,67
2	Performance	Scalability	The performance of the system scales with the addition of more middleware nodes up to at least 1,000 agents.	3,22
6	Modifiability	Descriptive programming	Developers can create simple full-featured agents via a descriptive language.	3,06

TABLE 8.2: Rating of the 15 scenarios from the quality attribute utility tree based on the first questionnaire from 5 (very important) to 1 (not important), sorted by importance.

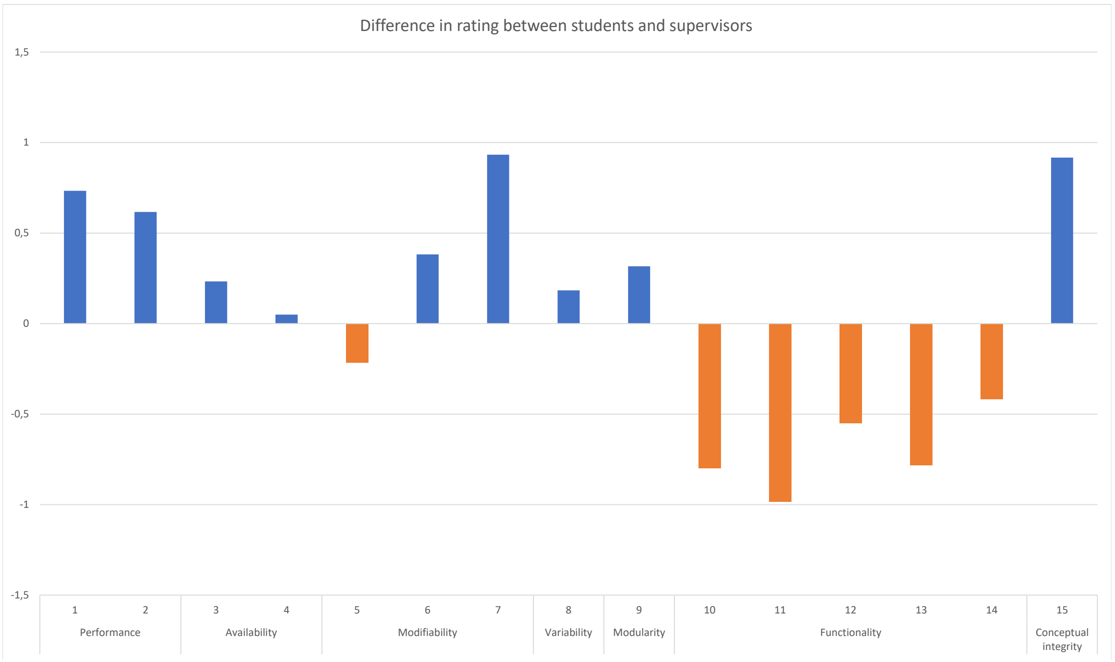


FIGURE 8.2: Differences in the average evaluation of the scenarios by students and staff. Positive values (in blue) show a higher average importance from the group of students negative values (in orange) from staff and professors.

8.1.4 ATAM Workshop

The ATAM workshop was conducted with five people. Besides the software architect and author of this thesis, four other people were present during the workshop. All identified stakeholders of the labs were represented, as at least one student, one staff member, and one professor were part of the workshop. All participants were or are part of the team of the laboratories studied here. This is common with ATAM, and it means they have worked with the architecture over a period of at least multiple semesters. To avoid conflicts of interest, participation in the workshop was entirely voluntary, and none of the participants was in any way dependent on the outcome of the workshop, for example through grades or something similar.

The workshop consisted of two appointments. In the first appointment an introduction to ATAM was given, the outputs of the workshop were explained, and the architecture was presented in detail. Then, evaluation of the architecture started and the process continued in the second workshop. In this way, all scenarios could be addressed. The exact procedure of the workshop follwed the ATAM described at the beginning of the chapter.

The further procedures of the workshop and the results are presented below.

8.1.5 Scenario Brainstorming

In the second part of the first questionnaire, the participants were asked to add further scenarios that they felt were missing from the first list. All scenarios were collected, merged where necessary, edited, and completed. This procedure was necessary because, firstly, some scenarios were given in German and had to be translated. Secondly, some scenarios were incomplete or only given in keywords, and thirdly, some scenarios were given by several participants. Based on the responses, a total of 11 further scenarios, numbered 16 to 26, were added.

With these 11 new scenarios from the first questionnaire, a second questionnaire was created and completed by all participants in the workshop. The questionnaire can be found in the Appendix in Section B.2. All new scenarios, sorted by their scores, are listed in Figure 8.3.

As a result of the first survey, additional functionality scenarios have been added that describe different use cases from the stakeholders' point of view. These include scenarios in which agent interfaces are examined and scenarios that describe the work of and with students. In addition, various scenarios were mentioned that involve recording and replaying messages, such as to carry out tests. In addition, two new scenarios were added, namely message integrity and message prioritisation.

All but one of the new functionality scenarios were rated as high as important. Only the *realisation of projects with students* scenario was rated as moderately important. Message integrity and prioritisation were rated the lowest of all scenarios and were considered only slightly important.

	Category	**Refinement**	**Scenario**	**Eval**
24	Functionality	Display agent activation conditions	Developers can use the system to find out which interfaces an agent supports and which messages it expects, to be able to estimate which conditions (such as sensor data) the agent is waiting for.	4.40
19	Variability	Freedom in the choice of attributes	Developers can encode any JSON-compatible attributes in messages sent to and from agents. These attributes can be searched for using queries to track these messages in the system.	4.20
20	Functionality	Simple message tracing	The system offers developers the option to trace messages and their responses for debug purposes.	4.20
22	Functionality	Project handover to following semester groups	A new group of students who wants to adapt or replace parts of the system can use the user interface to see which components are involved and what dependencies they have on the rest of the system.	4.20
16	Functionality	Checking interfaces of agents	Developers can see in the user interface whether agents implement the correct interfaces, which ones are missing, and which ones are not supported by the current communication partners.	3.80
17	Testing	Message recording and replay	Developers can save the communication between agents in a part of the system and replay it later for testing purposes. The display and processing via queries are possible in the same way as for all messages. These messages are additionally marked with an attribute to enable filtering.	3.60
21	Functionality	Cooperation between student groups	Student groups in a semester, each working on their agents, can use the system to exchange information about interfaces and dependencies of their agents.	3.60
23	Functionality	Realisation of projects with students	Teachers can see via the user interface which agents are integrated into the system by their students and which other agents are potentially influenced by them. If necessary, the exact information exchange of these new agents can be analysed in detail.	3.40
18	Variability	Freedom in the choice of name	Developers can freely choose group names and, if necessary, define their own naming conventions.	2.80
26	Performance	Message prioritisation	Messages from agents can be prioritised. If possible, a message with a higher priority is forwarded by the middleware before a message with a lower priority.	2.40
25	Security	Message integrity	Developers can be sure that the messages are not altered by other users and have to possibility to validate the integrity.	2.00

Table 8.3: Rating of the 11 additional scenarios from the second questionnaire from 5 (very important) to 1 (not important), sorted by importance.

Overall, it can be seen that performance and security aspects are far behind functionalities that make it easier for developers to work with the system.

8.1.6 Architectural Approaches Analysis

Now that all scenarios have been determined and evaluated, the second part of the ATAM workshop continues with an analysis of whether the architecture to be investigated is suitable for all scenarios. During the remaining part of the ATAM workshop, all scenarios were analysed one after the other, and the appropriate architectural approaches were assigned to each of them. Each of these mappings was described with Sensitivity Points, tradeoff points, risks and nonrisks.

- **Sensitivity Points** - A sensitivity point is a property of one or more components or the relationship between components that is important for the satisfaction of a quality attribute described by a scenario [Clements et al., 2009]. Sensitivity points show which system parts are important for the fulfilment of a quality characteristic. They show which system parts are important for the fulfilment of a quality characteristic and serve as an indication of what must be observed if certain system parts are to be changed.
- **Tradeoff Points** - A tradeoff point is a sensitivity point that affects more than one quality attribute [Clements et al., 2009]. For example, this is the case when a change to the architecture improves one quality attribute but simultaneously worsens another quality attribute.
- **Risks** - Risks indicate problems in the architectural decisions with regard to the scenarios. Risks can come from architectural sources, such as certain components or interfaces, as well as from non-architectural sources, such as too little communication between certain stakeholders [Clements et al., 2009].
- **Nonrisks** - Nonrisks document good architectural decisions [Clements et al., 2009]. Conclusions are drawn from the architectural approaches as to whether the decisions are appropriate based on the scenarios.

During the analysis, further architectural approaches can be identified in accordance with the methodology used, if this seems appropriate to describe the scenarios. This manifested itself during the workshop as questions to the architects on how a scenario would be implemented in the architecture under investigation. These were often based on the information given during the presentation at the beginning of the workshop. If a scenario could not be answered with the list of approaches provided before the workshop, the list had to be extended. In the first part of the workshop, A9 (Distributed Middleware Nodes) and A10 (Layer Design) were added, and in the second part, A11 (Monitoring and Logging) was added to the list of architectural approaches in Section 8.1.1.

To document the process of the architectural approach analysis, a form was filled out for each scenario during the workshop. Keywords were used to describe the four properties. After the workshop, these forms were then transferred into a digital format. The numbering of the properties was corrected and standardised between the two workshop dates. The results are described in the following section and can be found in the appendix in Section B.3.

8.2 Results

In the following, the further results of the ATAM Workshop are presented. These consist of the sensitivity points, tradeoff points, risks and nonrisks of the mappings of cenarios and architectural approaches.

8.2.1 Sensitivity Points

Sensitivity points show which entities in the software architecture are particularly important for the realisation of the scenarios examined. A total of 13 sensitivity points were documented during the workshop.

Notably, the interface between agents, or the agent framework, and middleware, or the middleware API, is one of the most sensitive points of the architecture. The protocol support of this interface and the agent framework are directly related to the support of heterogeneity (S1, S2). In addition, it is important for the monitoring functions of the middleware that agents correctly implement the registration process in the interface (S8).

It was also highlighted during the workshop that the message latency in the system depends on both the number of messages and the number of middleware nodes (S5, S6), as was also confirmed in the latency and scalability measurements in this thesis. This results in recommendations for the administrators of middleware installations: at least three, preferably five nodes should be deployed so that there is fault tolerance and the system does not suffer any performance losses even under very high load.

The results also demonstrate that compliance with the design decisions in the system could have an impact on the scenarios. These include open design and agent-based design in particular. All components should be implemented in such a way that inspection is easily possible (S7) and agent-based design principles (S3) should be followed. Adherence to both design decisions makes it easier to understand the system and debug errors.

Finally, the integration of CEP functions was identified as a particularly sensitive part of the architecture (S4, S10). The CEP engine as a component is very important for the implementation of many of the scenarios, and the type of integration influences the comprehensibility of system processes.

8.2.2 Tradeoff Points

During the workshop, a total of 23 tradeoff points were identified. Tradeoff points show the dependencies of entities in the software architecture on two or more quality attributes. Compared to the sensitivity points, additional care must be taken with the entities highlighted here because potentially conflicting properties could be affected.

When analysing the trade-off points, it is noticeable that the possibility of sending CEP queries to the system has a positive influence on many other scenarios (T4, T6, T9, T19). These include the exchange of developers for handovers (T19), the induction of new team members, and the identification and localisation of errors in the system (T4). CEP queries can also support the monitoring, testing, and acceptance of student projects (T19), which is similar for the visualisation functions of the middleware. These support debugging scenarios and make it easier to find components and identify dependencies.

An important point is that scaling the middleware nodes can affect the performance and fault tolerance of the system. Administrators should pay special attention to the requirements of the system in the environment and configure the software accordingly. The number of agents can affect several other factors (T12), including the minimum number of middleware nodes (T11, T15), message latency, and the clarity of the visualisation (T14) of the system state. Although filtering functions can help improve the overview of the latter, a smaller system is often easier to understand.

One design decision that influences many scenarios is that of open design. The possibility of freely accessing groups and the decision to give developers as much freedom as possible (T3, T8) when programming components leads, on the one hand, to ready support for heterogeneous components. On the other hand, these freedoms can also lead to components being programmed incompatibly with architectural decisions (T23), which can negatively affect the maintainability and comprehensibility of the system.

8.2.3 Risks

The risks describe potential problems with certain properties of the architecture or external influences. A total of 29 risks were identified during the workshop. In Clements et al. [2009] it is recommended when presenting risks to group them into themes if possible. These themes are presented one after the other in the following.

Risks are not necessarily negative and, in some cases, cannot be avoided. It is important to keep an eye on risk factors to be able to take appropriate measures to mitigate or avoid potentially negative impacts.

The vast majority of the risks identified during the workshop are dependent on the system programmers. For example, key elements of the architecture, such as agent-based design (R14) or CEP queries (R8, R22), need to be understood by the programmers in order to be used correctly. In addition, it is important that all developers adhere to the design choices of the architecture as much as possible and do not intentionally violate them (R23). In this context, several risks were introduced based on the workshop participants' experiences with student projects. These include incidents where existing systems were deliberately ignored and "improved" new implementations were made, or the opinion that the system could also work differently as in another project, leading to components being integrated that do not fit with the rest of the system (R21).

Several points were raised to counteract these risks. On the one hand, all developers should learn the key elements of architecture in courses or workshops and refresh them regularly. On the other hand, new projects should be accompanied by lecturers or staff to ensure that they do not have a negative impact on the system and can integrate well in the future. The aim of this is to prevent deliberate violations of design decisions in the system to operate the system in the long term at a reasonable cost. Working with students in such laboratory environments is one of the challenges that has been incorporated into the design of the architecture. The functions of the architecture with CEP queries and visualisation can help here to quickly check whether certain preconditions are fulfilled, as is also represented in several of the scenarios studied. The open design of the architecture means that all freedom in implementation is in the hands of the programmer. This has the advantage that the architecture is very flexible and easy to inspect, but it also facilitates the violation of central design principles.

Another group of risks arises from the complexity (e.g., R7, R10) and distribution of components in the system across the network (R20, R29), which create concurrency and can lead to errors that are difficult to identify. These risks cannot be completely avoided because of the environment and smart systems. Still, the architecture can mitigate the complexity and contribute to error prevention and it is therefore important to identify the parts of the architecture that are particularly critical here. During the workshop, the guarantees for message sequences were mentioned in this context. These can potentially be affected by a failure of the middleware nodes, for example.

A third area of risk is that individual components in the system can affect the performance of the overall system. Attention should be paid to the fact that the connection of further message systems or fully implemented components can have an influence on the behaviour and performance of the overall system (R11). Furthermore, the hardware used is also a potential cause of problems, such as if the network hardware is too slow (R7). In addition, errors in individual components can also have negative effects, for example if a component floods the system with messages or actively sends faulty messages (R6). In both cases, the visualisation of the middleware can help to estimate which parts of the system are potentially affected by new components.

8.2.4 Nonrisks

Nonrisks describe good architectural choices for the scenarios studied. Documenting nonrisks can therefore help to record these decisions so that they can be taken into account in any subsequent changes. Furthermore, nonrisks can provide information about how suitable the architecture is for the implementation of the scenarios. A total of 51 nonrisks were identified during the workshop, making nonrisks the most frequently described feature during the workshop.

The nonrisks are not only the most frequent properties in the workshop results, but all scenarios were also assigned at least one nonrisk, indicating that the scenarios are supported by at least one architecture decision. In the following paragraphs, the most important nonrisks are presented in the order of the architectural approaches. A list of all nonrisks can be found in the Appendix in Section B.3.5.

The **agent-based design (A1)** of the architecture ensures, among other things, that each component has its own separate task and only communicates with other components via messages. This supports program comprehension and debugging tasks because all communication between components is accessible for inspection (N3). As long as this is respected, it is also easier to find the component responsible for a particular task and to see which other components may be affected by it (N24). This can be helpful when training new team members, like new student groups (N38).

The **open design (A2)** principle of the architecture allows easy access to all messages in the system. This makes it possible to read all messages (N9, N33) as well as to publish additional messages everywhere (N19).

The **publish/subscribe layer (A3)** of the middleware ensures that all messages in the system can be subscribed to (N11) by all components at any time. In addition, external systems can be connected via this layer as long as they send JSON-based messages (N11). By distributing the messages to different groups, the load on the messaging layer can be distributed to several middleware nodes (N5).

The **open middleware API (A4)** ensures the easy connection of new components with a wide range of technologies and protocols (N1), handles the handover process in case of node failure (N15) and ensures that all agents adhere to the messaging protocols (N41).

The **agent framework (A5)** accelerates the development of new components and, among other things, manages the connection of new components to the middleware (N17).

The **API libraries (A6)** ensure that it is always known which interfaces an agent implements. This allows the middleware to check whether agents communicate with compatible messages (N40). It can manage multiple versions of interfaces (N12). In addition, the API promotes documentation about the given DSL and makes it possible to query it via the middleware (N43).

The **runtime environments (A7)** that can be controlled by the middleware ensure that agents can be migrated between different environments. This can increase availability in the event of a failure and possibly reduce latency if agents can be run closer to their communication partners.

The **integration of CEP (A8)** positively influences the largest number of scenarios of all architectural approaches. It enables filtered access to all messages (N2, N10) and thus the search for error messages (N4) and the checking of expectations (N44). In addition, CEP queries can be used to generate time-controlled messages, which can be used, for example, to play back test data and scenarios (N10). In addition, the visualisation can be filtered with CEP queries to increase clarity in larger systems (N25). Furthermore, the type of CEP integration ensures that all components continue to appear as agents in the visualisation, which increases the comprehensibility of the system (N39).

The distribution of the **middleware nodes (A9)** is the basis for the system's failure tolerance and scalability. By adding more middleware nodes, it is possible to distribute the load in the messaging layer to several nodes (N28), which can have a positive influence on message latency (N7). Since the system remains available as long as more than half of all nodes are reachable, the use of multiple nodes provides fault tolerance (N14).

The **layer design (A10)** of the architecture ensures that individual parts of the middleware can be exchanged (N13). This can help integrate the system into other environments or connect existing software with the middleware. In addition, special requirements can be met that are not provided for in the current implementation, such as prioritising individual messages without affecting the other layers (N50).

The **monitoring and logging (A11)** components of the middleware ensure that the state of agents and their message traffic can be inspected (N13). In addition, these components make it possible to record message traffic in parts of the system if desired and replay it at a later time (N50).

In summary, it can be stated that all architectural approaches are at least necessary for the implementation of at least one of the scenarios. Hence, the architectural approaches and thus the architecture fit the scenarios of the stakeholders and thus implement the necessary quality features.

8.3 Conclusion

A full scenario-based architecture analysis using the Architecture Trade-off Analysis Method (ATAM) was carried out and produced several outputs: a prioritised list of scenarios and quality attributes, a list of architectural approaches, the mapping of approaches to quality attributes and the associated sensitivity points, tradeoff points, risks and nonrisks. The ATAM workshop additionally contributed to the improvement of the presentation and documentation of the architecture and improved the understanding of the details of the architecture of all stakeholders involved. This is evident from the feedback of the workshop participants.

By collecting scenarios based on the literature and from all stakeholders and evaluating them via the questionnaires, it was shown that the important quality attributes were taken into account when evaluating the architecture. The assessment of the importance of the scenarios support that the functionality quality attributes have a high importance in the studied laboratory context compared to other attributes. The orientation of the architecture presented here therefore fits the needs of the environments studied. It can be assumed that these results are transferable to other similar laboratory environments. The expert interviews demonstrate that similar requirements and problems prevail in these other environments.

The analysis of the nonrisks shows that the architecture examined here is suitable for the scenarios and thus fulfils the desired quality characteristics and requirements from the literature and the surveyed stakeholders. It also indicates that all architectural approaches are important for the fulfilment of the scenarios.

The identified tradeoff points and risks do not show any major problems in the use of the architecture. However, they show that attention should be paid to the correct configuration of the system, such as the number of nodes, the correct training of the programmers, and comprehensive documentation.

Another aspect of the analysis of the identified risks is that in the operation of the laboratory environments for smart systems studied here, it is important to train all stakeholders and especially students in the technologies and paradigms used for the architecture and, if necessary, to check new components for compatibility with the existing architecture before they are integrated in the long term. This is necessary to ensure long-term operation, to keep maintenance low, and to keep the system understandable for new team members. In this regard, the results of the expert interviews indicate that this conclusion con be transferred to other environments

In contrast to the studies conducted in publications with comparable architectures (see Section 2.3.4), the procedure according to ATAM can be repeated, reviewed, and used for comparison with other architectures without additional information. All the steps required to perform an ATAM workshop to evaluate the architecture published here can be found in this thesis or in the literature cited. As such, the workshop can be repeated with other participants in the future. It is possible to include another architecture during this workshop to compare it with the one presented here.

Quantitative simulations and measurements are often implementation-dependent and cannot be verified or compared to other approaches without publishing the implementation. In addition, only performance and similar attributes can be tested, an evaluation of usability does not make sense with simulations because it is about interaction with humans. The latter is often evaluated in comparable works with case studies, but this evaluation is not structured and does not follow any published method. Therefore, it is difficult to verify the evaluation or reproduce the used application example to compare it with new approaches. Furthermore, a case study as conducted in comparable publications does not involve stakeholders or persons other than the authors of the paper, which can limit the validity of the research.

The scenario-based evaluation was conducted to answer research questions *Q3* and *Q4*. With respect to *Q3*, the results of the scenario-based evaluation show that the presented seamless CEP integration meets the identified requirements and is therefore suitable for this and probably similar architectures. Furthermore, the results of this scenario-based evaluation together with the expert interview results show, related to *Q4* that the developed architecture is suitable to support developers in laboratory environments with program comprehension and debugging tasks.

With the scenario-based analysis and the experimental evaluation, all identified requirements for the software architecture have now been investigated. In the following conclusion of the thesis, an overview of the evaluation of the requirements is given.

Chapter 9

Conclusions and Future Work

This chapter summarises the results of this work, the contributions, and implications for future research.

9.1 Summary

This thesis presented a software architecture for smart systems that supports the development of software and fast experiments in research laboratories. The main component of this architecture is a middleware, which provides publish/subscribe communication. The middleware contains a framework that facilitates the development of new components as well as supports the inspection of the system by collecting and processing status information about all components, for example about the active agents and groups. A seamless integration of CEP queries allows for processing of context information as well as exploring the state of the system. In this way, it is possible to perform debugging and program comprehension tasks on the running system.

Based on this and the research objectives stated in the introduction, the following four research questions (*Q1*-*Q4*) were formulated.

Method	Expert Interviews	Latency and Scalability Measurements	Scenario-based Architecture Analysis
Questions	Q1 / Q4	Q2	Q3 / Q4
Methodological Choices	qualitative interview study	quantitative experiment	qualitative scenario-based analysis
Reference	Chapter 6	Chapter 7	Chapter 8
Results	The presented architecture is suitable for different use cases in smart system laboratories, which have additional requirements compared to normal smart systems due to the research context.	The measurements show low latencies and a high scalability suitable for intended environments and for user interaction.	The scenario-based analysis shows that the architecture implements the necessary requirements and is well suited for use in smart systems research laboratories.

TABLE 9.1: Overview of the studies conducted in this thesis

- **Q1**: What are the requirements for a software architecture in research labs, that uses loosely coupled distributed applications in smart systems?
- **Q2**: How is it possible to create a software architecture to support development tasks in research labs without compromising the performance of the system?
- **Q3**: How can CEP be integrated into a middleware for smart systems without increasing the complexity of the system for developers?
- **Q4**: How can such a middleware with an integrated CEP engine support program comprehension and debugging tasks in smart systems?

To investigate these research questions, three separate studies were conducted. An overview of these three studies is provided in Table 9.1.

Firstly, expert interviews (see Chapter 6) were conducted to analyse the requirements for software in smart system laboratory environments (*Q1*) and to evaluate the transferability of the CEP tools presented (*Q4*) in this thesis. Based on the analysis of the expert interviews, further requirements were formulated that apply specifically to software in smart systems laboratory environments. These include a strong requirement for support of the software development process. The reasons for this are, changing team compositions, working with students, short project durations, and problems with the documentation. The results of the interviews also support that the architecture presented here can also be used in other laboratory environments with loosely coupled systems, and the experts see use cases for the systems they are working with (*Q4*), incuding the evaluation of system states, the evaluation of sensor data, the creation of test settings, the automatic detection of errors, and the targeted recording of events. All these use cases help to support development tasks in different phases of software development.

Secondly, the performance of the middleware presented here was evaluated to ensure that the additional logging and system state analysis of the software, which serve to support development processes, do not slow down the system. The latency measurements based on the middleware show that the architecture presented here can handle a very large number of agents and high message rates (see Section 7.2). Tests were conducted with numbers of up to 18,000 agents, which is an extremely large number compared to most of the systems analysed in the literature and all comparable environments mentioned during the expert interviews. In addition, the scalability of the middleware was tested with two to seven middleware nodes and 10,000 agents (see Section 7.3). This answers *Q2* and shows that the middleware is scalable and that by adding more middleware nodes, the load on the system can be reduced, thereby reducing latency. A measurement with 10,000 agents showed an average message latency of 4.5 milliseconds with seven middleware nodes. This is more than sufficient for the intended use cases. Based on this, the scalability of CEP integration in Section 7.4 was confirmed. Since the CEP layer was implemented entirely based on the messaging layer, it shares its performance characteristics.

Finally, in Chapter 8 the ATAM illustrated that the presented architecture is well suited for the intended use cases in smart systems and supports the development of new components, debugging of errors, and program comprehension. Hence, with regard to *Q4*, the frameworks and tools developed in this thesis have to possibility of improving the development process of smart systems. This is also supported by the two case studies presented in Section 4.4.4. The results of the ATAM workshop also indicate that a integration of a CEP engine is possible without hindering the

Requirement	Scenarios	Architectural Approaches
Scalability and latency	1, 2, 26	Middleware and framework are multi-threaded and use actor programming. Tested by the latency and scalability measurements
Support for heterogeneity	8	Open middleware API that supports various network protocols
Support for mobility	15	Agent-based architecture with event sourcing and runtime environments
Tolerance for component failures	3, 4	Middleware nodes and, optionally, agents are redundant
Support for debugging	10, 11, 13, 14, 16, 17, 20, 24	Open design, CEP queries, middleware web interface
Program comprehension support	10, 11, 12, 13, 24	Open design, CEP queries, middleware web interface
Support for fast experiments	6, 13, 14	Fast creation of agents through CEP queries and the frameworks
Deployment and configuration	13, 14, 15	Runtime environments and CEP queries allow dynamic configuration and deployment
Flexibility	5, 6, 7, 18, 19	The open design allows the integration of anything that can communicate via JSON messages
Support of communication between developers	21, 23	Agent group graph and dynamic result lists of CEP queries

TABLE 9.2: Overview of the evaluation of the requirements for the presented architecture

development processes by increasing complexity and that the work of developers can even be supported by using the CEP queries (*Q3*). Table 9.2 contains an overview of all requirements and the scenarios through which they were evaluated in this thesis.

9.2 Contributions

This PhD thesis provides three research contributions:

- **C1**: A smart system software architecture and middleware with low message latency and a high scalability designed for research laboratories (see Eichler et al. [2017]).

- **C2**: A seamless integration of CEP into an agent-based distributed system (see Eichler et al. [2020]).
- **C3**: Improvement of program comprehension and debugging in smart systems with CEP queries evaluated by expert interviews and a scenario-based evaluation.

Smart systems are an active research topic [Laghari et al., 2022, Fang, 2021, Santos et al., 2022]. One of the challenges continues to be how to manage the complexity of these systems due to various factors, including context dependency, heterogeneity of components, and the integration of sensors and actors. Tools are needed to help developers build and better understand these systems, as traditional software engineering methods face limits [Fang, 2021]. The literature review in this thesis revealed a lack of research on middleware, frameworks, and tools for smart systems research environments and architectures that focus on supporting developing tasks. The contribution *C1* is an architecture, including middleware, tailored to the needs of research environments in smart systems. This includes support for program comprehension and debugging through tools for analysing the state of the system at runtime. A prior version of the architecture, including a middleware and latency and scalability measurements, was published in Eichler et al. [2017].

Part of the architecture presented in this thesis is a seamlessly integrated CEP engine (*C2*), which allows for the processing of messages and context information and helps to analyse the system state. The advantage of this seamless integration over the approaches known in the literature (see Section 2.4.3) is that it allows developers to reason about the CEP engine as an agent-based system. There is no need to switch between two different paradigms, agent-based and event-based, during development tasks, which reduces complexity. The entire software architecture presented here, including the integration of the CEP engine, was evaluated with the ATAM (see Chapter 8). This new way of integrating a CEP engine into an agent-based system for smart systems, including an evaluation using the two case studies presented in this thesis, was published in Eichler et al. [2020].

The seamless integration of a CEP engine allows one to analyse the state of the system with the help of descriptive CEP queries. As the scenario-based evaluation shows, debugging and program comprehension tasks can be implemented with CEP queries during runtime. In this way, errors can be detected and localised, test scenarios prepared, and the state of the entire system analysed. This contributes to the open question of how the complexity of smart systems can be made manageable in the future. The approach of using CEP for development support (*C3*) was evaluated with the help of the ATAM. Furthermore, the expert interviews support that the approach can be transferred to other laboratory environments.

9.3 Limitations

The research design, methods and external conditions may impose limitations on the results.

Possible limitations arise from the research design and the choice of research methods. There might be limitations to the results of the expert interviews because, as usual for this method, they come from from a small number of participants. It cannot be ruled out that additional participants with different experiences in smart environments would have changed the results. They are always based only on the environments analysed during the evaluation. The qualitative evaluation is also affected by limitations because the coding frame, which was designed by the author, has a direct impact on the result. To mitigate this, the complete evaluation process was documented in this thesis to ensure reproducibility.

The measurements carried out may also be affected by limitations. The results of the latency and scalability measurements depend on the described test setup, such as the specific machines and network components. Influences on the results due to the use of virtualised machines and the specific hardware are not ruled out.

The development of the software architecture presented here took place in two research labs, the Living Place Hamburg and the CSTI. The case studies used for the evaluation and the workshop for the scenario-based analysis also took place in these labs. The availability of these laboratories was the primary selection criteria. In general, the research conducted here is aimed at research laboratories in the field of smart systems, but the choice of laboratory environment could have directly influenced the results, because the implementation was only tested in these environments. Based on the expert interviews, transferability to other environments is assumed but can not be ensured for all possible environments.

There are also limitations due to external influences and circumstances. The research conducted in this thesis was impacted by the restrictions imposed during the COVID-19 pandemic. Due to the temporary closure of the laboratories as a result of contact restrictions, no long-term studies could be carried out. The collected results from the interviews are based on the experiences of the participants at the time of the studies and the architecture presented during the interviews and the workshop. This limitation could be improved for the interviews through a future long-term study including practical sessions with the interviewees. However, this would have required a significantly longer time commitment and multiple trips for a face-to-face meeting in a research lab for the demonstrations.

Based on the literature review in Chapter 2 there is a lack of similar studies on the requirements of middleware for research labs in smart systems for fast experimentation and rapid prototyping. However, the literature review is dependent on the limitations of the literature analysis and particularly the procedure for the literature analysis and the selection criteria for comparable work. Future studies are needed to confirm the requirements, possibly complete them, and, if necessary, bring them into line with other requirements for smart systems in production environments.

Furthermore, limitations arise due to the delimitation of certain research topics during the literature search. Some topics were deliberately left out to distinguish this thesis from other works. Loosely coupled agent-based systems, especially with publish/subscribe-based messaging, which are prevalent in smart system architectures, are considered. Transferability to other types of systems, for example, based on tuple spaces, is not considered.

Security and privacy are important topics in the field of smart systems, because sensitive information and communication are often handled via unsecured networks. This thesis does not consider these topics; only basic security functions to protect data integrity on the network level are included, because the focus here is on rapid prototyping in a laboratory environment, as described in Chapter 3.2 during the requirement analysis. This is a limitation for the transferability of the results to other environments that may depend on security and privacy requirements. There are already approaches in the literature regarding how to encrypt and authenticate multi-agent and publish/subscribe systems that could be applied to the architecture presented here [Foner, 1997, Sulaiman et al., 2009, Thangam and Chandrasekaran, 2016].

Debugging and program comprehension tasks are considered in this thesis at development time and runtime, and as well as with respect to distributed systems, where the complexity lies primarily in the interaction between the components. Verifications and proofs of the correctness of programmes are are not always possible in this area and not considered here because they conflict with the goal of allowing fast experimentation.

9.4 Research Implications

The results of the expert interviews support that there are further requirements for the software used in research environments in the field of smart systems. These include requirements that arise from the research environment and team composition. Rapidly changing projects and teams with high turnover make software reusability and knowledge transfer more difficult, which should be taken into account in research environments. The results of this thesis show that one way to address these additional requirements is to use a middleware, which should be easily inspectable and offer frameworks for development support and tools for easy debugging.

The architecture developed in this thesis aims to enable rapid experimentation in a research context to help future research projects and research laboratories to explore topics in the field of smart systems. Furthermore, the architecture demonstrates that it is possible to seamlessly integrate the functionalities of CEP engines into an agent-based middleware. This should be considered and further investigated in future research. The advantages in terms of system comprehensibility and complexity could also be useful in future approaches.

Finally, this thesis shows the possibility of reproducibly evaluating architectures in the field of smart systems. The use of a scenario-based analysis method allows the evaluation of performance requirements as well as usability aspects and interactions between these requirements. The description of the methods used in this thesis, including adaptations for evaluation over a longer period of time in a research context, could contribute to improving future research in this area.

9.5 Future Research

It is planned to extend the architecture presented in this thesis with further functions in the future to continue to improve the development process in smart systems. To this end, more information about the system and the agents will be made available to the CEP engine. For example, this could be more meta-information about individual agents and groups as well as historical data about the system. For this, the CEP engine would have to be extended by a persistent database, and it would have to be investigated how this could be accessed via CEP queries without impairing the existing functions. In addition, the question is open as to how this integration could be implemented without negatively affecting overall scalability.

Since this thesis is primarily concerned with the development of software in laboratory environments, security was not a focus of this work. The architecture presented in this thesis and the CEP query approaches were designed without security or privacy requirements, as mentioned in Chapter 3.2 during the requirement analysis. However, the communication protocols between the agents and the middleware were designed to be interchangeable and to allow different protocols to be used simultaneously by different agents, which could help during the implementation of message encryption. Publish/Subscribe systems are widely used, and there are approaches for encryption and authentication in these systems. A unification of these approaches with an architecture designed to support the development process, as in this thesis, can be the subject of future research.

To better assess the impact of smart systems middleware with development support and debugging tools on projects in general and research projects in particular, long-term studies are needed to accompany these projects and the teams over a longer period of time. The approaches presented here were also used in teaching to develop projects during several semester courses with students. An evaluation of the impact on teaching in this area is planned for the future.

In addition, the approaches presented in this thesis for using CEP queries for program comprehension and debugging could be transferred to other environments in the field of smart systems or even to other distributed, loosely coupled systems in the future.

Bibliography

Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing // Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, HUC '99, pages 304–307, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-66550-1. URL http://dl.acm.org/citation.cfm?id=647985.743843.

Adnan Akbar, Francois Carrez, Klaus Moessner, Juan Sancho, and Juan Rico. Context-aware stream processing for distributed iot applications. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*. IEEE, 2015. doi: 10.1109/wf-iot.2015.7389133.

Cesare Alippi, Stavros Ntalampiras, and Manuel Roveri. Model-free fault detection and isolation in large-scale cyber-physical systems. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 1(1):61–71, 2017. ISSN 2471-285X. doi: 10.1109/TETCI.2016.2641452.

Mahmoud Ammar, Giovanni Russello, and Bruno Crispo. Internet of things: A survey on the security of iot frameworks. *Journal of Information Security and Applications*, 38:8–27, 2018. ISSN 22142126. doi: 10.1016/j.jisa.2017.11.002.

Darko Anicic, Paul Fodor, Sebastian Rudolph, Roland Stühmer, Nenad Stojanovic, and Rudi Studer. A rule-based language for complex event processing and reasoning. In Pascal Hitzler and Thomas Lukasiewicz, editors, *Web reasoning* *and rule systems*, volume 6333 of *Lecture notes in computer science*, pages 42–57. Springer, Berlin and Heidelberg, 2010. ISBN 9783642159176. doi: 10.1007/978-3-642-15918-3_5.

Ioannis Arapakis, Xiao Bai, and B. Barla Cambazoglu. Impact of response latency on user behavior in web search. In Shlomo Geva, editor, *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, ACM Digital Library, pages 103–112, New York, NY, 2014. ACM. ISBN 9781450322577. doi: 10.1145/2600428.2609627.

Ross D. Arnold and Jon P. Wade. A definition of systems thinking: A systems approach. *Procedia Computer Science*, 44:669–678, 2015. ISSN 18770509. doi: 10.1016/j.procs.2015.03.050.

Kevin Ashton. That 'internet of things' thing. *RFID journal*, 22(7):97–114, 2009.

Juan C. Augusto and Chris D. Nugent. The use of temporal reasoning and management of complex events in smart homes. *Proceedings of the 16th European Conference on Artificial Intelligence*, 2004. doi: 10.5555/3000001.3000165.

Juan Carlos Augusto and Chris D. Nugent, editors. *Designing smart homes: The role of artificial intelligence*, volume 4008 of *Lecture notes in computer science.* Springer and Springer-Verlag, Berlin and New York // Berlin, Heidelberg, 2006. ISBN 354035994X // 3-540-35994-X.

M. A. Babar and I. Gorton. Comparison of scenario-based software architecture evaluation methods. In *Proceedings / 11th Asia-Pacific Software Engineering Conference*, pages 600–607, Los Alamitos, Calif., 2004. IEEE Computer Society. ISBN 0-7695-2245-9. doi: 10.1109/APSEC.2004.38.

M. A. Babar, L. Zhu, and R. Jeffery. A framework for classifying and comparing software architecture evaluation methods. In Paul Strooper, editor, *Proceedings / 2004 Australian Software Engineering Conference, ASWEC 2004, 13 - 16 April 2004, Melbourne, Australia*, pages 309–318, Los Alamitos, Calif., 2004. IEEE Computer Society. ISBN 0-7695-2089-8. doi: 10.1109/ASWEC.2004.1290484.

Sebastian Bader, Gernot Ruscher, and Thomas Kirste. A middleware for rapid prototyping smart environments. In Jakob E. Bardram, editor, *Proceedings of the 12th ACM international conference adjunct papers on Ubiquitous computing - Adjunct*, page 355, New York, NY, 2010. ACM. ISBN 9781450302838. doi: 10.1145/1864431.1864433.

Sarah Elsie Baker and Rosalind Edwards. How many qualitative interviews is enough? expert voices and early career reflections on sampling and cases in qualitative research. *National Centre for Research Methods*, 2012.

R. Baldoni, M. Contenti, S. T. Piergiovanni, and A. Virgillito. Modeling publish/subscribe communication systems: towards a formal approach. In *Proceedings / the Eighth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 304–311, Los Alamitos, Calif., 2003. IEEE Computer Society. ISBN 0-7695-1929-6. doi: 10.1109/WORDS.2003.1218097.

Simonetta Balsamo and Moreno Marzolla. A simulation-based approach to software performance modeling. *SIGSOFT Softw. Eng. Notes*, 28(5):363–366, 2003. ISSN 0163-5948. doi: 10.1145/949952.940122.

Luciano Baresi and Carlo Ghezzi. The disappearing boundary between development-time and run-time. In Gruia-Catalin Roman, editor, *Proceedings of the FSESDP workshop on Future of software engineering research*, New York, NY, 2010. ACM. ISBN 9781450304276. doi: 10.1145/1882362.1882367.

John Bates, Jean Bacon, Ken Moody, and Mark Spiteri. Using events for the scalable federation of heterogeneous components. In Paulo Guedes and Jean Bacon, editors, *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications - EW 8*, pages 58–65, New York, New York, USA, 1998. ACM Press. doi: 10.1145/319195.319205.

Joseph Bates, A. Bryan Loyall, and W. Scott Reilly. An architecture for action, emotion, and social behavior. In Cristiano Castelfranchi, editor, *Artificial social systems*, volume 830 of *Lecture notes in computer science Lecture notes in* *artificial intelligence*, pages 55–68. Springer, Berlin and Heidelberg, 1994. ISBN 978-3-540-58266-3. doi: 10.1007/3-540-58266-5_4.

Samuel Beck, Sebastian Frank, Alireza Hakamian, and André van Hoorn. How is transient behavior addressed in practice? In Dan Feng, editor, *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*, ACM Digital Library, pages 105–112, New York,NY,United States, 2022. Association for Computing Machinery. ISBN 9781450391597. doi: 10.1145/3491204.3527483.

Jonathan Becker, Uli Meyer, Tobias Eichler, and Susanne Draheim. A supernatural vr environment for spatial user rotation. In *2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, pages 850–851, 2019. doi: 10.1109/VR.2019.8798290.

Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. Developing multi–agent systems with a fipa–compliant agent framework. *Software: Practice and Experience*, 31(2):103–128, 2001. ISSN 00380644. doi: 10.1002/1097-024X(200102)31:2<103::AID-SPE358>3.0.CO;2-O.

Melvin Bender, Erkin Kirdan, Marc-Oliver Pahl, and Georg Carle. Open-source mqtt evaluation. In *CCNC 2021: 2021 IEEE 18th Annual Consumer Communications & Networking Conference (CCNC)*, pages 1–4, Piscataway, NJ, 2021. IEEE. ISBN 978-1-7281-9794-4. doi: 10.1109/CCNC49032.2021.9369499.

Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D. Ernst. Debugging distributed systems. *Communications of the ACM*, 59(8):32–37, 2016. ISSN 0001-0782. doi: 10.1145/2909480.

Claudio Bettini, Oliver Brdiczka, Karen Henricksen, Jadwiga Indulska, Daniela Nicklas, Anand Ranganathan, and Daniele Riboni. A survey of context modelling and reasoning techniques. *Pervasive Mob. Comput.*, 6(2):161–180, 2010. ISSN 1574-1192. doi: 10.1016/j.pmcj.2009.06.002.

Anol Bhattacherjee. *Social Science Research: Principles, Methods, and Practices.* Textbooks Collection. 3, 2012.

J. Boardman and B. Sauser. System of systems - the meaning of of. In *Proceedings / 2006 IEEE/SMC International Conference on System of Systems Engineering*, pages 118–123, Piscataway, NJ, 2006. IEEE Operations Center. ISBN 1-4244-0188-7. doi: 10.1109/SYSOSE.2006.1652284.

Alexander Bogner, Beate Littig, and Wolfgang Menz, editors. *Interviewing experts*. Research methods series. Palgrave Macmillan, Basingstoke England and New York, 2009. ISBN 978-0-230-24427-6. doi: 10.1057/9780230244276.

Pedro Victor Borges, Chantal Taconet, Sophie Chabridon, Denis Conan, Everton Cavalcante, and Thais Batista. Taming internet of things application development with the iotvar middleware. *ACM Transactions on Internet Technology*, 23(2):1–21, 2023. ISSN 1533-5399. doi: 10.1145/3586010.

J. Bosch and P. Molin. Software architecture design: evaluation and transformation. In *Proceedings / ECBS '99, IEEE Conference and Workshop Engineering of Computer-Based Systems, March 7 - 12, 1999, Nashville, Tennessee*, pages 4–10, Los Alamitos, Calif., 1999. IEEE Computer Society. ISBN 0-7695-0028-5. doi: 10.1109/ECBS.1999.755855.

Tibor Bosse, Dung N. Lam, and K. Suzanne Barber. Tools for analyzing intelligent agent systems. *Web Intelligence and Agent Systems: An International Journal*, 6(4):355–371, 2008. ISSN 15701263. doi: 10.3233/WIA-2008-0145.

Pyrros Bratskas, Nearchos Paspallis, and George A. Papadopoulos. An evaluation of the state of the art in context-aware architectures. In Chris Barry, Michael Lang, Wita Wojtkowski, Kieran Conboy, and Gregory Wojtkowski, editors, *Information Systems Development*, pages 1117–1128. Springer US and Springer, Boston, MA, 2009. ISBN 978-0-387-78577-6. doi: 10.1007/978-0-387-78578-3_42.

Melanie Brinkschulte, Christian Becker, and Christian Krupitzer. Towards a qos-aware cyber physical networking middleware architecture. In Unknown, editor, *Proceedings of the 1st International Workshop on Middleware for Lightweight,* *Spontaneous Environments - MISE '19*, pages 7–12, New York, New York, USA, 2019. ACM Press. ISBN 9781450370349. doi: 10.1145/3366616.3368149.

Jessica Broscheit. Livingplace gallery, 2022. URL https://livingplace.haw-hamburg.de/gallery/. 2022-06-26.

Christian Bühler. Ambient intelligence in working environments. In *International Conference on Universal Access in Human-Computer Interaction*, pages 143–149. Springer, Berlin, Heidelberg, 2009. doi: 10.1007/978-3-642-02710-9_17. URL https://link.springer.com/chapter/10.1007/978-3-642-02710-9_17.

C. Y. Chen, J. H. Fu, T. Sung, P. F. Wang, E. Jou, and M. W. Feng. Complex event processing for the internet of things and its applications. In *2014 IEEE International Conference on Automation Science and Engineering (CASE)*, pages 1144–1149, 2014.

Hong Chen. Applications of cyber-physical system: A literature review. *Journal of Industrial Integration and Management*, 02(03):1750012, 2017. ISSN 2424-8622. doi: 10.1142/S2424862217500129.

E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 7(1):13–17, 1990. ISSN 07407459. doi: 10.1109/52.43044.

Mark Claypool. The effect of latency on user performance in real-time strategy games. *Computer Networks*, 49(1):52–70, 2005. ISSN 13891286. doi: 10.1016/j.comnet.2005.04.008.

Paul Clements, Rick Kazman, and Mark Klein. *Evaluating software architectures: Methods and case studies*. SEI series in software engineering. Addison-Wesley, Boston, 8. printing edition, 2009. ISBN 9780201704822.

I. Cobeanu and V. Comnac. Embedding of event processing into multi-agent systems decision mechanism. In *2011 6th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI)*, pages 105–109, 2011. doi: 10.1109/SACI.2011.5872981.

Michael H. Coen et al. Design principles for intelligent environments. In *AAAI/IAAI*, pages 547–554, 1998.

Diane Cook and Sajal Das. *Smart Environments: Technology, Protocols and Applications (Wiley Series on Parallel and Distributed Computing) // Smart Environments: Technologies, Protocols, and Applications*. Wiley-interscience series in discrete mathematics and optimization. Wiley-Interscience and s.n, New York, NY, USA, 1st ed. edition, 2004. ISBN 978-0-471-68659-0. doi: 10.1002/047168659X.

Diane J. Cook. Multi-agent smart environments. *J. Ambient Intell. Smart Environ.*, 1(1):51–55, 2009. ISSN 1876-1364.

Diane J. Cook and Sajal K. Das. How smart are our environments? an updated look at the state of the art. *Pervasive Mob. Comput.*, 3(2):53–73, 2007. ISSN 1574-1192. doi: 10.1016/j.pmcj.2006.12.001.

B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009. ISSN 0098-5589. doi: 10.1109/TSE.2009.28.

Ricardo Costa, Davide Carneiro, Paulo Novais, Luís Lima, José Machado, Alberto Marques, and José Neves. Ambient assisted living. In Juan M. Corchado, Dante I. Tapia, and José Bravo, editors, *3rd Symposium of Ubiquitous Computing and Ambient Intelligence 2008*, pages 86–94, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-540-85867-6.

CSTI. Csti website. URL https://csti.haw-hamburg.de. 2022-06-26.

Gianpaolo Cugola and Alessandro Margara. Deployment strategies for distributed complex event processing. *Computing*, 95(2):129–156, 2013. ISSN 1436-5057. doi: 10.1007/s00607-012-0217-9.

Miyuru Dayarathna and Srinath Perera. Recent advancements in event processing. *ACM Computing Surveys*, 51(2):1–36, 2018. ISSN 0360-0300. doi: 10.1145/3170432.

Jonathan Deber, Ricardo Jota, Clifton Forlines, and Daniel Wigdor. How much faster is fast enough? In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 1827–1836. ACM, 2015. doi: 10.1145/2702123.2702300.

Ankita Deohate and Dinesh Rojatkar. Middleware challenges and platform for iot-a survey. In *Proceedings of the 5th International Conference on Trends in Electronics and Informatics (ICOEI 2021)*, pages 463–467, Piscataway, NJ, 2021. IEEE. ISBN 978-1-6654-1571-2. doi: 10.1109/ICOEI51242.2021.9452923.

L. Dobrica and E. Niemela. A survey on software architecture analysis methods. *IEEE Transactions on Software Engineering*, 28(7):638–653, 2002. ISSN 0098-5589. doi: 10.1109/TSE.2002.1019479.

Ali Dorri, Salil S. Kanhere, and Raja Jurdak. Multi-agent systems: A survey. *IEEE Access*, 6:28573–28593, 2018. doi: 10.1109/ACCESS.2018.2831228.

Jürgen Dunkel. On complex event processing for sensor networks. In *2009 International Symposium on Autonomous Decentralized Systems*, pages 1–6, 2009. doi: 10.1109/ISADS.2009.5207376.

Jürgen Dunkel. Towards a multiagent-based software architecture for sensor networks. In *2011 Tenth International Symposium on Autonomous Decentralized Systems*, pages 441–448, 2011. doi: 10.1109/ISADS.2011.64.

Bruce Edmonds and Joanna J. Bryson. The insufficiency of formal design methods " the necessity of an experimental approach - for the understanding and control of complex mas. In *Proceedings of the Third International Joint Conference on*

Autonomous Agents and Multiagent Systems - Volume 2 // Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS '04, pages 938–945, USA, 2004. IEEE Computer Society. ISBN 1581138644.

Tobias Eichler. *Agentenbasierte Middleware zur Entwicklerunterstützung in einem Smart-Home-Labor.* PhD thesis, HAW Hamburg, 2014.

Tobias Eichler, Susanne Draheim, Christos Grecos, Qi Wang, and Kai von Luck. Scalable context-aware development infrastructure for interactive systems in smart environments. In *2017 IEEE 13th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 147–150, 2017. doi: 10.1109/WiMOB.2017.8115848.

Tobias Eichler, Susanne Draheim, Kai von Luck, Christos Grecos, and Qi Wang. Integration of complex event processing into multi-agent systems: two use cases for distributed software development support. *Thirteenth International Tools and Methods of Competitive Engineering Symposium*, 2020.

R. Elchaama, B. Dafflon, R. K. Chamoun, and Y. Ouzrout. Toward a traffic regulation based on event processing agent system. In *2017 International Conference on Engineering, Technology and Innovation (ICE/ITMC)*, pages 1350–1356, 2017. doi: 10.1109/ICE.2017.8280038.

Abdessalam Elhabbash, Vatsala Nundloll, Yehia Elkhatib, Gordon S. Blair, and Vicent Sanz Marco. An ontological architecture for principled and automated system of systems composition. In Shinichi Honiden, Elisabetta Di Nitto, and Radu Calinescu, editors, *Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 85–95, New York, NY, USA, 2020. ACM. ISBN 9781450379625. doi: 10.1145/3387939.3391602.

Abdessalam Elhabbash, Yehia Elkhatib, Georgios Bouloukakis, and Maria Salama. A middleware for automatic composition and mediation in iot systems. In Evangelos Niforatos, Gerd Kortuem, Nirvana Meratnia, Josh Siegel, and Florian

Michahelles, editors, *Proceedings of the 12th International Conference on the Internet of Things*, pages 127–134, New York, NY, USA, 2022. ACM. ISBN 9781450396653. doi: 10.1145/3567445.3567451.

Lee D. Erman, Frederick Hayes-Roth, Victor R. Lesser, and D. Raj Reddy. The hearsay-ii speech-understanding system: Integrating knowledge to resolve uncertainty. *ACM Computing Surveys*, 12(2):213–253, 1980. ISSN 0360-0300. doi: 10.1145/356810.356816.

EsperTech. Esper - espertech: http://www.espertech.com/esper/, 2021. URL `http://www.espertech.com/esper/`.

Zhemei Fang. System-of-systems architecture selection: A survey of issues, methods, and opportunities. *IEEE Systems Journal*, pages 1–12, 2021. ISSN 2373-7816. doi: 10.1109/jsyst.2021.3119294.

Robert Farrow, Francisco Iniesto, Martin Weller, and Rebecca Pitt. *GO-GN Research Methods Handbook*. Open Education Research Hub, 2020. URL `https://go-gn.net/gogn_outputs/research-methods-handbook/`.

Ludger Fiege, Gero Mühl, and FELIX C. GÄRTNER. Modular event-based systems. *The Knowledge Engineering Review*, 17(4):359–388, 2002. ISSN 0269-8889. doi: 10.1017/S0269888903000559.

David Flater. Debugging agent interactions: A case study. In G. B. Lamont, editor, *Proceedings of the 2001 ACM symposium on Applied computing - SAC '01 // Proceedings of the 2001 ACM symposium on Applied computing*, pages 107–114, New York, New York, USA, 2001. ACM Press and ACM. ISBN 1581132875. doi: 10.1145/372202.372288.

Leonard N. Foner. Yenta: A multi-agent, referral-based matchmaking system. In W. Lewis Johnson, editor, *Proceedings of the first international conference on Autonomous agents*, ACM Conferences, pages 301–307, New York, NY, 1997. ACM. ISBN 0897918770. doi: 10.1145/267658.267732.

Giancarlo Fortino, Antonio Guerrieri, Michelangelo Lacopo, Matteo Lucia, and Wilma Russo. An agent-based middleware for cooperating smart objects. In Juan Manuel Corchado, editor, *Highlights on Practical Applications of Agents and Multi-Agent Systems // Highlights on practical applications of agents and multi-agent systems*, Communications in Computer and Information Science, pages 387–398. Springer, Berlin, 2013. ISBN 978-3-642-38061-7.

Giancarlo Fortino, Antonio Guerrieri, Wilma Russo, and Claudio Savaglio. Middlewares for smart objects and smart environments: Overview and comparison. In Giancarlo Fortino and Paolo Trunfio, editors, *Internet of Things Based on Smart Objects // Internet of things based on smart objects*, Internet of Things, pages 1–27. Springer International Publishing and Springer, Cham, 2014. ISBN 978-3-319-00490-7. doi: 10.1007/978-3-319-00491-4_1.

Giancarlo Fortino, Claudio Savaglio, Giandomenico Spezzano, and MengChu Zhou. Internet of things as system of systems: A review of methodologies, frameworks, platforms, and tools. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 51(1):223–236, 2021. ISSN 2168-2216. doi: 10.1109/TSMC.2020.3042898.

Cristian González García, Daniel Meana-Llorián, Juan Manuel Cueva Lovelle, et al. A review about smart objects, sensors, and actuators. *International Journal of Interactive Multimedia & Artificial Intelligence*, 4(3), 2017.

Alexandros Gazis and Eleftheria Katsiri. Middleware 101. *Queue*, 20(1):10–23, 2022. ISSN 1542-7730. doi: 10.1145/3526211.

Jochen Gläser and Grit Laudel. On interviewing “good” and “bad” experts. In Alexander Bogner, Beate Littig, and Wolfgang Menz, editors, *Interviewing experts*, Research methods series, pages 117–137. Palgrave Macmillan, Basingstoke England and New York, 2009. ISBN 978-0-230-24427-6. doi: 10.1057/9780230244276_6.

Pradyumna Gokhale, Omkar Bhat, and Sagar Bhat. Introduction to iot. *International Advanced Research Journal in Science, Engineering and Technology*, 5 (1):41–44, 2018.

Tao Gu, Hung Keng Pung, and Qing Da Zhang. Toward an osgi-based infrastructure for context-aware applications. *IEEE Pervasive Computing*, 3(4):66–74, 2004. ISSN 1536-1268. doi: 10.1109/MPRV.2004.19.

Tao Gu, Hung Keng Pung, and Qing Da Zhang. A service-oriented middleware for building context-aware services. *J. Netw. Comput. Appl.*, 28(1):1–18, 2005. ISSN 1084-8045. doi: 10.1016/j.jnca.2004.06.002.

Sylvain Hallé, Sébastien Gaboury, and Bruno Bouchard. Activity recognition through complex event processing: First findings. In *AAAI Workshop on artificial intelligence applied to assistive technologies and smart environments (ATSE-16). Association for the Advancement of Artificial Intelligence*, 2016.

Glenda Hannibal. Focusing on the vulnerabilities of robots through expert interviews for trust in human-robot interaction. In Cindy Bethel, editor, *Companion of the 2021 ACM/IEEE International Conference on Human-Robot Interaction*, ACM Digital Library, pages 288–293, New York,NY,United States, 2021. Association for Computing Machinery. ISBN 9781450382908. doi: 10.1145/3434074.3447178.

Stefan Haselbock, Rainer Weinreich, and Georg Buchgeher. An expert interview study on areas of microservice design. In *IEEE 11th International Conference on Service-Oriented Computing and Applications*, pages 137–144, Piscataway, NJ, 2018. IEEE. ISBN 978-1-5386-9133-5. doi: 10.1109/SOCA.2018.00028.

Klaus Havelund. Rule-based runtime verification revisited. *International Journal on Software Tools for Technology Transfer*, 17(2):143–170, 2015. ISSN 1433-2779. doi: 10.1007/s10009-014-0309-2.

K. Henricksen and J. Indulska. Modelling and using imperfect context information. In *Proceedings, Second IEEE Annual Conference on Pervasive Computing and Communications workshops*, pages 33–37, Los Alamitos, Calif., 2004.

IEEE Computer Society. ISBN 0-7695-2106-1. doi: 10.1109/PERCOMW.2004.1276901.

Karen Henricksen, Jadwiga Indulska, and Ted Mcfadden. Middleware for distributed context-aware systems. In *Proceedings of the 2005 Confederated international conference on On the Move to Meaningful Internet Systems - Volume / Part I*, pages 846–863, 2005.

C. Hewitt and H. G. Baker. Actors and continuous functionals. Technical report, USA, 1978. URL https://dl.acm.org/doi/book/10.5555/889802.

Carl Hewitt. Actor model of computation: Scalable robust information systems. Technical Report v32, 2010.

Masayuki Higashino, Shin Osaki, Shinya Otagaki, Kenichi Takahashi, Takao Kawamura, and Kazunori Sugahara. Debugging mobile agent systems. In *Proceedings of International Conference on Information Integration and Web-based Applications & Services*, IIWAS '13, pages 667:667–667:670, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2113-6. doi: 10.1145/2539150.2539261. URL http://doi.acm.org/10.1145/2539150.2539261.

Jens Ellenberg, Bastian Karstaedt, Sören Voskuhl, Kai von Luck, and Birgit Wendholt. An environment for context-aware applications in smart homes. In *International Conference on Indoor Positioning and Indoor Navigation (IPIN) // 2011 International Conference on Indoor Positioning and Indoor Navigation (IPIN 2011)*, Piscataway, NJ, 2011. IEEE. ISBN 978-1-4577-1804-5.

R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere. The architecture tradeoff analysis method. In *Proceedings / Fourth IEEE International Conference on Engineering of Complex Computer Systems, August 10 - 14, 1998, Monterey, California*, pages 68–78, Los Alamitos, Calif., 1998. IEEE Computer Soc. Press. ISBN 0-8186-8597-2. doi: 10.1109/ICECCS.1998.706657.

Matthias Koch. New re dimensions for digital ecosystems - initial results from an expert interview study. In Daniela Damian, Anna Perini, and Seok-Won Lee, editors, *2019 IEEE 27th International Requirements Engineering Conference*, pages 398–403, Piscataway, NJ, 2019. IEEE. ISBN 978-1-7281-3912-8. doi: 10.1109/RE.2019.00052.

Ilya Kolchinsky and Assaf Schuster. Join query optimization techniques for complex event processing applications. Technical Report 11, 2018. URL https://arxiv.org/pdf/1801.09413.

Klaus Krippendorff. *Content analysis: An introduction to its methodology*. SAGE, Thousand Oaks Calif., 2nd ed. edition, 2004. ISBN 0761915443.

Udo Kuckartz. *Qualitative Inhaltsanalyse: Methoden, Praxis, Computerunterstützung*. Grundlagentexte Methoden. Beltz Juventa, Weinheim and Basel, 4. auflage edition, 2018. ISBN 3779936828. URL http://www.beltz.de/de/nc/verlagsgruppe-beltz/gesamtprogramm.html?isbn=978-3-7799-3682-4.

Fadwa Lachhab, Mohammed Essaaidi, Mohamed Bakhouya, and Radouane Ouladsine. Towards a context-aware platform for complex and stream event processing. In *2016 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2016. doi: 10.1109/hpcsim.2016.7568438.

Asif Ali Laghari, Kaishan Wu, Rashid Ali Laghari, Mureed Ali, and Abdullah Ayub Khan. A review and state of art of internet of things (iot). *Archives of Computational Methods in Engineering*, 29(3):1395–1413, 2022. ISSN 1134-3060. doi: 10.1007/s11831-021-09622-6.

D. N. Lam and K. S. Barber. Comprehending agent software. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems // Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, AAMAS '05, pages 586–593, New York, NY, USA, 2005. Association for Computing Machinery and ACM. ISBN 1595930930 // 1-59593-093-0. doi: 10.1145/1082473.1082562.

Eike Langbehn, Tobias Eichler, Sobin Ghose, Kai von Luck, Gerd Bruder, and Frank Steinicke. Evaluation of an omnidirectional walking-in-place user interface with virtual locomotion speed scaled by forward leaning angle. In *Proceedings of the GI Workshop on Virtual and Augmented Reality (GI VR/AR)*, pages 149–160, 2015.

Youn Kyu Lee, Jae young Bang, Joshua Garcia, and Nenad Medvidovic. Viva: a visualization and analysis tool for distributed event-based systems. In Pankaj Jalote, editor, *Companion Proceedings of the 36th International Conference on Software Engineering*, ACM Digital Library, pages 580–583, New York, NY, 2014. ACM. ISBN 978-1-4503-2768-8. doi: 10.1145/2591062.2591074. URL http://doi.acm.org/10.1145/2591062.2591074.

Leopold Lehner, Johannes Magenheim, Wolfgang Nelles, Thomas Rhode, Niclas Schaper, Sigrid Schubert, and Peer Stechert. Informatics systems and modelling – case studies of expert interviews. In Nicolas Reynolds and Márta Turcsányi-Szabó, editors, *Key competencies in the knowledge society*, volume 324 of *IFIP Advances in Information and Communication Technology*, pages 222–233. Springer, Berlin and Heidelberg, 2010. ISBN 978-3-642-15377-8. doi: 10.1007/978-3-642-15378-5_22.

Shancang Li, Li Da Xu, and Shanshan Zhao. The internet of things: a survey. *Information Systems Frontiers*, 17(2):243–259, 2015. ISSN 1387-3326. doi: 10.1007/s10796-014-9492-7.

Lightbend. Akka. URL https://akka.io/. 2022-08-16.

Stanley Lima, Jaime Correia, Filipe Araujo, and Jorge Cardoso. Improving observability in event sourcing systems. *Journal of Systems and Software*, 181: 111015, 2021. ISSN 01641212. doi: 10.1016/j.jss.2021.111015.

Jinfeng Lin, Yalin Liu, and Jane Cleland-Huang. Supporting program comprehension through fast query response in large-scale systems. In *Proceedings of the 28th International Conference on Program Comprehension*, ICPC '20, pages 285–295, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379588. doi: 10.1145/3387904.3389260.

Livingplace. Livingplace website: https://livingplace.haw-hamburg.de/. URL https://livingplace.haw-hamburg.de/. 2022-06-26.

David C. Luckham. *The power of events: An introduction to complex event processing in distributed enterprise systems.* Addison-Wesley, Boston and San Francisco and New York and Toronto and Montreal and London and Munich and Paris and Madrid and Capetown and Sydney and Tokyo and Singapore and Mexico City, 6. printing edition, 2002. ISBN 9780321951830.

David C. Luckham and Frasca Brian. Complex event processing in distributed systems. Technical report, Stanford University, 1998.

Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology*, 23(4):1–37, 2014. ISSN 1557-7392. doi: 10.1145/2622669.

Cristiano Maciel, Patricia Cristiane de Souza, José Viterbo, Fabiana Freitas Mendes, and Amal El Fallah Seghrouchni. A multi-agent architecture to support ubiquitous applications in smart environments. In Fernando Koch, Felipe Meneguzzi, and Kiran Lakkaraju, editors, *Agent Technology for Intelligent Mobile Services and Smart Societies*, volume 498 of *Communications in Computer and Information Science*, pages 106–116. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015. ISBN 978-3-662-46240-9. doi: 10.1007/978-3-662-46241-6_9.

Stefano Mariani and Andrea Omicini. Coordinating activities and change. *Eng. Appl. Artif. Intell.*, 41(C):298–309, 2015. ISSN 0952-1976. doi: 10.1016/j.engappai.2014.10.006.

Mark Mason. Sample size and saturation in phd studies using qualitative interviews. *Forum qualitative Sozialforschung/Forum: qualitative social research*, 2010, 2010.

Stan McClellan, Jesus A. Jimenez, and George Koutitas, editors. *Smart Cities: Applications, Technologies, Standards, and Driving Factors*. Springer International Publishing, Cham, 2018. ISBN 9783319593814. doi: 10.1007/978-3-319-59381-4.

Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005. ISSN 0360-0300. doi: 10.1145/1118890.1118892.

A. Meslin, N. Rodriguez, and M. Endler. A scalable multilayer middleware for distributed monitoring and complex event processing for smart cities. In *2018 IEEE International Smart Cities Conference (ISC2)*, pages 1–8, 2018.

Michael Meuser and Ulrike Nagel. ExpertInneninterviews: vielfach erprobt, wenig bedacht. In Detlef Garz and Klaus Kraimer, editors, *Qualitativ-empirische Sozialforschung*, Springer eBook Collection, pages 441–471. VS Verlag für Sozialwissenschaften, Wiesbaden, 1991. ISBN 978-3-322-97024-4. doi: 10.1007/978-3-322-97024-4_14.

Michael Meuser and Ulrike Nagel. The expert interview and changes in knowledge production. In Alexander Bogner, Beate Littig, and Wolfgang Menz, editors, *Interviewing experts*, Research methods series, pages 17–42. Palgrave Macmillan, Basingstoke England and New York, 2009. ISBN 978-0-230-24427-6. doi: 10.1057/9780230244276_2.

Sven Meyer and Andry Rakotonirainy. A survey of research on context-aware homes. In *Proceedings of the Australasian Information Security Workshop Conference on ACSW Frontiers 2003 - Volume 21*, ACSW Frontiers '03, pages 159–168, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc. ISBN 1-920682-00-7. URL `http://dl.acm.org/citation.cfm?id=827987.828005`.

Peter Mikulecky. User adaptivity in smart workplaces. In Jeng-Shyang Pan, editor, *Intelligent information and database systems*, volume 7197 of *Lecture notes in computer science Lecture notes in artificial intelligence*, pages 401–410. Springer, Heidelberg, 2012. ISBN 978-3-642-28489-2. doi: 10.1007/978-3-642-28490-8_42.

Biswajeeban Mishra and Attila Kertesz. The use of mqtt in m2m and iot systems: A survey. *IEEE Access*, 8:201071–201086, 2020. doi: 10.1109/ACCESS.2020.3035849.

National Science Foundation. Cyber-physical systems (cps). Technical report, National Science Foundation, 2021. URL https://www.nsf.gov/pubs/2021/nsf21551/nsf21551.htm.

Claus Ballegaard Nielsen, Peter Gorm Larsen, John Fitzgerald, Jim Woodcock, and Jan Peleska. Systems of systems engineering: Basic concepts, model-based techniques, and research directions. *ACM Computing Surveys*, 48(2):1–41, 2015. ISSN 0360-0300. doi: 10.1145/2794381.

Stavros Nousias, Nikos Piperigkos, Gerasimos Arvanitis, Apostolos Fournaris, Aris S. Lalos, and Konstantinos Moustakas. *Empowering cyberphysical systems of systems with intelligence*. arXiv, 2021. doi: 10.48550/arXiv.2107.02264.

OASIS. Mqtt version 5.0, 2019-03-07. URL https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html.

Andrea Omicini, Giancarlo Fortino, and Stefano Mariani. Blending event-based and multi-agent systems around coordination abstractions. In Tom Holvoet and Mirko Viroli, editors, *Coordination Models and Languages // Coordination models and languages*, Lecture notes in computer science, pages 186–193, Cham, 2015. Springer International Publishing and Springer. ISBN 978-3-319-19282-6.

Shin Osaki, Masayuki Higashino, Kenichi Takahashi, Takao Kawamura, and Kazunori Sugahara. A framework to mitigate debugging difficulty on agent migration. In Stephane Loiseau, editor, *Proceedings of the International Conference on Agents and Artificial Intelligence*, pages 190–197, S.l., 2015. SCITEPRESS. ISBN 978-989-758-073-4. doi: 10.5220/0005224401900197.

Marc-Oliver Pahl and Stefan Liebald. Designing a data-centric internet of things. In *2019 International Conference on Networked Systems (NetSys) (NetSys'19)*, 2019.

F. Paraiso, G. Hermosillo, R. Rouvoy, P. Merle, and L. Seinturier. A middleware platform to federate complex event processing. In *2012 IEEE 16th International Enterprise Distributed Object Computing Conference*, pages 113–122, 2012. doi: 10.1109/EDOC.2012.22.

Juan Marcelo Parra-Ullauri, Antonio García-Domínguez, Juan Boubeta-Puig, Nelly Bencomo, and Guadalupe Ortiz. Towards an architecture integrating complex event processing and temporal graphs for service monitoring. In Chih-Cheng Hung, editor, *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, ACM Digital Library, pages 427–435, New York,NY,United States, 2021. Association for Computing Machinery. ISBN 9781450381048. doi: 10.1145/3412841.3441923.

Adrian Paschke and Paul Vincent. A reference architecture for event processing. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 25:1–25:4, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-665-6. doi: 10.1145/1619258.1619291. URL http://doi.acm.org/10.1145/1619258.1619291.

Pankesh Patel and Damien Cassou. Enabling high-level application development for the internet of things. *Journal of Systems and Software*, 103:62–84, 2015. ISSN 01641212. doi: 10.1016/j.jss.2015.01.027.

Anil Patidar and Ugrasen Suman. A survey on software architecture evaluation methods. *Proceedings of the 9th INDIACom*, 2015.

Orasa Patsadu, Chakarida Nukoolkit, and Bunthit Watanapa. Human gesture recognition using kinect camera. In Boonserm Kijsirikul, editor, *2012 International Joint Conference on Computer Science and Software Engineering (JCSSE 2012)*, pages 28–32, Piscataway, NJ, 2012. IEEE. ISBN 978-1-4673-1921-8. doi: 10.1109/JCSSE.2012.6261920.

Richard Plate. Assessing individuals' understanding of nonlinear causal structures in complex systems. *System Dynamics Review*, 26(1):19–33, 2010. ISSN 08837066. doi: 10.1002/sdr.432.

Johanna Plattner, Elena Oberrauner, Daniela Elisabeth Ströckl, and Johannes Oberzaucher. Using iot middleware solutions in interdisciplinary research projects in the context of aal. In Fillia Makedon, editor, *Proceedings of the 13th ACM International Conference on PErvasive Technologies Related to Assistive Environments*, ACM Digital Library, pages 1–6, New York,NY,United States, 2020. Association for Computing Machinery. ISBN 9781450377737. doi: 10.1145/3389189.3397986.

Stefan Poslad. *Ubiquitous Computing: Smart Devices, Environments and Interactions*. John Wiley & Sons Ltd, Hoboken, 2nd ed. edition, 2009. ISBN 9780470035603. URL http://gbv.eblib.com/patron/FullRecord.aspx?p=427911.

David Poutakidis, Lin Padgham, and Michael Winikoff, editors. *Debugging multi-agent systems using design artifacts: Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems ; July 15 - 19, 2002, Palazzo de Enzo, Bologna, Italy ; featuring 6th International Conference on Autonomous Agents, 5th International Conference on Multiagent Systems, 9th International Workshop on Agent Theories, Architectures, and Languages*. ACM Press, New York, NY, 2002. ISBN 1581134800. doi: 10.1145/544862.544966.

Ella Rabinovich, Opher Etzion, Sitvanit Ruah, and Sarit Archushin. Analyzing the behavior of event processing applications. In Jean Bacon, editor, *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, pages 223–234, New York, NY, 2010. ACM. ISBN 9781605589275. doi: 10.1145/1827418.1827465.

Randall R. Stewart. Stream control transmission protocol, 2007. URL https://www.rfc-editor.org/info/rfc4960.

Anand Ranganathan and Roy H. Campbell. A middleware for context-aware agents in ubiquitous computing environments. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, pages 143–161. Springer-Verlag New York, Inc, 2003. URL http://dl.acm.org/citation.cfm?id=1515915.1515926.

Mohammad Abdur Razzaque, Marija Milojevic-Jevric, Andrei Palade, and Siobhan Clarke. Middleware for internet of things: A survey. *IEEE Internet of Things Journal*, 3(1):70–95, 2016. doi: 10.1109/JIOT.2015.2498900.

Mitchel Resnick. All i really need to know (about creative thinking) i learned (by studying how children learn) in kindergarten. In *Proceedings of the 6th ACM SIGCHI Conference on Creativity &Amp; Cognition*, C&C '07, pages 1–6, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-712-4. doi: 10.1145/1254960.1254961.

Ralf H. Reussner, Heinz W. Schmidt, and Iman H. Poernomo. Reliability prediction for component-based software architectures. *Journal of Systems and Software*, 66(3):241–252, 2003. ISSN 01641212. doi: 10.1016/S0164-1212(02)00080-8.

Alessandro Ricci. From actor event-loop to agent control-loop. In Elisa Gonzalez Boix, editor, *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control*, ACM Digital Library, pages 121–132, New York, NY, 2014. ACM. ISBN 9781450321891. doi: 10.1145/2687357.2687361.

Stuart Russell and Peter Norvig. *Artificial intelligence: A modern approach*. Always learning. Pearson, Boston, third edition edition, 2016. ISBN 9781292153971. doi: Stuart. URL https://ebookcentral.proquest.com/lib/kxp/detail.action?docID=5831883.

Daniel S. Santos, Brauner R. N. Oliveira, Rick Kazman, and Elisa Y. Nakagawa. Evaluation of systems-of-systems software architectures: State of the art and future perspectives. *ACM Computing Surveys*, 2022. ISSN 0360-0300. doi: 10.1145/3519020.

Sathish and S. Smys. A survey on internet of things (iot) based smart systems. *Journal of ISMAC*, 2(4):181–189, 2020. doi: 10.36548/jismac.2020.4.001.

Mark Saunders. *Research methods for business students*. Pearson, Harlow, United Kingdom, eigth edition edition, 2019. ISBN 9781292208794. URL https://elibrary.pearson.de/book/99.150005/9781292208794.

Claudio Savaglio, Giancarlo Fortino, Maria Ganzha, Marcin Paprzycki, Costin BÄfdicÄf, and Mirjana Ivanović. Agent-based computing in the internet of things: A survey. *Studies in Computational Intelligence*, 737:307–320, 2017. doi: 10.1007/978-3-319-66379-1_27.

B. N. Schilit and M. M. Theimer. Disseminating active map information to mobile hosts. *IEEE Network*, 8(5):22–32, 1994. ISSN 0890-8044. doi: 10.1109/65.313011.

Margrit Schreier. *Qualitative content analysis in practice*. SAGE, Los Angeles and London and New Delhi and Singapore and Washington DC, 2012. ISBN 9781849205924.

Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. Distributed complex event processing with query rewriting. In Aniruddha Gokhale, editor, *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, page 1, New York, NY, 2009. ACM. ISBN 9781605586656. doi: 10.1145/1619258.1619264.

P. Shanmugapriya and R. M. Suresh. Software architecture evaluation methods a survey. *International Journal of Computer Applications*, 49(16):19–26, 2012. ISSN 0975-8887. doi: 10.5120/7711-1107.

Jianhua Shi, Jiafu Wan, Hehua Yan, and Hui Suo. A survey of cyber-physical systems. In *2011 International Conference on Wireless Communications and*

Signal Processing (WCSP 2011), pages 1–6, Piscataway, NJ, 2011. IEEE. ISBN 978-1-4577-1010-0. doi: 10.1109/WCSP.2011.6096958.

Yoav Shoham. Agent-oriented programming. *Artif. Intell.*, 60(1):51–92, 1993. ISSN 0004-3702. doi: 10.1016/0004-3702(93)90034-9.

Janet Siegmund. Program comprehension: Past, present, and future. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2016. doi: 10.1109/SANER.2016.35.

John Soldatos, Ippokratis Pandis, Kostas Stamatis, Lazaros Polymenakos, and James L. Crowley. Agent based middleware infrastructure for autonomous context-aware ubiquitous computing services. *Comput. Commun.*, 30(3):577–591, 2007. ISSN 0140-3664. doi: 10.1016/j.comcom.2005.11.018.

John Soldatos, Nikos Kefalakis, Manfred Hauswirth, Martin Serrano, Jean-Paul Calbimonte, Mehdi Riahi, Karl Aberer, Prem Prakash Jayaraman, Arkady Zaslavsky, Ivana Podnar Žarko, Lea Skorin-Kapov, and Reinhard Herzog. Openiot: Open source internet-of-things in the cloud. In Ivana Podnar Žarko, Krešimir Pripužić, and Martin Serrano, editors, *Interoperability and open-source solutions for the internet of things*, volume 9001 of *Lecture notes in computer science*, pages 13–25. Springer, Cham, 2015. ISBN 978-3-319-16545-5. doi: 10.1007/978-3-319-16546-2_3.

Eleni Stroulia and Tarja Systä. Dynamic analysis for reverse engineering and program understanding. *ACM SIGAPP Applied Computing Review*, 10(1):8–17, 2002. ISSN 1559-6915. doi: 10.1145/568235.568237.

Sriskandarajah Suhothayan, Kasun Gajasinghe, Isuru Loku Narangoda, Subash Chaturanga, Srinath Perera, and Vishaka Nanayakkara. Siddhi: A second look at complex event processing architectures. In *Proceedings of the 2011 ACM Workshop on Gateway Computing Environments*, GCE '11, pages 43–50, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1123-6. doi: 10.1145/2110486.2110493.

Rossilawati Sulaiman, Dharmendra Sharma, Wanli Ma, and Dat Tran. A multi-agent security architecture. In *2009 Third International Conference on Network and System Security*. IEEE, 2009. doi: 10.1109/nss.2009.78.

Nagender Kumar Suryadevara and Subhas Chandra Mukhopadhyay. *Smart Homes*, volume 14. Springer International Publishing, Cham, 2015. ISBN 978-3-319-13556-4. doi: 10.1007/978-3-319-13557-1.

H. Takahashi, T. Suganuma, and N. Shiratori. Amuse: an agent-based middleware for context-aware ubiquitous services. In *Parallel and Distributed Systems, 2005. Proceedings. 11th International Conference on // Proceedings / 11th International Conference on Parallel and Distributed Systems*, volume 1, pages 743–749 Vol. 1, Los Alamitos, Calif., 2005. IEEE Computer Society. ISBN 0-7695-2281-5. doi: 10.1109/ICPADS.2005.66.

Andre L.C. Tavares and Marco Tulio Valente. A gentle introduction to osgi. *ACM SIGSOFT Software Engineering Notes*, 33(5):1–5, 2008. ISSN 0163-5948. doi: 10.1145/1402521.1402526.

Kerry Taylor and Lucas Leidinger. Ontology-driven complex event processing in heterogeneous sensor networks. In Grigoris Antoniou, editor, *The semantic web: research and applications*, Lecture notes in computer science, pages 285–299, Heidelberg, 2011. Springer. ISBN 9783642210631.

V. Thangam and K. Chandrasekaran. Elliptic curve based proxy re-encryption. In Unknown, editor, *Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies*, ACM Digital Library, pages 1–6, New York, NY, 2016. ACM. ISBN 9781450339629. doi: 10.1145/2905055.2905337.

Marc H. van Liedekerke and Nicholas M. Avouris. Debugging multi-agent systems. *Information and Software Technology*, 37(2):103–112, 1995. ISSN 0950-5849. doi: 10.1016/0950-5849(95)93487-Y.

VERBI Software. Maxqda 2022. computer program, 2019. URL https://www.maxqda.com.

Chao Wang, Christopher Gill, and Chenyang Lu. Real-time middleware for cyber-physical event processing. *ACM Trans. Cyber-Phys. Syst.*, 3(3), 2019. ISSN 2378-962X. doi: 10.1145/3218816.

Yongheng Wang and Kening Cao. Context-aware complex event processing for event cloud in internet of things. In *2012 International Conference on Wireless Communications and Signal Processing (WCSP)*. IEEE, 2012. doi: 10.1109/wcsp.2012.6542861.

M. Weiser, R. Gold, and J. S. Brown. The origins of ubiquitous computing research at parc in the late 1980s. *IBM Syst. J.*, 38(4):693–696, 1999. ISSN 0018-8670. doi: 10.1147/sj.384.0693.

Mark Weiser. The computer for the 21st century. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3(3):3–11, 1991. ISSN 1559-1662. doi: 10.1145/329124.329126. URL http://doi.acm.org/10.1145/329124.329126.

V. Williams, S. Terence J., and J. Immaculate. Survey on internet of things based smart home. In *2019 International Conference on Intelligent Sustainable Systems (ICISS)*, pages 460–464, 2019. doi: 10.1109/ISS1.2019.8908112.

Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995. ISSN 0269-8889. doi: 10.1017/S0269888900008122.

World Wide Web Consortium. Owl 2 web ontology language, 2012. URL https://www.w3.org/TR/owl2-overview/.

Chao-Lin Wu, Chun-Feng Chun-Feng Liao, and Li-Chen Li-Chen Fu. Service-oriented smart-home architecture based on osgi and mobile-agent technology. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 37(2):193–205, 2007. ISSN 1094-6977. doi: 10.1109/TSMCC.2006.886997.

Muneer Bani Yassein, Mohammed Q. Shatnawi, Shadi Aljwarneh, and Razan Al-Hatmi. Internet of things: Survey and open issues of mqtt protocol. In *2017*

International Conference on Engineering & MIS (ICEMIS'2017), pages 1–6, Piscataway, NJ, 2017. IEEE. ISBN 978-1-5090-6778-7. doi: 10.1109/ICEMIS.2017.8273112.

Robert K. Yin. *Case study research: Design and methods*, volume 5 of *Applied social research methods series*. SAGE, Los Angeles, Calif., 4. ed., [nachdr.] edition, 2010. ISBN 9781412960991.

G. Michael Youngblood, Edwin O. Heierman, Lawrence B. Holder, and Diane J. Cook. Automation intelligence for the smart environment. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, IJCAI'05, pages 1513–1514, San Francisco, CA, USA, 2005. Morgan Kaufmann Publishers Inc.

A. A. Zaidan and B. B. Zaidan. A review on intelligent process for smart home applications based on iot: coherent taxonomy, motivation, open challenges, and recommendations. *Artificial Intelligence Review*, 53(1):141–165, 2020. ISSN 0269-2821. doi: 10.1007/s10462-018-9648-9.

Ariane Ziehn. Complex event processing for the internet of things. In *International Conference on Very Large Data Bases. International Conference on Very Large Data Bases (VLDB-2020), PhD Workshop, befindet sich VLDB, August 31-September 4, Tokio, Japan*. CEUR-WS, 2020.

Xiangrong Zu, Yan Bai, and Xu Yao. Data-centric publish-subscribe approach for distributed complex event processing deployment in smart grid internet of things. In *2016 7th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 710–713, 2016. doi: 10.1109/ICSESS.2016.7883166.

Sotiris Zygiaris. Smart city reference model: Assisting planners to conceptualize the building of smart city innovation ecosystems. *Journal of the Knowledge Economy*, 4(2):217–231, 2013. ISSN 1868-7873. doi: 10.1007/s13132-012-0089-4.

Appendix A

Expert Interview Details

A.1 Expert Interview Guideline

The interview guideline contains key questions for each of the three topics. Each key question is accompanied by multiple sub-questions that can be used during the interview to find out more details about important aspects.

A.1.1 Middleware and Software Development in Smart System Laboratory Environments

- In which software laboratories do you have worked so far?
  - What is/was your role inside this laboratory?
  - What is/was the goal of this environment?
  - How is/was the rough software architecture?
- What are the main requirements to the software in the laboratory?
  - What software platform or middleware is used?
  - Are there any special characteristics regarding the environment?

  - What are the differences to software environments in a business context?
  - Are students involved in the team? How are they supervised?
  - How are changes documented?

- What are according to you the biggest challenges during the operation of this environment?
  - How are changes to the architecture are made? Who is responsible?
  - How high is perceived complexity of the system?
    - Does the provided documentation help?
    - How fast are program comprehension tasks?
    - What are the difficulties in working with students?

A.1.2 Program Comprehension and Debugging in Smart System Laboratory Environments

- In your experience, what problems occurred most often in these environments or with this type of software?
  - Problems with message delivery
  - Integration
  - Faulty agents
  - External data / sensors
  - Documentation
- What solution strategies have you used to solve these problems?
  - How can components be localised in the event of an error?
  - How can you find out which messages are involved in the error?
  - How can it be determined whether a message has really been delivered in case of doubt?

- How is complex sensors (e.g. Skeleton Tracking Sensor) handled in the event of a problem?
- Are these strategies applicable at runtime?

- What software has been used for debugging in the lab environments you are familiar with?
  - During development?
  - During testing?
  - In the production system?

A.1.3 Evaluation of Program Comprehension and Debugging with CEP

- What experience do you have with CEP so far?
  - No: Show an example with an event processing pipeline that combines multiple events
  - Yes: Have you used CEP in your lab environments so far?
  - Can you see possible use cases of CEP inside your work environment?
- 1. Demo CEP Queries Agent Development
  - CEP query for sensor testing/verification
  - CEP query for ad hoc visualisation of message data
- What is your opinion about using CEP queries for agent development?
  - Would agent development with CEP Queries be helpful for the lab environments you know?
  - What about ad hoc agents for event generation/message verification/etc.?
  - What agents could be replaced by this development method?
- 2. Demo CEP Queries Debugging

  - Visualisation of agent/group graphs
  - Selection of involved agents and groups of an incident
- What is your opinion regarding debugging with CEP queries?
  - Would the type of debugging be helpful for the lab environments you know?
  - Can it improve the localisation of messages and agents?
  - Is the selection of subsystems meaningful?

A.2 Coding Frame

- Personal Information
  - Role
  - Experiences
- Software Laboratories
  - Goals
  - Software Architecture
    - Complexity
    - Messaging API
    - Messaging Format
    - Messaging Software
    - Security
    - Architecture Change Management
- **Challenges due to the environment**
  - Team Size
  - Training Process

  - Documentation Process

- **Challenges**
  - Perceived Complexity
  - Multi User Application
  - Working with Students
- **Problem Fields**
  - Complexity
  - Distributed Systems
  - Faulty Agents
  - Messaging
  - Documentation
  - Sensors
- **Approaches**
  - Message Validation
  - Debugging Strategies
  - Debugging and Testing Software
- **Experience with CEP**
  - A lot of experience - CEP used in own projects or used in research.
  - Some experience - Heard from CEP but only superficial experience with the use
  - No experience - No knowledge about CEP
- **Feedback**
  - Temporal logic
  - Speed up processes

  - Detection of complex events
  - Query language
  - Topic for further research

- **Potential Use Cases**
  - Detection of complex events
  - Monitoring
  - Program Comprehension
  - Message debugging
  - Replay of messages

- **Limitations**
  - Needs ACK messages
  - Technical limitations
  - Training time
  - Low level errors
  - Performance

A.3 Interview Information Sheet and Consent Form

The following 2 pages contain the interview information sheet and then the consent form. Both documents were used to conduct the expert interviews in Chapter 6.

Interview - Software Development in Smart Environment Research Laboratories
Researcher: Tobias Eichler (tobias.eichler@haw-hamburg.de)

Thank you for your interest in participating in this user study. The purpose of this research is to get insights on how we work with software in smart environment laboratories to improve software development and debugging tasks in these challenging environments.

Do I have to take part?
It is up to you to decide whether to take part. If you do decide to take part you will be asked to sign a consent form. You are free to withdraw at any time and without giving a reason. A decision not to participate will not affect you in anyway.

What will happen if I take part?
I will invite you to participate in a online 30-45 minutes long interview and ask you different questions about your experiences and opinions about working with smart environment or distributed loosely coupled software.

Will my participation in this study be kept confidential?
All the information which is collected about you during the course of the research will be kept strictly confidential. Any published information will have your name, address and all other identifiable details removed so that you cannot be recognised from it.

What will happen to the results of the research study?
Results obtained using the data in these experiments may be used in research publications such as conference papers, research journals and dissertations.

Who is organising the research?
This PhD research study is organised by the University of Applied Sciences Hamburg and the School of Computing, Engineering and Physical Sciences at the University of the West of Scotland.

Who has reviewed the study?
The research proposal has been reviewed by the PhD supervisors and the University of the West of Scotland's Ethics committee.

Further Information

Please contact:
Principle Investigator:
Tobias Eichler: Tobias.Eichler@haw-hamburg.de

Supervisor Team:
Qi Wang: qi.wang@uws.ac.uk
Kai von Luck: kai.vonluck@haw-hamburg.de
Susanne Draheim: susanne.draheim@haw-hamburg.de

Participant Consent Form

Interview - Software Development in Smart Environment Research Laboratories

Researcher: Tobias Eichler (tobias.eichler@haw-hamburg.de)

I voluntarily agree to participate in this research study.

I confirm that I have read and understand the plain language statement for the above study and have had the opportunity to ask questions.

I understand that even if I agree to participate now, I can withdraw at any time or refuse to answer any question without any consequences of any kind.

I understand that I can withdraw permission to use data from my interview within two weeks after the interview, in which case the material will be deleted.

I understand that I will be asked questions and my responses will be video- and audio-recorded, but that I will not be identified by name in any resulting published work. All information I provide for this study will be treated confidentially.

I agree that my answers can be published anonymised as part of scientific papers and dissertations.

Name of Participant:	**Researcher:**
Date:	**Date:**
Signature:	**Signature:**

Appendix B

Scenario-based Analysis

B.1 Questionnaire 1

The following 3 pages contain the questionnaire used for the assessment and collection of additional scenarios for the scenario-based evaluation in Chapter 8.

Questionnaire: Scenario-based Architecture Evaluation of Smart System Research Labs

Living Place Hamburg / CSTI

In both the Living Place Hamburg (LP) and the Creative Space for Technical Innovations (CSTI) we use a software architecture based on a loose coupling of software components via a publish/subscribe system. The following questionnaire is intended to further investigate the requirements of different stakeholders for the software architecture in these laboratories. The questions are aimed at evaluating known scenarios and identifying new ones that are of interest to the stakeholders of the laboratories. The results of this survey will be compiled into a list of scenarios sorted by importance, which will be used as a basis for further research on the suitability of architectures.

Consent

Participation in this survey is voluntary and anonymous. All data will be stored locally and processed in accordance with the DSGVO (in particular articles 6 and 16). This ensured that at no time an identification of specific individuals is possible based on the collected data.

The results of the questionnaires will be used as part of my PhD thesis and future publications.

By submitting the questionnaire, you agree to the above conditions.

Definitions

Agent - Component in the system that communicates via messages.

Group - Communication channel (also often called topic) to which all agents in the system can send messages. Messages are then delivered to all agents who have previously subscribed to the selected group.

Middleware Node - Distributed software component that provides the publish/subscribe communication layer for the agents.

Queries - Queries in a descriptive language (e.g. similar to SQL) about messages from agents and system states, optionally filtered by logical and temporal logical expressions, aggregated or joined with other message channels.

Contact

If you have any questions, please contact tobias.eichler@haw-hamburg.de

---

Questions

1.) How would you describe your professional role? (Multiple selections possible)

☐ Bachelor / Master Student
☐ PhD Student
☐ Postdoc
☐ Professor
☐ Staff Member
other:

2.) Please rate the following scenarios from not important to very important from your perspective and experience of working with past or current architectures in the Living Place and/or CSTI. Please mark (x) one cell per line.

	Category	**Scenario**	Not Important	Slightly Important	Moderately Important	Important	Very Important
1	Performance	Developers can use the system to implement user interaction with reasonable message latencies					
2	Performance	The performance of the system scales with the addition of more middleware nodes up to at least 1000 agents					
3	Availability	System functions are available as long as more than half of all of the middleware nodes are available					
4	Availability	Uncontrolled sending of messages from an agent does not affect the functioning of the system					
5	Modifiability	Developers can connect additional technologies, like other messaging systems or previously unsupported but compatible protocols, to the middleware within a day					
6	Modifiability	Developers can create simple full-featured agents via a descriptive language					
7	Modifiability	Developers can add or change message formats within a day					
8	Variability	Developers can connect new components written in commonly used programming languages (such as Java, C and Javascript) to the middleware in a matter of hours.					
9	Modularity	Administrators can exchange system components and use them independently of each other					
10	Functionality	The system offers developers the possibility to subscribe to, edit, filter and forward messages from agents via queries					
11	Functionality	The system can show developers the current communication graph, filtered on demand, with all agents and groups					
12	Functionality	Developers can search for specific agents and messages with specific properties via queries and thus localise errors in the system					
13	Functionality	Developers can query the system state and messages of agents ad hoc via queries to interactively learn about the structure of a running system					
14	Functionality	Developers can generate test settings and test data with a descriptive language without having to write their own components					
15	Conceptual integrity	Developers can think about all components of the system as agents with dedicated tasks					

3.) Please add any other scenarios you can think of that are not listed above. The Category column is optional, and all fields can be completed in either English or German.

	Category	Scenario	Not Important	Slightly Important	Moderately Important	Important	Very Important
16							
17							
18							
19							
20							
21							
22							
23							

Thank you for participating in this survey!

Please send the completed form to tobias.eichler@haw-hamburg.de

B.2 Questionnaire 2 and Consent Form

Questionnaire: Scenario-based Architecture Evaluation of Smart System Research Labs / Second survey

Living Place Hamburg / CSTI

In both the Living Place Hamburg (LP) and the Creative Space for Technical Innovations (CSTI) we use a software architecture based on a loose coupling of software components via a publish/subscribe system. The following questionnaire is intended to further investigate the requirements of different stakeholders for the software architecture in these laboratories. The questions are aimed at evaluating known scenarios and identifying new ones that are of interest to the stakeholders of the laboratories. The results of this survey will be compiled into a list of scenarios sorted by importance, which will be used as a basis for further research on the suitability of architectures.

This second questionnaire is used to evaluate the scenarios given in the first questionnaire.

Consent

Participation in this survey is voluntary and anonymous. All data will be stored locally and processed in accordance with the DSGVO (in particular articles 6 and 16). This ensured that at no time an identification of specific individuals is possible based on the collected data.

The results of the questionnaires will be used as part of my PhD thesis and future publications.

By submitting the questionnaire, you agree to the above conditions.

Definitions

Agent - Component in the system that communicates via messages.

Group - Communication channel (also often called topic) to which all agents in the system can send messages. Messages are then delivered to all agents who have previously subscribed to the selected group.

Queries - Queries in a descriptive language (e.g. similar to SQL) about messages from agents and system states, optionally filtered by logical and temporal logical expressions, aggregated or joined with other message channels.

Contact

If you have any questions, please contact tobias.eichler@haw-hamburg.de

---

Questions

1.) How would you describe your professional role? (Multiple selections possible)

☐ Bachelor / Master Student
☐ PhD Student
☐ Postdoc
☐ Professor
☐ Staff Member
other:

2.) Please rate the following scenarios from not important to very important from your perspective and experience of working with past or current architectures in the Living Place and/or CSTI. Please mark (x) one cell per line.

	Category	**Scenario**	Not Important	Slightly Important	Moderately Important	Important	Very Important
16	Functionality	Developers can see in the user interface whether agents implement the correct interfaces, which ones are missing and which ones are not supported by the current communication partners.					
17	Testing	Developers can save the communication between agents in a part of the system and replay it later for testing purposes. The display and processing via queries are possible in the same way as for all messages. These messages are additionally marked with an attribute to enable filtering.					
18	Variability	Developers can freely choose group names and, if necessary, define their own naming conventions.					
19	Variability	Developers can encode any JSON-compatible attributes in messages sent to and from agents. These attributes can be searched for using queries to track these messages in the system.					
20	Functionality	The system offers developers the option to trace messages and their responses for debug purposes.					
21	Functionality	Student groups in a semester, each working on their agents, can use the system to exchange information about interfaces and dependencies of their agents.					
22	Functionality	A new group of students who wants to adapt or replace parts of the system can use the GUI to see which components are involved and what dependencies they have on the rest of the system.					
23	Functionality	Teachers can see via the GUI which agents are integrated into the system by their students and which other agents are potentially influenced by them. If necessary, the exact information exchange of these new agents can be analysed in detail.					
24	Functionality	Developers can use the system to find out which interfaces an agent supports and which messages it expects, to be able to estimate which conditions (such as sensor data) the agent is waiting for.					
25	Security	Developers can be sure that the messages are not altered by other users and have to possibility to validate the integrity.					
26	Performance	Messages from agents can be prioritised. If possible, a message with a higher priority is forwarded by the middleware before a message with a lower priority.					

Thank you for participating in this survey!

Please send the completed form to tobias.eichler@haw-hamburg.de

Participant Consent Form

ATAM Workshop – Scenario-based Evaluation of an Architecture for Smart System Research Laboratories

Researcher: Tobias Eichler (tobias.eichler@haw-hamburg.de)

I voluntarily agree to participate in this research study and I confirm having had the opportunity to ask all questions I had about the study conducted here.

I understand that even if I agree to participate now, I can withdraw at any time during the workshop or refuse to answer any questions without any consequences of any kind.

I agree that the workshop will be documented with photos and that my contributions to the discussion during the workshop will be transcribed, but that I will not be identified by name in any resulting published work. All data is stored and processed in compliance with the EU-General Data Protection Regulation (GDPR).

I agree that the results of the workshop can be published anonymised as part of scientific papers and dissertations.

Name of Participant:	**Researcher:**
Date:	**Date:**
Signature:	**Signature:**

B.3 ATAM Workshop Results

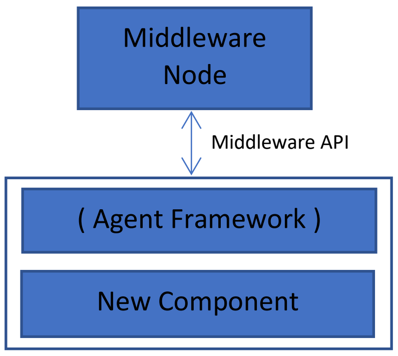
B.3.1 Analysis of Architectural Approaches

Analysis of Architectural Approach

Scenario ID:	8 - Heterogeneity
Scenario:	Developers can connect new components written in commonly used programming languages (such as Java, C and Javascript) to the middleware in a matter of hours.
Attributes:	Variability
Environment:	Normal operations
Stimulus:	Need of a new component
Response:	Successful connection of the new component

Architectural Decisions	**Sensitivity**	**Tradeoff**	**Risk**	**Nonrisk**
A4 – Open Middleware API	S1		R3	N1
A6 – API Libraries	S2		R2	
A7 – Runtime Environments			R1	

Reasoning / Diagram



Analysis of Architectural Approach

Scenario ID:	12 – System entity search
Scenario:	Developers can search for specific agents and messages with specific properties via queries and thus localise errors in the system
Attributes:	Variability
Environment:	Normal operations
Stimulus:	New CEP search query
Response:	Dynamically updated list of relevant agents and messages

Architectural Decisions	**Sensitivity**	**Tradeoff**	**Risk**	**Nonrisk**
A1 – Agent-based Design	S3		R5	N3
A8 – Integration of CEP	S4	T1, T2	R4	N2, N4

Reasoning / Diagram

Analysis of Architectural Approach

Scenario ID:	4 – Agent failure tolerance
Scenario:	Uncontrolled sending of messages from an agent does not affect the functioning of the system.
Attributes:	Availability
Environment:	Normal operations
Stimulus:	An agent starts to send messages uncontrollably
Response:	Normal operation

Architectural Decisions	**Sensitivity**	**Tradeoff**	**Risk**	**Nonrisk**
A3 – Publish/Subscribe communication		T4	R6	N5
A1 – Agent-based Design	S5	T3		N6

Reasoning / Diagram

Analysis of Architectural Approach

Scenario ID:	1 – Message latency
Scenario:	Developers can use the system to implement user interaction with reasonable message latencies.
Attributes:	Performance
Environment:	Normal operations
Stimulus:	Agent sends a message
Response:	Message is received with appropriate latency

Architectural Decisions	**Sensitivity**	**Tradeoff**	**Risk**	**Nonrisk**
A9 – Distributed Middleware Nodes	S6	T5		N7
A7 – Runtime Environments				N8
A1 – Agent-based Design			R7	

Reasoning / Diagram

Analysis of Architectural Approach

Scenario ID:	10 – Context processing
Scenario:	The system offers developers the possibility to subscribe to, edit, filter and forward messages from agents via queries.
Attributes:	Functionality
Environment:	Normal operations
Stimulus:	New CEP query
Response:	Dynamically updated list of relevant messages

Architectural Decisions	**Sensitivity**	**Tradeoff**	**Risk**	**Nonrisk**
A2 – Open Design	S7		R9	N9
A8 – Integration of CEP			R8	N10
A1 – Agent-based Design		T6		

Reasoning / Diagram

Analysis of Architectural Approach

Scenario ID:	7 – Change of message format
Scenario:	Developers can add or change message formats within a day.
Attributes:	Modifiability
Environment:	Normal operations
Stimulus:	A new message format is needed
Response:	Communication with other components via the middleware

Architectural Decisions	**Sensitivity**	**Tradeoff**	**Risk**	**Nonrisk**
A3 – Publish/Subscribe Communication		T7	R10	N11
A6 – API Libraries		T8, T9		N12

Reasoning / Diagram

Analysis of Architectural Approach

Scenario ID:	9 – Reusability of system components
Scenario:	Administrators can exchange system components and use them independently of each other.
Attributes:	Modularity
Environment:	Normal operations
Stimulus:	Parts of the system are to be reused
Response:	Normal operations in the new system

Architectural Decisions	**Sensitivity**	**Tradeoff**	**Risk**	**Nonrisk**
A10 – Layer Design			R11	N13
A2 – Open Design		T10		

Reasoning / Diagram

Analysis of Architectural Approach

Scenario ID:	3 – Node failure tolerance
Scenario:	System functions are available as long as more than half of all of the middleware nodes are available.
Attributes:	Availability
Environment:	Normal operations
Stimulus:	A middleware node fails
Response:	Normal operations

Architectural Decisions	**Sensitivity**	**Tradeoff**	**Risk**	**Nonrisk**
A9 – Distributed Middleware Nodes				N14
A4 – Open Middleware API		T11, T12	R12	N15

Reasoning / Diagram

Analysis of Architectural Approach

Scenario ID:	5 – Add new technologies
Scenario:	Developers can connect additional technologies, like other messaging systems or previously unsupported but compatible protocols, to the middleware within a day.
Attributes:	Functionality
Environment:	Normal operations
Stimulus:	Connection of a separate messaging system
Response:	Normal operations

Architectural Decisions	**Sensitivity**	**Tradeoff**	**Risk**	**Nonrisk**
A2 – Open Design				N16
A4 – Open Middleware API		T13		
A1 – Agent-based Design			R13	N18
A5 – Agent Framework				N17

Reasoning / Diagram

Analysis of Architectural Approach

Scenario ID:	14 – Test settings
Scenario:	Developers can generate test settings and test data with a descriptive language without having to write their own components.
Attributes:	Functionality
Environment:	Testing or production environment
Stimulus:	New CEP query
Response:	Test messages are sent

Architectural Decisions	**Sensitivity**	**Tradeoff**	**Risk**	**Nonrisk**
A2 – Open Design				N19
A8 – Integration of CEP				N20
A6 – API Libraries				N21

Reasoning / Diagram

Analysis of Architectural Approach

Scenario ID:	15 – Agent-based
Scenario:	Developers can think about all components of the system as agents with dedicated tasks.
Attributes:	Conceptual integrity
Environment:	Normal operations
Stimulus:	Developer works with the system
Response:	Everything behaves like an agent

Architectural Decisions	**Sensitivity**	**Tradeoff**	**Risk**	**Nonrisk**
A1 – Agent-based Design		T1	R14	N22
A8 – Integration of CEP				N23

Reasoning / Diagram

Analysis of Architectural Approach

Scenario ID:	11 – System status visualization
Scenario:	The system can show developers the current communication graph, filtered on demand, with all agents and groups.
Attributes:	Functionality
Environment:	Normal operations
Stimulus:	Visualization request
Response:	Dynamically updated graph of agents and groups

Architectural Decisions	**Sensitivity**	**Tradeoff**	**Risk**	**Nonrisk**
A1 – Agent-based Design	S9	T14		N24
A8 – Integration of CEP				N25
A4 – Open Middleware API	S8			

Reasoning / Diagram

Analysis of Architectural Approach

Scenario ID:	13 – Ad hoc queries
Scenario:	Developers can query the system state and messages of agents ad hoc via queries to interactively learn about the structure of a running system.
Attributes:	Functionality
Environment:	Normal operations
Stimulus:	New CEP query
Response:	Dynamically updated list of messages

Architectural Decisions	**Sensitivity**	**Tradeoff**	**Risk**	**Nonrisk**
A8 – Integration of CEP			R15	N26
A2 – Open Design				N27
A1 – Agent-based Design			R16	

Reasoning / Diagram

Analysis of Architectural Approach

Scenario ID:	2 - Scalability
Scenario:	The performance of the system scales with the addition of more middleware nodes up to at least 1000 agents.
Attributes:	Performance
Environment:	Normal operations
Stimulus:	New agents join the system
Response:	System scales with added middleware nodes

Architectural Decisions	**Sensitivity**	**Tradeoff**	**Risk**	**Nonrisk**
A9 – Distributed Middleware Nodes		T15		N28

Reasoning / Diagram

Analysis of Architectural Approach

Scenario ID:	6 – Descriptive programming
Scenario:	Developers can create simple full-featured agents via a descriptive language.
Attributes:	Modifiability
Environment:	Normal operations
Stimulus:	New CEP query
Response:	Query runs and behaves like an agent

Architectural Decisions	**Sensitivity**	**Tradeoff**	**Risk**	**Nonrisk**
A8 – Integration of CEP			R15	N30
A1 – Agent-based Design			R17	
A2 – Open Design				N29

Reasoning / Diagram

Analysis of Architectural Approach

Scenario ID:	24 – Display agent activation conditions
Scenario:	Developers can use the system to find out which interfaces an agent supports and which messages it expects, to be able to estimate which conditions (such as sensor data) the agent is waiting for.
Attributes:	Functionality
Environment:	Normal operations
Stimulus:	Developer opens Web GUI
Response:	Agent activation conditions

Architectural Decisions	**Sensitivity**	**Tradeoff**	**Risk**	**Nonrisk**
A1 – Agent-based design			R18	N32
A6 – API Libraries				N31, N32
A2 – Open Design				N33

Reasoning / Diagram

Analysis of Architectural Approach

Scenario ID:	19 – Freedom in the choice of attributes
Scenario:	Developers can encode any JSON-compatible attributes in messages sent to and from agents. These attributes can be searched for using queries to track these messages in the system.
Attributes:	Variability
Environment:	Normal operations
Stimulus:	Message is sent
Response:	No error

Architectural Decisions	**Sensitivity**	**Tradeoff**	**Risk**	**Nonrisk**
A6 – API Libraries			R19	N34
A2 – Open Design			R18, R20	
A8 – Integration of CEP	S10			N35

Reasoning / Diagram

Analysis of Architectural Approach

Scenario ID:	20 – Simple message tracing
Scenario:	The system offers developers the option to trace messages and their responses for debug purposes.
Attributes:	Functionality
Environment:	Normal operations
Stimulus:	Message is sent
Response:	Response message from Agent

Architectural Decisions	**Sensitivity**	**Tradeoff**	**Risk**	**Nonrisk**
A11 – Monitoring and Logging		T16, T17		N36, N37
A9 – Distributed Middleware Nodes			R20	

Reasoning / Diagram

Analysis of Architectural Approach

Scenario ID:	22 – Project handover to following semester groups
Scenario:	A new group of students who wants to adapt or replace parts of the system can use the GUI to see which components are involved and what dependencies they have on the rest of the system.
Attributes:	Functionality
Environment:	Normal operations
Stimulus:	Student group opens web GUI graph view
Response:	Dependency graph

Architectural Decisions	**Sensitivity**	**Tradeoff**	**Risk**	**Nonrisk**
A2 – Open Design			R21	N33
A11 – Monitoring and Logging		T18		
A1 – Agent-based Design				N38
A3 – Publish/Subscribe Communication				N38
A8 – Integration of CEP			R22	N39

Reasoning / Diagram

Analysis of Architectural Approach

Scenario ID:	16 – Checking interfaces of agents
Scenario:	Developers can see in the user interface whether agents implement the correct interfaces, which ones are missing and which ones are not supported by the current communication partners.
Attributes:	Functionality
Environment:	Normal operations
Stimulus:	Developer opens web GUI
Response:	List of incompatible APIs

Architectural Decisions	**Sensitivity**	**Tradeoff**	**Risk**	**Nonrisk**
A6 – API Libraries				N40
A4 – Open Middleware API			R23	N41
A1 – Agent-based Design			R24	

Reasoning / Diagram

Analysis of Architectural Approach

Scenario ID:	17 – Message recording and replay
Scenario:	Developers can save the communication between agents in a part of the system and replay it later for testing purposes. The display and processing via queries are possible in the same way as for all messages. These messages are additionally marked with an attribute to enable filtering.
Attributes:	Testing
Environment:	Normal operations
Stimulus:	Developer starts message recording
Response:	Recording of all messages from selected components

Architectural Decisions	**Sensitivity**	**Tradeoff**	**Risk**	**Nonrisk**
A11 – Monitoring and Logging		T16, T17		N36, N37
A9 – Distributed Middleware Nodes			R20	

Reasoning / Diagram

- During the analysis, it was found that the implementation of this scenario is very similar to S20.

Analysis of Architectural Approach

Scenario ID:	21 – Cooperation between student groups
Scenario:	Student groups in a semester, each working on their agents, can use the system to exchange information about interfaces and dependencies of their agents.
Attributes:	Functionality
Environment:	Normal operations
Stimulus:	Student group starts working, opens web GUI
Response:	Dependency graph

Architectural Decisions	**Sensitivity**	**Tradeoff**	**Risk**	**Nonrisk**
A1 – Agent-based Design			R25	
A3 – Publish/Subscribe Communication			R25	
A6 – API Libraries			R26	N43
A8 – Integration of CEP				N44

Reasoning / Diagram

Analysis of Architectural Approach

Scenario ID:	23 – Realisation of projects with students
Scenario:	Teachers can see via the GUI which agents are integrated into the system by their students and which other agents are potentially influenced by them. If necessary, the exact information exchange of these new agents can be analysed in detail.
Attributes:	Functionality
Environment:	Normal operations
Stimulus:	Teacher opens web GUI to collect information about student projects
Response:	Dependency graph, Interaction graph, Message recordings

Architectural Decisions	**Sensitivity**	**Tradeoff**	**Risk**	**Nonrisk**
A2 – Open Design				N42
A1 – Agent-based Design			R25	
A3 – Publish/Subscribe Communication			R25	N45
A8 – Integration of CEP		T19		N44
A11 – Monitoring and Logging				N42

Reasoning / Diagram

Analysis of Architectural Approach

Scenario ID:	18 – Freedom in the choice of group names
Scenario:	Developers can freely choose group names and, if necessary, define their own naming conventions.
Attributes:	Variability
Environment:	Normal operations
Stimulus:	Message is sent to arbitrary group name
Response:	Message is delivered to all subscribers

Architectural Decisions	**Sensitivity**	**Tradeoff**	**Risk**	**Nonrisk**
A2 – Open Design			R27, R28	N47
A3 – Publish/Subscribe Communication				N46

Reasoning / Diagram

Analysis of Architectural Approach

Scenario ID:	26 – Message prioritisation
Scenario:	Messages from agents can be prioritised. If possible, a message with a higher priority is forwarded by the middleware before a message with a lower priority.
Attributes:	Performance
Environment:	Normal operations
Stimulus:	Message is sent with high priority
Response:	Message is delivered before message with lower priority

Architectural Decisions	**Sensitivity**	**Tradeoff**	**Risk**	**Nonrisk**
A4 – Open Middleware API				N48
A3 – Publish/Subscribe Communication	S11	T20	R29	
A11 – Monitoring and Logging		T21		
A1 – Agent-based Design				N49
A5 – Agent Framework	S12			
A10 – Layer Design				N50

Reasoning / Diagram

Analysis of Architectural Approach

Scenario ID:	25 – Message integrity
Scenario:	Developers can be sure that the messages are not altered by other users and have to possibility to validate the integrity.
Attributes:	Security
Environment:	Normal operations
Stimulus:	Message is sent
Response:	Agent receives messages and detects whether it was compromised

Architectural Decisions	**Sensitivity**	**Tradeoff**	**Risk**	**Nonrisk**
A2 – Open Design		T22, T23		
A4 – Open Middleware API	S13			
A6 – API Libraries				N51

Reasoning / Diagram

- A key management system would be necessary for all agents
- Not really necessary in research context when all participants can be trusted

B.3.2 Sensitivity Points

Reference	**Description**
S1	The connection of new components is influenced by the number of supported protocols in the middleware API.
S2	The time required to connect a new component depends on whether a compatible agent framework is available for the programming language used. An implementation for Java, Javscript and C is available.
S3	Identifying components that trigger errors is easier if all components adhere to the agent-based design, because this implies that they have separate tasks.
S4	The performance of CEP integration directly influences the ability to perform CEP queries. This concerns the possible functional elements in a query and the speed of execution.
S5	Additional messages can potentially influence the message latencies (S1).
S6	The number of middleware nodes influences the message latency.
S7	The open design of the system is necessary to access all groups to process contextual information.
S8	For a complete visualisation, it is necessary that all agents adhere to the registration process provided in the middleware API.
S9	The clarity of the visualisation depends on the graph layout algorithm used. Here it is important that the position of the nodes is maintained when changes are made to the graph.
S10	CEP queries must be adapted to the custom attributes.
S11	The prioritisation of messages affects the order of delivery of messages, which affects the message ordering guarantees of the middleware.

Reference	Description
S12	The prioritisation of messages must be considered in the agent framework. This concerns both the specification of a priority when sending a message and the prioritised handling of incoming messages, if needed.
S13	In order for the middleware to check message integrity, the middleware API would have to be adapted. This could affect other features and make the work on the system more difficult.

B.3.3 Tradeoff Points

Reference	Description
T1	The scenario S23 (Realisation of projects with students) supports the testing of components in the system by a lecturer or supervisor. This can ensure the correct naming and description of components, which improves the likelihood of finding individual components via search terms.
T2	To start a search for a faulty component, a developer must at least partially understand the structure of the system. This can be improved by scenario S11 (System status visualisation).
T3	The free choice of message attributes (S19), together with uncontrolled sending for messages, can cause high loads in the system.
T4	The identification of faulty agents via CEP queries (S12) can help to quickly switch off faulty agents.
T5	The uncontrolled sending of messages can affect the scalability of the system (S2) because it increases the load disproportionately.

Reference	**Description**
T6	The more tasks in the system are performed by CEP queries, the less the system can be conventionally described as an agent-based system (S15). This is mitigated by the fact that all CEP queries are realised with the help of agents and thus have a representation in the agent-based view.
T7	The effort required for the connection of other message formats is influenced by the protocol support of the middleware API (S8).
T8	The free choice of message attributes (S19) may affect the ability to support other message formats.
T9	The ability to find obsolete or inappropriate agent interfaces via CEP queries (S16) makes it easier to change an interface over time.
T10	When replacing a system component, it is likely that the open design aspect of the architecture will be compromised. This could limit the ability to freely access the groups and thus flexibly process contextual information (S10).
T11	The failure of middleware nodes can increase the load in the system and thus affect the message latency (S1).
T12	The number of agents can affect the time it takes to perform a handover from agent to another node. This can affect the message latency (S1).
T13	The connection of external systems can have an influence on the message latencies in the system (S1).
T14	If the number of agents is very high (S2), the visualisation could become cluttered. The filter functions can help here, but require that the user knows what he is looking for.
T15	The scaling of the system (i.e. the number of middleware nodes) can influence the error tolerance of the middleware (S3) and the message latency (S1).

Reference	Description
T16	The ability to save messages and resend them later may conflict with the message integrity scenario (S25).
T17	The possibility of saving messages and sending them again later could be made more difficult by prioritising messages (S26), because the order of the messages then also depends on the priority of the messages.
T18	The visualisation of the system state (S11) can help in the exchange of information between developers.
T19	When testing student projects, testing with the help of CEP queries (S14) can be helpful.
T20	Prioritisation of messages could conflict with message latency (S1). Firstly, because prioritising all messages could produce additional administrative overhead and secondly, because lower prioritised messages would potentially have to wait longer.
T21	Message prioritisation must be taken into account when recording messages (S17) and may make the process more complicated.
T22	The introduction of a signature in messages would not allow messages to be sent freely into the system and could cause problems with the record and replay feature (S17).
T23	The free choice of attributes (S19) could cause problems together with message integrity.

B.3.4 Risks

Reference	Description
R1	A connection in a short time is only guaranteed for languages for which the agent framework exists. This is the case for all languages mentioned in the scenario, but in the future further requirements could be added.

Reference	**Description**
R2	For other languages not covered by the Agent Framework, the serialisation of JSON messages has to be implemented with the help of an external library or by yourself. This could be an error-prone task.
R3	Low-powered components, such as microcontrollers, may have performance problems with message serialisation.
R4	Only the communication of agents via the middleware can be examined. If an agent communicates via other channels, this cannot be found out via CEP queries.
R5	In order for agents and messages to be found via CEP queries, they must be provided with suitable names and, in the case of agents, with a complete description if possible.
R6	If incorrect messages are published to a group that another agent has subscribed to, the function and performance of that other agent may be affected.
R7	The network performance (for example switches) influences the message latency.
R8	The CEP query language must be understood by the users in order to use this function.
R9	The ability to simply generate contextual messages can be exploited by users to cause malicious effects or confusion. However, this is controllable because only known persons work with the system or are supervised by lecturers or superiors.
R10	Changing the message format may affect other agents. All subscribers of the groups to which messages are sent in the new format are potentially affected.
R11	Despite defined interfaces between the layers, performance differences could cause problems in the system when replacing a component.

Reference	**Description**
R12	Agents that do not use one of the agent frameworks may have faulty implementations of the handover to another node and lose the connection to the middleware.
R13	Care must be taken that the connected system has comparable performance characteristics so that all messages can be delivered.
R14	All developers must be able to think agent-based and follow the rules.
R15	The CEP request language must be understood by the developer. Unnecessarily complex queries can potentially generate longer agent chains that can make the system seem more complicated.
R16	There is a risk that an agent's behaviour cannot be read from its messages because it has been programmed not to communicate status changes.
R17	CEP queries represent a different programming paradigm than conventional procedural programming languages. It is possible that some features desired by programmers are not easy to implement.
R18	Agents are autonomous and independent. There is a risk that they may not implement the API correctly without being noticed.
R19	If custom attributes are encoded in JSON values, care must be taken to ensure that all agents implementing the associated API can also process this attribute.
R20	The use of multiple nodes may affect the order in which messages are recorded and played back. For example, in the case of connection errors and agent handover operations between nodes.
R21	There is a risk that students do not want to deal with existing components in the system and directly implement everything themselves.

Reference	Description
R22	Students need to understand how CEP integration works in order to understand how a component works when CEP queries are involved.
R23	All developers must adhere to the specification and make correct claims when using APIs.
R24	When studying APIs, it must be understood that it is a distributed system and agents can change their specifications at any time.
R25	Architecture must be understood so that it can be implemented by all students.
R26	Student documentation is often non-existent, incorrect or incomplete. Must be checked by the supervisor.
R27	A free choice of names could lead to misleading names that make the system harder to understand.
R28	Special characters and characters from other alphabets could cause technical problems. Extra test cases should be introduced here.
R29	With prioritisation of messages, it would be unclear to agents when a message would arrive. This could cause more effort in programming.

B.3.5 Nonrisks

Reference	**Description**
N1	If the programming language supports the corresponding protocols of the middleware API, a connection is possible. This is the case for the programming languages specified in the scenario (Java, Javascript and C) because the languages support at least one protocol from TCP, UDP and Websockets as well as JSON serialisation (possibly via extra libraries).
N2	The nature of the CEP engine integration allows access to all messages from agents.
N3	If the system is agent-based and all components communicate via messages, then the complete communication between the agents is also available for analysis via CEP queries.
N4	A mistake is the failure to meet expectations. If the expectations of an agent are known and can be checked via the agent's messages, then the error can also be identified via the CEP engine (see N2 and N3).
N5	As the system can handle a very high message load and is scalable, no negative effects on the rest of the system are to be expected.
N6	By separating tasks between agents, only a small part of the system should be directly affected by the false messages. Transitive effects can be reduced by correct error handling.
N7	As long as more middleware nodes can be added, the latency of messages can be reduced if necessary, provided the message load is spread over several groups.
N8	The runtime environments could run the agents in controlled environments, offsetting negative effects on message latency. For example, agents can be migrated to another runtime environment when CPU utilisation is high.
N9	The open design of the system allows access to all messages. This means that all contextual information can also be evaluated in this way.

Reference	**Description**
N10	As the CEP integration also supports temporal logic and timings, it is possible to send timed messages. This is necessary, for example, to deliver control messages for actuators at the right time.
N11	If the new message format is JSON compatible, then it is possible to integrate this format without making major changes to the system. The middleware already contains functions to convert such messages into a supported format.
N12	The versioning of the APIs allows the message format to be changed without having to adapt all agents at the same time.
N13	The design of the system in several layers enables the exchange of individual layers and increases reusability.
N14	As long as more than half of the middleware nodes are active, the system can continue to run.
N15	The middleware API regulates the handover of agents to another middleware node in the event of an error.
N16	The open design approach allows external systems to be connected and all groups to be accessed.
N17	The middleware API provides various network protocols (TCP, SCTP, UDP) to allow the connection of external systems.
N18	The agent-based design allows external systems to be encapsulated as agents and thus integrated into the system.
N19	The open design approach makes it possible to send test data to all groups.
N20	The integration of CEP makes it possible to read messages from agents, modify them if necessary and send them to other agents.
N21	By using API libraries, the middleware can help create test settings and ensure that messages are sent that match the API.
N22	The system intends that all components are agents and communicate via messages.

Reference	**Description**
N23	Since the CEP integration itself is implemented with the help of agents and all requests are realised with the help of agents, the entire system remains agent-based.
N24	Since all components of the system are agents, all parts of the system are visible in the visualisation
N25	As long as the user knows what they are looking for in the system, the graph can be filtered to that part of the system.
N26	As the CEP integration allows access to all groups in the system, it is also possible to read, edit and forward all messages in the system to any group via the CEP requests.
N27	The open design supports this function. All groups are accessible and the messages generated by a CEP request are treated like all other messages and are therefore also visible to the developer.
N28	The scalability of the system is ensured by the possibility of adding further middleware nodes. The load in the system is distributed among all middleware nodes as long as the messages are divided among several groups.
N29	As long as the agent has simple functions and is a context interpreter, the agent can most likely be implemented as a CEP query. However, sensors and actors can only be simulated because the CEP engine has no possibility to access the outside world.
N30	The seamless integration of the CEP engine allows CEP queries to act as agents in the system. All agents that realise CEP queries are registered with the middleware and are able to provide information about their status via the CEP Manager Agent.
N31	This is possible because the APIs are defined and the messages are JSON formatted and therefore easily readable.

Reference	Description
N32	Possible as long as all agents know about their interfaces anc communicate them, which is specified according to the agent-based design and the API libraries, which contain all messages specified in the provided DSL.
N33	The open design approach allows access to all information and messages of all agents.
N34	According to the DSL specification for the API libraries, attribute names are freely selectable. However, reserved words such as "type" must be escaped, which is done automatically by the API generator.
N35	As long as only JSON compatible names and attributes are used as specified in the scenario, this can also be processed by the CEP engine.
N36	As long as the messages are logged via the middleware, requests and responses from agents can also be displayed for the developers.
N37	When the messages from the selected part of the system are recorded by an agent and later resent in the order they were received, they arrive at other agents in the order.
N38	If students are able to think agent-based and understand publish/subscribe communication, they will be able to navigate the system through the given functions. If the students are able to think in an agent-based way, they can find their way around the system using the given functions.
N39	Since all CEP queries are visible as agents in the system, N38 also applies to them.
N40	If all agents adhere to the API specification, the identification of an API mismatch is possible.
N41	The middleware API ensures that all agents specify which API agents expect messages from when they perform a subscribe.

Reference	**Description**
N42	The open design and logging features of the architecture allow a supervisor to see and inspect the components and their communication built by students at any time.
N43	If the documentation is complete, it is easy to find via the middleware API and can help with the handover of projects.
N44	CEP queries can be used to check the preconditions and side effects of agents, which facilitates the handover and inspection of projects.
N45	Messages can be read via groups without interfering with students' work.
N46	The group names are freely selectable in the Publish/Subscribe system.
N47	It is good that the names are freely selectable, otherwise this would be very difficult to realise in a research context because the necessary stuctures might be missing.
N48	The middleware API would have to be adapted in order to specify the priority when publishing a message. This would be unproblematic by adding an optional parameter.
N49	Prioritising entire agents would be easier to implement than prioritising individual messages because it would be easier to respect the message ordering guarantees.
N50	If necessary, it would be possible to exchange the message layer to allow prioritisation of messages.
N51	The introduction of a signature in the message API would be easily possible. For this, another attribute would have to be reserved.