APPSTACLE

(ITEA 3 – 15017)

open standard APplication Platform
for carS and TrAnsportation vehiCLEs

Deliverable: D 3.1

Specification of Data Management, Cloud Platform Architecture and Features of the Automotive IoT Cloud Platform

Work Package: 3
Automotive IoT Cloud Platform

Task: 3.1, 3.2
Cloud Platform Architecture Specification, Evaluation of Existing IoT and Automotive Cloud Platforms

Document Type:	Deliverable	**Classification:**	Public
Document Version:	final	**Contract Start Date:**	01.01.2017
Document Preparation Date:	31.01.2018	**Duration:**	31.12.2019

ITEA3
INFORMATION TECHNOLOGY FOR EUROPEAN ADVANCEMENT

EUREKA

History

Rev.	Content	Resp. Partner	Date
0.1	initial document structure	Philipp Heisig	17.05.2017
0.2	added introduction and content for state of the art, evaluation, and architecture specification	Philipp Heisig	12.06.2017
0.3	added content for data storage and management, message routing, and data analytic and visualization	Philipp Heisig	20.07.2017
0.4	added content for device management	Philipp Heisig	04.09.2017
0.5	added automotive user stories	Philipp Heisig	13.11.2017
0.6	added generic cloud platform architecture specification	Philipp Heisig	19.12.2017
0.7	added evaluation for data management and message gateway	Philipp Heisig	15.01.2018
0.8	added evaluation for IoT Cloud Platform and application and service integration	Ahmad Bani Jamali	18.01.2018
0.9	added evaluation for data analytic and visualization and device management	Tobias Rawald	22.01.2018
1.0	added conclusion	Philipp Heisig	29.01.2018
1.1	specification & conclusion review	Robert Hoettger	06.02.2018
1.2	added evaluation of multi-model databases	Fatih Ayvaz	25.03.2018

Contents

History ii

Summary viii

List of Abbreviations ix

1 Introduction 1

1.1 Document Structure . . . 3

2 State of the art 4

2.1 Communication Protocol . . . 4

2.1.1 REST . . . 4

2.1.2 CoAP . . . 5

2.1.3 AMQP . . . 5

2.1.4 LWM2M . . . 5

2.1.5 MQTT . . . 6

2.1.6 XMPP . . . 7

2.2 IoT Cloud Platform . . . 8

2.2.1 Eclipse Kapua . . . 11

2.3 Application and Service Integration . . . 12

2.3.1 Eclipse Ditto . . . 13

2.3.2 Mihini . . . 14

2.3.3 Appstore . . . 14

2.3.4 Eureka . . . 14

2.3.5 Apache Camel . . . 15

2.3.6 Spring Integration . . . 16

2.4 Data Analytic and Visualization . . . 18

2.4.1 Apache Storm . . . 18

2.4.2 Apache Flink . . . 20

2.4.3 Eclipse BIRT . . . 21

2.4.4 Grafana . . . 22

2.5 Data Storage and Management . . . 24

2.5.1 Apache CouchDB . . . 25

2.5.2 MongoDB . . . 25

2.5.3 Neo4j . . . 27

2.5.4 InfluxDB . . . 28

2.5.5 ArangoDB . . . 29

2.5.6 OrientDB . . . 30

2.5.7 Couchbase . . . 30

2.6 Device Management . . . 30

2.6.1 Eclipse hawkBit . . . 31

2.6.2 Eclipse Leshan . . . 31
2.6.3 Eclipse Wakaama . . . 32
2.6.4 Eclipse Vorto . . . 32
2.6.5 OGC SensorThings API . . . 33
2.7 Message Gateway . . . 33
2.7.1 Eclipse Hono . . . 33
2.7.2 Apache Kafka . . . 35
2.7.3 Eclipse Mosquitto & Paho . . . 36
2.8 Security . . . 37
2.8.1 Eclipse Keti . . . 38
2.8.2 Keycloak . . . 39

3 Automotive User Stories 40
3.1 User Story 01: Roadside Assistance . . . 41
3.2 User Story 02: Vehicle Tracking . . . 41
3.3 User Story 03: Wrong Way Driver Warning . . . 42
3.4 User Story 04: Augment vehicle functionality . . . 43
3.5 User Story 05: Data Collection Fleet Learning . . . 43
3.6 User Story 06: IoT Data concentration . . . 44
3.7 User Story 07: Driver Seat Configuration . . . 45
3.8 User Story 08: Parking Space Finder . . . 45
3.9 User Story 09: Improved Carpooling System . . . 46
3.10 User Story 10: Car Accident Registration by Video . . . 46
3.11 User Story 11: Car Theft Registration & Car Vandalism Registration . . . 47
3.12 User Story 12: Traffic Jam Warning & Traffic Jam Avoidance . . . 47
3.13 User Story 13: Chat Service for Car Drivers . . . 48
3.14 User Story 14: Traffic Enforcement Camera Warning . . . 48
3.15 User Story 15: Advertising Services for Drivers . . . 49
3.16 User Story 16: Social Media . . . 49
3.17 User Story 17: Ambulance Assist . . . 50
3.18 User Story 18: System Surveillance and Maintenance . . . 50
3.19 User Story 19: Pool car management . . . 51
3.20 User Story 20: In-vehicle behavior learning . . . 52
3.21 User Story 21: Secure Car2X data exchange . . . 52
3.22 User Story 22: Emergency Braking & Evading Assistance System (EBEAS) . . . 53

4 Architecture Specification 55

5 Architecture Evaluation 58
5.1 Evaluation criteria . . . 58
5.2 Technology Evaluation . . . 59
5.2.1 IoT Cloud Platform . . . 59
5.2.2 Application and Service Integration . . . 59
5.2.3 Data Analytic and Visualization . . . 63
5.2.4 Data Storage and Management . . . 67
5.2.5 Device Management . . . 71
5.2.6 Message Gateway . . . 74

5.3 Technology Selection 76
5.3.1 IoT Cloud Platform 76
5.3.2 Application and Service Integration 77
5.3.3 Data Analytics and Visualization 77
5.3.4 Data Storage and Management 78
5.3.5 Device Management 79
5.3.6 Message Gateway 80

6 Conclusion **81**
6.1 Subsequent activities 82

List of Figures

1.1 Software stack for an IoT Cloud Platform [12] 1
1.2 Internet of Things (IoT) Platform reference architecture according to Gartner Inc. 2
1.3 End-to-end IoT architecture accroding to Intel 2
1.4 Microsoft Azure IoT reference architecture 3

2.1 Publish/subscribe pattern for Message Queue Telemetry Transport (MQTT) [] 6
2.2 Architecture of Eclipse Kapua [55] 11
2.3 Web-based administration console for all device and data management operations [55] 12
2.4 Eclipse Ditto in the overall Eclipse IoT landscape [52] 13
2.5 Mihini Architecture 14
2.6 Exemplary Eureka architecture including different cluster [61] 15
2.7 Coordination between master and slave nodes via Apache Zookeeper [2] 19
2.8 Topology in Apache Storm [2] 19
2.9 The Apache Flink 20
2.10 The architecture of Eclipse BIRT [51] 21
2.11 The BIRT viewer [51] 22
2.12 Example Grafana dashboard with different panels [18] 23
2.13 MVCC concept in CouchDB [49] 25
2.14 Data representation in a MongoDB database [38] 26
2.15 Architecture of MongoDB [37] 26
2.16 Monitoring facilities of MongoDB [38] 27
2.17 Example for a graph database 28
2.18 Example schema for a InfluxDB 29
2.19 Architecture of hawkBit 32
2.20 Sensing Entities of the OGC SensorThings API 34
2.21 Kafka Architecture [63] 36
2.22 IoT Reference Model [48] 38

4.1 The cloud platform architecture with generic building blocks 57

6.1 The cloud platform architecture with specific technology 82
6.2 Proposed architecture of Eclipse Kuksa 83

List of Tables

2.1 An overview of some existing IoT platforms 8

5.1 Eclipse Kapua (cf. Sec. 2.2.1). 59
5.2 Eclipse Ditto (cf. Sec. 2.3.1). 60
5.3 Eclipse Mihini (cf. Sec. 2.3.2). 60
5.4 Eureka (cf. Sec. 2.3.4). 61
5.5 Apache Camel (cf. Sec. 2.3.5). 61
5.6 Spring Integration (cf. Sec. 2.3.6). 62
5.7 Apache Storm (cf. Sec. 2.4.1). 63
5.8 Apache Flink (cf. Sec. 2.4.2). 64
5.9 Eclipse BIRT (cf. Sec. 2.4.3). 65
5.10 Grafana (cf. Sec. 2.4.4). 66
5.11 Apache CouchDB (cf. Sec. 2.5.1). 67
5.12 MongoDB (cf. Sec. 2.5.2). 68
5.13 Neo4j (cf. Sec. 2.5.3). 68
5.14 InfluxDB (cf. Sec. 2.5.4). 69
5.15 ArangoDB (cf. Sec. 2.5.5). 69
5.16 OrientDB (cf. Sec. 2.5.6). 70
5.17 Couchbase (cf. Sec. 2.5.7). 70
5.18 Eclipse hawkBit (cf. Sec. 2.6.1). 71
5.19 Eclipse Leshan (cf. Sec. 2.6.2). 72
5.20 Eclipse Wakaama (cf. Sec. 2.6.3). 72
5.21 Eclipse Vorto (cf. Sec. 2.6.4). 73
5.22 OGC SensorThings API (cf. Sec. 2.6.5). 73
5.23 Eclipse Hono (cf. Sec. 2.7.1). 74
5.24 Apache Kafka (cf. Sec. 2.7.2). 75
5.25 Eclipse Mosquito (cf. Sec. 2.7.3). 75
5.26 Evaluation criteria scores 76
5.27 The evaluation of IoT Cloud Platform technologies 76
5.28 The evaluation of Service Integration technologies 77
5.29 Application of evaluation criteria to the data analytics and visualization technologies. 78
5.30 Application of the evaluation criteria on Database Management System (DBMS) technologies 79
5.31 Application of the evaluation criteria on multi-model databases 79
5.32 The evaluation of technologies for device management. 79
5.33 Application of the evaluation criteria on message gateway technologies 80

Summary

List of Abbreviations

ACID	atomicity, consistency, isolation und durability
AMQP	Advanced Message Queuing Protocol
ANSI	American National Standards Institute
API	Application Programming Interface
BSON	Binary JSON
CoAP	Constrained Application Protocol
CoRE	Constrained RESTful Environments
CRUD	Create, Read, Update, and Delete
DBMS	Database Management System
DMF	Device Management Federation
DPWS	Devices Profile for Web Services
DSL	Domain-Specific Language
DTLS	Datagram Transport Layer Security
ECU	Electronic Control Unit
EPL	Eclipse Public License
EXI	Efficient XML Interchange
HMI	Human-Machine Interface
HTTP	Hypertext Transfer Protocol
IDS	Intrusion Detection System
IETF	Internet Engineering Task Force
IoT	Internet of Things
IIRA	Industrial Internet Reference Architecture
ISO	International Organization for Standardization
JDBC	Java Database Connectivity
JMS	Java Messaging Service
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LWM2M	Lightweight M2M
M2M	Machine-to-Machine
MQTT	Message Queue Telemetry Transport
MQTT-SN	MQTT-Sensors Network
MSA	Microservice Architecture
MVCC	Multi-Version Concurrency Control
NFC	Near Field Communication
OEM	Original Equipment Manufacturer
OGC	Open Geospatial Consortium
OMA-DM	Open Mobile Alliance-Device Management
OMA	Open Mobile Alliance
OTA	over-the-air
P2P	Peer-to-Peer

PaaS	Platform as a Service
POSIX	Portable Operating System Interface
QoS	Quality of Service
RPC	Remote Procedure Call
REST	Representational State Transfer
SASL	Simple Authentication and Security Layer
SQL	Structured Query Language
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TSDB	Time Series Database
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
WSAN	Wireless Sensor and Actuator Network
WSN	Wireless Sensor Network
WWW	World Wide Web
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol

1 Introduction

The purpose of this document is the specification of the IoT Cloud Platform architecture including a secured management and distribution of vehicle data as well as the integration of external services and applications. Such an IoT Cloud Platform is expected to scale both horizontally to support the large number of devices connected and vertically to address the variety of IoT solutions [12]. Due to the heterogeneity of application domains, also the requirements that IoT systems should comply with differ among each other [27]. However, an IoT architecture comprises core building blocks that are applied across different domains and regardless of the use case or connected *Things*. Depending on the according reference architecture, such building blocks differ in its naming and granularity level. For example, Figure 1.1 shows typically core features for the Eclipse open source software stack for an IoT Cloud Platform, while Figure 1.2 depict the reference architecture according to [14] with facilities for device management, data storage and management, visualization, analytics, marketplace, and stream processing. Similar to them, the IoT reference architecture by Intel [21] includes building blocks for data storage, security, analytics, management, or application integration (cf. Figure 1.3). The Microsoft Azure IoT reference architecture in Figure 1.4 comprises multiple components for device registration and discovery, data collection, transformation, analytics, business logic, and visualizations [34]. Among other things, the Industrial Internet Reference Architecture (IIRA) [31] exhibit the following key system characteristics: security, integrability, connectivity, data management, and analytics. In [19], the authors present a taxonomy of required IoT components from a high level perspective including components for device management, data storage and analytics, and visualization.

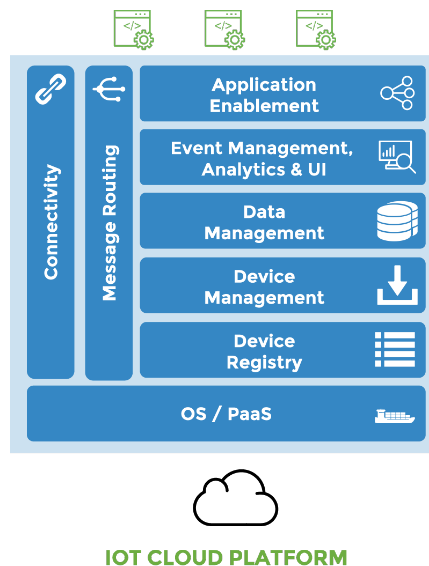

Figure 1.1: Software stack for an IoT Cloud Platform [12]

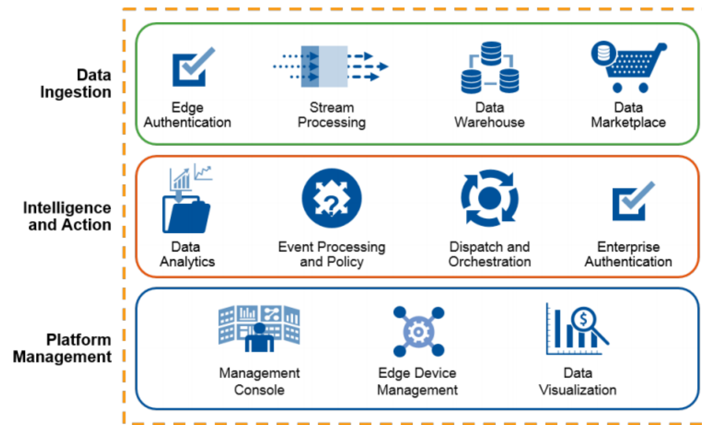

Figure 1.2: IoT Platform reference architecture according to Gartner Inc. [14]

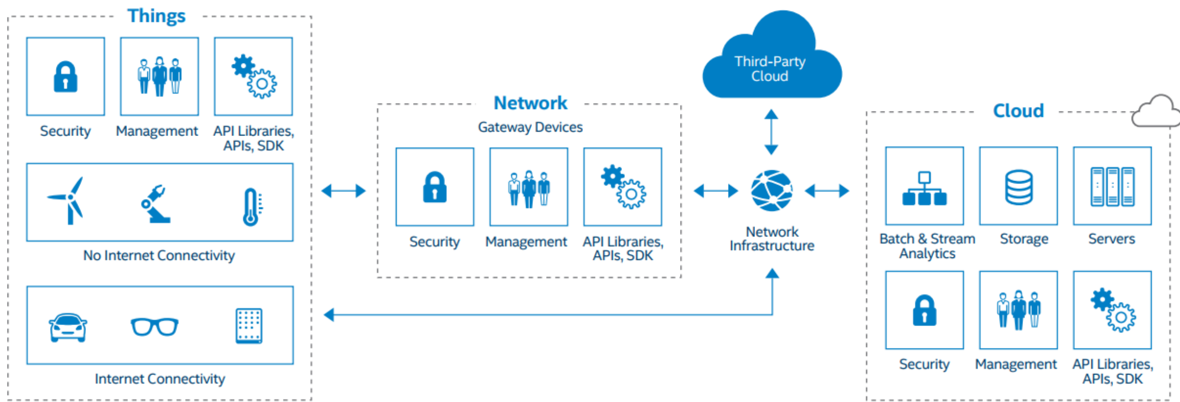

Figure 1.3: End-to-end IoT architecture according to Intel [21]

In general, an IoT architecture consists of the following building blocks according to the different reference architectures:

- Message gateway: A component for sending and receiving data to and from an arbitrary amount of (constrained) devices via different kind of protocols. As this component is the central point of interaction with the cloud back-end, the message gateway transforms data after ingress to events and act as broker by redirecting the events to other components for further processing.
- Data storage and management: A component for persisting data within different types of databases.
- Data analytic and visualization: Components for analyzing existing data including big data analyses and visualizing data in a suitable and valuable way.
- Device management: A component for device management allows to authenticate, configure and control, monitor, maintain, and update devices.

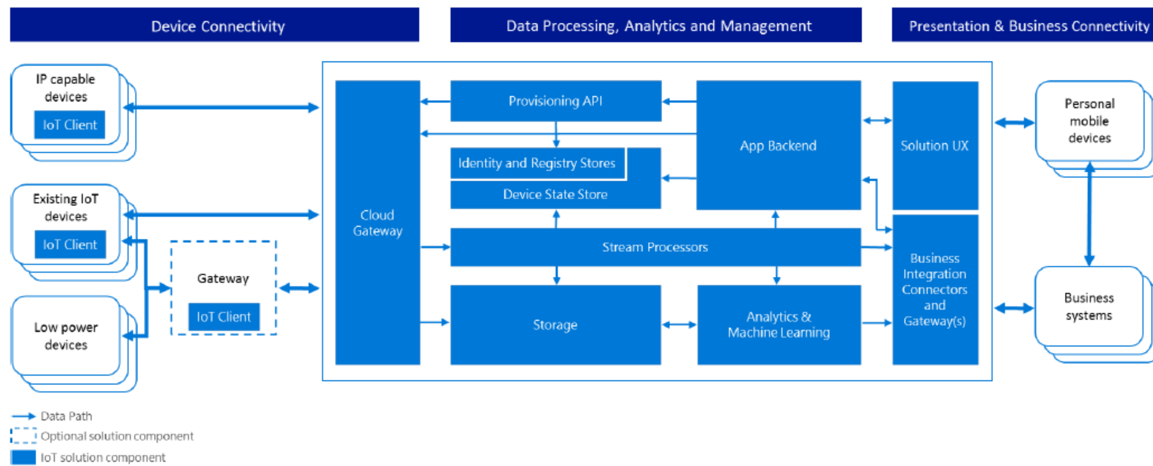

Figure 1.4: Microsoft Azure IoT reference architecture [34]

- Application and service integration: Components that support the development and provision of applications and services within the cloud back-end. For example, the digital representation of a physical asset allows to apply data analysis, machine learning, or monitoring.
- Security: Components that realize authentication, authorization, privacy, and a secured communication.

As the resulting architecture specification will be the foundation for the subsequent development activities, the architecture has to be well defined with respect to (i) compatibility to the In-Car platform of WP1 as well as the connectivity solution and network middleware of WP2; (ii) storing, processing, and analyzing a large amount of heterogeneous vehicle data via appropriate big data concepts [14]; (iii) a holistic security approach for the reception, storage, management, and distribution of vehicle data; (iv) adaptability to the end user needs like different storage mechanisms or security levels.

1.1 Document Structure

The rest of this document is structured as follows: Chapter 2 investigates existing IoT building blocks by means of protocols, components, platforms, and ecosystem approaches in the form of a state of the art analysis. Chapter 3 then introduces different user stories from the automotive domain which depict the usage of the APPSTACLE ecosystem from different points of view. Based on the state of the art analysis and the automotive user stories, Chapter 4 specifies the main building blocks of the IoT Cloud Platform and their data flow among each other. Afterward, Chapter 5 evaluates the results from the previous chapters regarding their applicability to the APPSTACLE infrastructure and proposes concrete technologies for the according building blocks of the cloud architecture. Finally, Chapter 6 concludes this document with a discussion on the resulting architecture specification and their compatibility with the other work packages as well as an outlook on future work within APPSTACLE.

2 State of the art

As already stated in Chapter 1, IoT architectures are applied across various domains with different domain-specific requirements. However, regardless of the according domain, an IoT architecture comprises common building blocks. Thus, this chapter presents a state of the art analysis for the different common building blocks of the cloud ecosystem like communication, data management, big data analyses, data visualization, and device management. For each aspect, we provide a general description on the purpose, requirements, and challenges. In particular, we investigate different existing technologies for the implementation of the specific aspects. Due to the large variety of cloud-related technologies, it is not feasible to review each existing solution. Thus, our state of the art analysis only regard established projects and further relies on the following inclusion criteria:

- *Documentation:* The application of the respective technology should be documented and, if necessary, demonstrated via appropriate examples.
- *Java-based:* The technology should be based on Java to ensure a seamless integration with the other work packages and third parties.
- *Open-source:* According to [13], open source software is a dominant provider of critical infrastructure technology for the general software industry as the open model of development and royalty-free distribution has proven to be an effective way to build production quality software. Furthermore, open-source projects maintained by a community typically exhibit a better quality as well as security and allows for more customizability and flexibility. Also technology maintenance and updates is given for a rather long life cycle. Thus, the architecture should rely on open-source technology.

2.1 Communication Protocol

As communication protocols have been described in numerous existing studies, e. g. in [1, 25], we merely summarize the most important protocols for the understanding of this chapter.

2.1.1 REST

Representational State Transfer (REST) [16] denotes an architectural style for distributed hypermedia and web service systems. It introduces resources as time dependent mappings to a set of information entities. A resource may be referenced by an Uniform Resource Identifier (URI) like `http://www.example.com/resource`. Additionally, REST defines a generic interface to resources, consisting of methods like `GET` for requesting and `POST` for creating resource representations. Within the World Wide Web (WWW), client applications, like browsers or mobile apps, may invoke these methods, typically via the Hypertext Transfer Protocol (HTTP). Extensible Markup Language (XML) or JavaScript Object Notation (JSON) [8] are widely used to encode the transmitted representation.

2.1.2 CoAP

With the advent of the IoT vision, and Machine-to-Machine (M2M) communication being one of its main drivers [4], specialized communication techniques, which consider possible resource limitations of M2M devices, e. g. less available energy and lower network connectivity [66], are needed. Hence, the Constrained Application Protocol (CoAP) was designed for M2M applications involving constrained networks and nodes like microcontrollers [46]. It supports the realization of Constrained RESTful Environments (CoRE), i. e. the communication between M2M nodes via a subset of REST. CoAP thus provides an URI addressing scheme for M2M resources and methods like `GET` and `POST` to interact with them. However, in contrast to REST, CoAP doesn't rely on HTTP but defines its own User Datagram Protocol (UDP)-based protocol to achieve reduction of communication overhead by a compact transmission format [5]. In particular, CoAP provide more compact binary headers and reduces the set of methods that can be used (GET, POST, PUT, DELETE) [64]. Additionally, CoAP payloads might be encoded with the Efficient XML Interchange (EXI) to further lower packet fragmentation rates in constrained networks as well as the use of computational resources for payload parsing [45].

Several open source implementations of CoAP exist. For Java the Eclipse Californium project[1] provides libraries for client- and server-implementations under the Eclipse Public License (EPL).

2.1.3 AMQP

Advanced Message Queuing Protocol (AMQP) is a standardized, binary communication protocol for message oriented middleware, i.e. it supports the application layer communication between clients and message brokers. The standard originated from the financial industry but is designed to cover a broad range of middleware-related problems. It is currently hosted at OASIS [40] and also approved as official International Organization for Standardization (ISO) standard.

Overall, the standard consists of six parts (core, types, transport, messaging, transactions, and security), which define a type system for encoding messages as well as a linking protocol that defines communication patterns.

Some open source messaging middleware exists that understands AMQP. One is the Apache 2.0 licensed message broker Apache Qpid™[43], another is the Apache 2.0 licensed Apache ActiveMQ™[50] message broker.

2.1.4 LWM2M

Lightweight M2M (LWM2M) [41] is a protocol for device management in the IoT. It provides capabilities to monitor devices, change device parameters as well as updating the firmware of a device. The protocol has been defined by the Open Mobile Alliance (OMA) and is the successor of Open Mobile Alliance-Device Management (OMA-DM). LWM2M defines interfaces for [41, p. 15]

1. Bootstrap (initialize objects on a LWM2M client in order to communicate with a LWM2M server)
2. Client Registration (registering a client at a server)

[1] `https://www.eclipse.org/californium/`

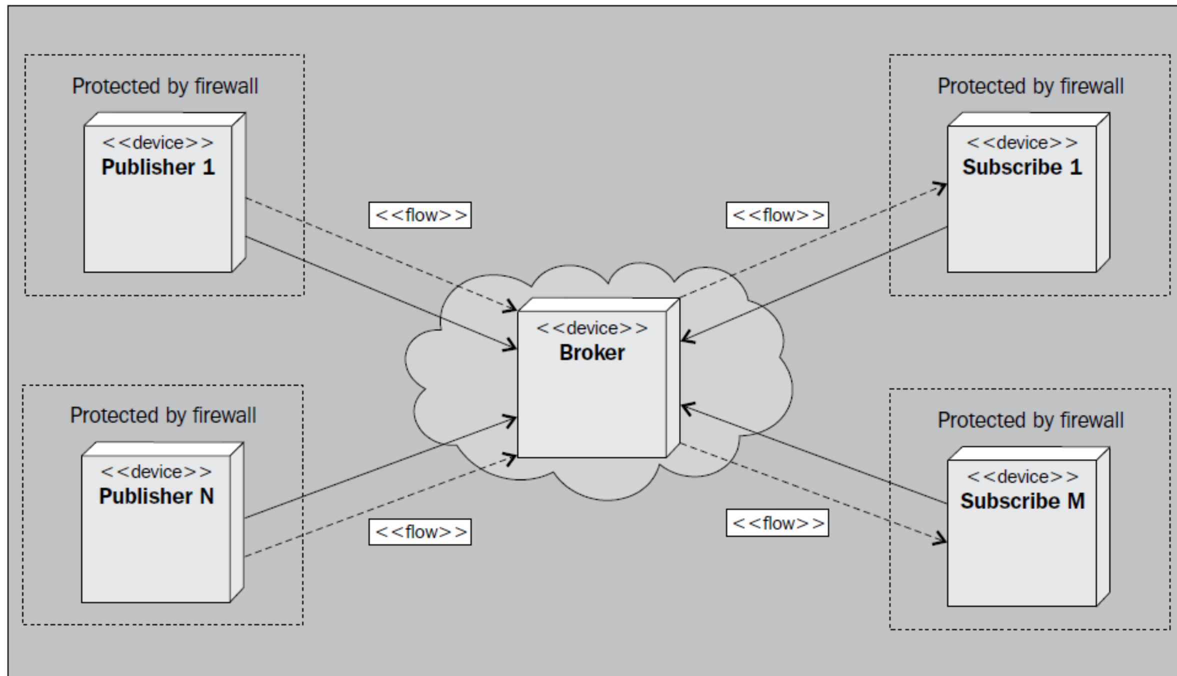

Figure 2.1: Publish/subscribe pattern for MQTT []

3. Device Management and Service Enablement (reading and manipulating parameters of a device and activating services of a device)
4. Information Reporting (registering to get notified about parameter changes at a device)

Several Open Source implementations exists of which Eclipse Leshan is implemented in Java and released under the EPL [56] (cf. Section 2.6.2). A corresponding implementation for embedded systems in C is provided by the Eclipse Wakaama project [60] (cf. Section 2.6.3).

2.1.5 MQTT

MQTT is a publish-subscribe messaging protocol submitted to the OASIS by IBM and now standardized by the ISO [22]. The protocol is based on TCP/IP and is designed for small devices with limited network connection. As shown in Figure 2.1, the publish subscribe nature of the protocol requires that devices and clients communicate with a broker, which is responsible for receiving messages and sending them to interested parties. This implies that the protocol provides means to connect and disconnect to a broker, subscribe and unsubscribe to certain topics, and to publish messages to a certain topic.

MQTT comprises three types of actors for the publish/subscribe pattern [64]:

- Publisher: The role of the publisher is to connect to the message broker and publish content
- Subscriber: They connect to the same message broker and subscribe to content that they are interested in

- Message broker: This makes sure that the published content is relayed to interested subscribers

The content is identified by a topic. When publishing content, the publisher can choose whether the content should be retained by the server or not. If retained, each subscriber will receive the latest published value directly when subscribing [64]. To build applications on top of MQTT several implementations for clients as well as for message broker exists (cf. Section 2.7.3). A Java and other client-implementations are provided by the Eclipse Paho project [58] under the EPL. A message broker especially dedicated to MQTT and available under the EPL is Mosquitto from the Eclipse Mosquitto project [57].

According to [64], the MQTT protocol itself does not consider security. For example, the support for user authentication in MQTT relies on plain text and is thus weak. However, MQTT can also be used over an encrypted connection using Secure Sockets Layer (SSL)/Transport Layer Security (TLS) to circumvent the security problems. In general, the developers have to consider security for MQTT themselves.

MQTT-Sensors Network (MQTT-SN) denotes a more lightweight version of MQTT and thus has some advantages, especially for embedded devices. Among other things, MQTT-SN does not require TCP/IP stack. Instead of this, UDP or simple link protocols can be used, which are more resource-efficient.

2.1.6 XMPP

While the MQTT is limited to the publish/subscribe pattern, Extensible Messaging and Presence Protocol (XMPP)[2] supports, apart from the publish/subscribe pattern, other communication patterns such as point-to-point request/response or asynchronous messaging [64]. Initially designed for instant messaging applications and thus capable of multi-party chats as well as voice and video calls, XMPP has been extended to a communications protocol for message-oriented middleware based on XML and standardized by the Internet Engineering Task Force (IETF). It enables the near-real-time exchange of structured yet extensible data between any two or more network entities. All of the existing XMPP servers, clients, and programming libraries support the key features of an IM system, such as one-to-one and multi-party messaging, presence subscriptions and notifications, and contact lists. This wealth of code enables developers to easily build new applications in a secure and scalable way.

XMPP was originally developed in the Jabber open-source community and thus offers several key advantages over such services:

- Open: The XMPP protocols are free, open, public, and easily understandable
- Standard: The XMPP specifications were published as RFC 3920 and RFC 3921, RFC 6120, RFC 6121, and RFC 7622.
- Proven: Hundreds of developers are working on these technologies
- Decentralized: The architecture of the XMPP network is similar to email; as a result, anyone can run their own XMPP server
- Secure: Any XMPP server may be isolated from the public network (e. g. on a company intranet) and robust security using Simple Authentication and Security Layer (SASL) and TLS, which have been built into the core XMPP specifications.

[2] `https://xmpp.org/`

2.2 IoT Cloud Platform

IoT Cloud platform is a paradigm to integrate two concepts of Internet of Things and Cloud computing. An IoT Cloud platform provides features of ubiquitous IoT applications and services that are applicable in a wide range of domains. It enables us to access anything, anywhere at any time, without concerning about issues such as storage capacities, operational performance, processing capacity etc.

IoT and Cloud Computing can be integrated by either (1) bringing the Cloud to IoT to improve the IoT technological issues such as limited storage, small processing capabilities, and energy capacities; or (2) moving IoT to the Cloud to leverage IoT capabilities and offers as pay-per-use services on the Cloud called "Cloud of Things".

As a part of an IoT Cloud platform, an IoT middleware is a software layer or a set of layers between applications and underlying technological layer (communication, processing, and sensing) [15, 47]. It is an important software building block for hiding the complexity related to the diversities and specifities of distributed smart devices as well as the networking environment.

By establishing an automotive IoT Cloud platform, we confront several challenges. For example, Cloud service systems in the vehicle may be subjected to real-time requirements, while sensor data or controlling actuators have to be accessible via the web from devices outside the vehicle.

In addition, it is necessary to redesign the software engineering process as the traditional automotive software development is different from Cloud-based software development in the sense of the lifecycle, release duration etc. Finally, assuring safety, security and reliability is a major concern in this domain.

Since each domain exhibit different requirements towards the functionality of an IoT Cloud platform, platforms are tailored to the needs of distinct user and application groups such as government, healthcare, communication, transportation, or manufacturing. This results in considerable number of platforms which are available. While there is no standardized architecture for such IoT Cloud platforms, platforms exhibit different aspects at different level of granularity. According to [44], device management, data management, and application management are covered by most platforms, whereas heterogeneity management, analytic, and visualization lack of support.

Table 2.1 provides a list of major IoT Cloud platforms and middleware along with a short description of them. An more detailed overview on existing platforms is given in [9, 36, 44].

Table 2.1: An overview of some existing IoT platforms

Name	Description
AFME	A middleware solution designed for wireless pervasive systems to tackle the performance and code management issues associated with executing agents only on mobile devices
ASPIRE	An open source and scalable IoT middleware along with tools to development and deployment of RFID-based applications
AWS IoT	Amazon Web Services IoT (AWS IoT) is a managed cloud platform for the IoT

CarrIoTs	A middleware especially for M2M communication that focuses on cost effectiveness, scalability, and ease of use
CHOReOS	A Quality of Service (QoS)-aware middleware for enabling large-scale choreographies and heterogeneous services in IoT
DIMMER	A service IoT platform aiming at involving different stakeholders to increase the energy efficiency of a city
EMMA	An adaptation of Java message service (JMS) for mobile ad hoc environments that is used for multiparty video communication systems
Echelon	An IoT platform with a full suite of chips, stacks, modules, interfaces, and management software for developing devices and Peer-to-Peer (P2P) communities
FIWARE	A platform for the future internet that provide a novel service infrastructure built of reusable components
GREEN	A run-time and configurable event-based middleware to support pervasive computing applications
GSN	It uses virtual sensors to control processing priority, management of resources, and stored data. Using declarative specifications, virtual sensors can be deployed and reconfigured in GSN containers at runtime
Hermes	A middleware created for large-scale distributed applications that uses a scalable routing algorithm and fault-tolerance mechanisms that can tolerate different failures
HyCache	An application-level caching middleware for distributed file systems. Distributed file systems are deployed on top of HyCache on all data node
IoTCloud	A platform that controls and manages the sensors and messages over the cloud online with the different modules, i. e. controller, message broker, and sensors
KASOM	A knowledge-aware middleware for pervasive embedded networks, especially for Wireless Sensor and Actuator Networks (WSANs)
LinkSmart (Hydra)	A middleware for ambient intelligence services and systems to provide syntactical and semantic level interoperability
Mires	A message-oriented middleware for Wireless Sensor Networks (WSNs) using the publish/subscribe pattern
MobIoT	A thing-based service-oriented middleware that offers discovery, composition, and access functionalities for IoT applications
MUSIC	A middleware to provide a self-adaptive component-based architecture to support the building of systems in ubiquitous environment, where dynamic changes may occur in service providers and service consumers contexts
Octopus	An open source and extensible system to support data management and programming models for IoT
OM2M	The Eclipse project OM2M denotes an implementation of the oneM2M and SmartM2M standard. As such, it provides a horizontal M2M service platform for developing applications and services independently of the underlying network in terms of heterogeneous devices

OpenIoT	An open-source middleware for IoT applications by means of a service model, which is available at a cloud environment that can be transparently accessed and configured by users
Paraimpu	A tool that allows users to compose and mash-up things to react with events, sensors, or social activities
PRISMA	A middleware with a high-level and standardized interface for data access to support interoperability of the heterogeneous network technologies
PSWare	A real-time middleware to support composite events
RUNES	A middleware for large-scale distributed heterogeneous network of embedded systems
SENSEI	A middleware to develop an architecture including a context model, context services, actuation tasks, and dynamic service composition of both primitive and advanced services for the real-world internet
SenseWrap	A middleware that combines the zeroconf protocols with hardware abstraction using virtual sensors to discover sensor-hosted services
Sensation	A middleware developed for WSN applications, and designed to provide support for different sensors, network infrastructures, and middleware technologies
SensorBus	A message-oriented middleware that allows exchange of more than one communication mechanism among sensor nodes
SiteWhere	An open source IoT platform that provides a system that facilitates the ingestion, storage, processing, and integration of device data
SOCRADES	A middleware that abstracts physical things as services using devices profile for Devices Profile for Web Services (DPWS). Its architecture consists of a layer for application services and a layer for device services
Steam	A middleware service, designed for the mobile computing domain to address dynamic reconfiguration problem of the network, scalability of a system, and the real-time delivery of events
TinyDDS	A middleware enables interoperability between WSNs as well as access networks and provides programming language and protocol interoperability
UBiROAD	A semantic middleware for context-aware smart road environments that deals with interoperability between in-car and roadside heterogeneous devices
UBiWARE	A middleware for the creation of autonomous, complex, flexible, and extendible industrial systems to support automatic resource discovery, monitoring, composition, invocation, and execution of different applications
Xively (Cosm)	A Platform as a Service (PaaS) that provides middleware services to create products and solutions for IoT
Kaa	A PaaS that provides middleware services to create products and solutions for IoT (`https://www.kaaproject.org/`)

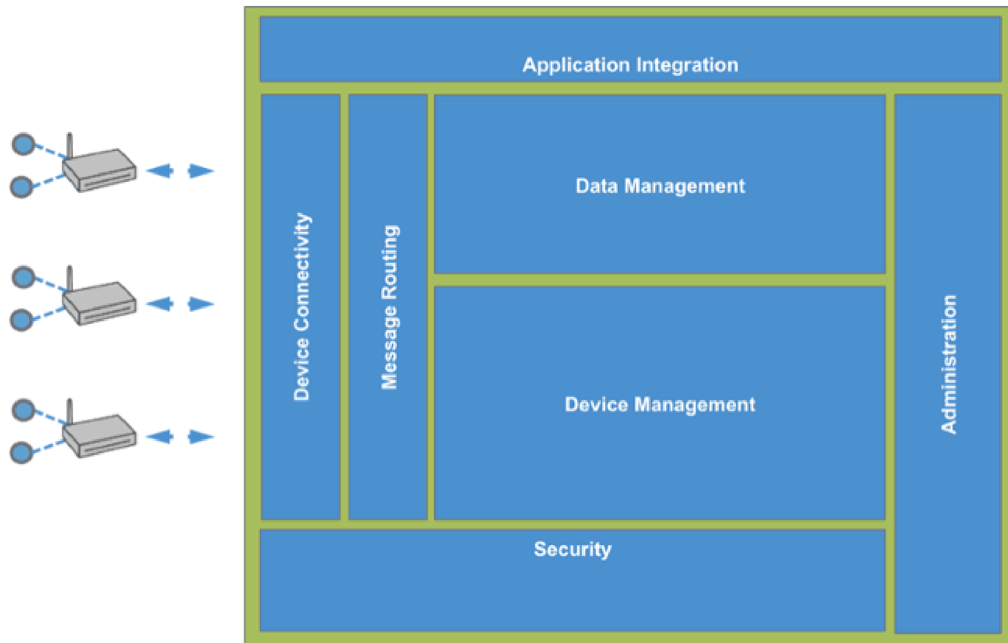

Figure 2.2: Architecture of Eclipse Kapua [55]

2.2.1 Eclipse Kapua

The Eclipse project Kapua[3] is a modular designed IoT platform for the comprehensive management of edge IoT nodes including their connectivity, configuration, and application life cycle [55]. Further features of Kapua are a web-based administration console for device and data management operations (cf. Figure 2.3), a REST Application Programming Interface (API) for application integration, and aggregated real-time data streams [55]. Kapua does not focus on a specific domain and thus can be seen as a generic IoT Cloud platform. Figure 2.2 depict the functional architecture of Eclipse Kapua [55]:

- Device Connectivity: Within Kapua, the device connectivity realize the authentication and authorization of connections and further maintain a device registry. Therefore, a multi-protocol message broker manages the connection to the devices via MQTT as well as AMQP and Websockets for application integration.
- Message Routing: The component for the message routing (cf. Section 2.7) allows for flexible handling of message streams.
- Device Management: Kapua supports device management via the MQTT protocol. In particular, Kapua allows to manage (i) the device configuration; (ii) the device service including start and stop operations; (iii) device applications by means of installing, updating, and removing applications; (iv) the execution of remote OS commands; and (v) device attributes and resources.
- Data Management: For a persistent storage of the telemetry data, Kapua make use of a NoSQL data storage and index sent data by timestamp, topic, and originating asset.

[3] http://projects.eclipse.org/projects/iot.kapua

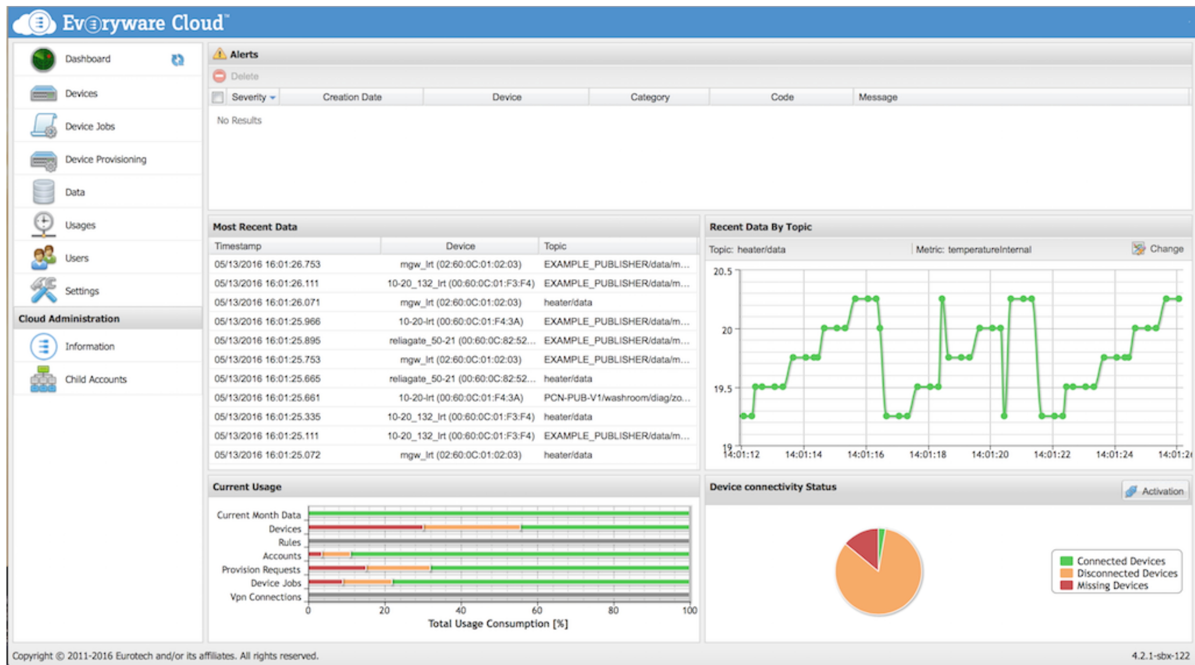

Figure 2.3: Web-based administration console for all device and data management operations [55]

Furthermore, data management in Kapua is also responsible for the data registry which maintains topics and metrics.

- Security: The security module in Kapua includes a hierarchical access control structure for the secured authentication of users through provided credentials or SSL. A Role Based Access Control further realize a authorization concept.
- Application Integration: The integration of application in Kapua is enabled via a REST-based web service API. While the API exposes all the platform functionality such as device and data management, it also provide the option to bypass the message broker and directly route commands from applications to devices.

The latest available version of Kapua is 0.2 under the EPL. Currently, the developers migrating the platform towards a Microservice Architecture (MSA) and plan to reuse existing Eclipse projects for the appropriate functionalities like Eclipse Hono for the message routing (cf. Section 2.7).

2.3 Application and Service Integration

This section aims to provide an overview on existing solutions for supporting the development and provision of applications or services from the cloud to the vehicle. Such an application store and service provision has to consider different kinds of vehicles, personalized settings, and other context-specific data. Thus, application and service should be provided in a variable way, depending on the certain context. To realize distinct functionality via domain-specific services, also a digital representation of certain devices is important.

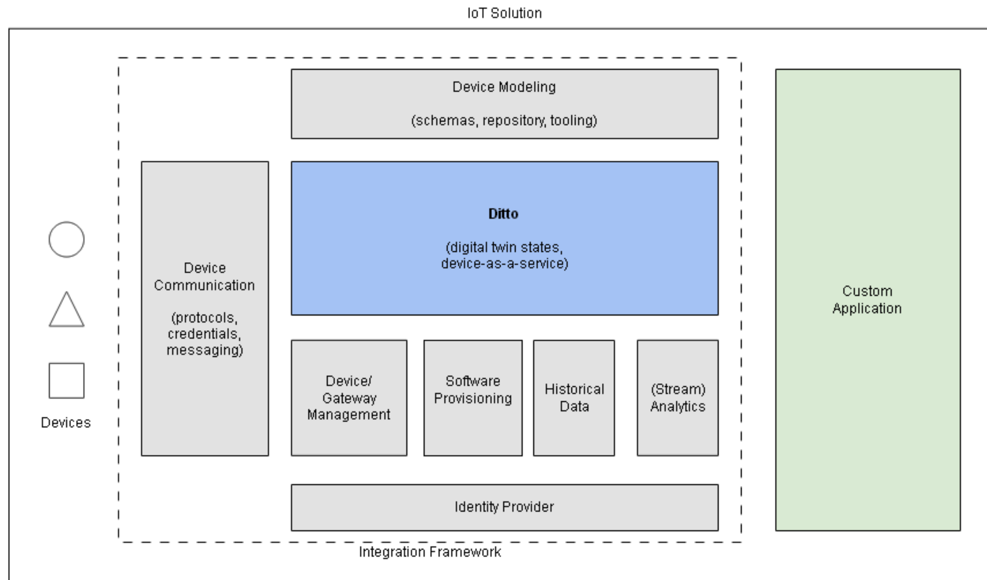

Figure 2.4: Eclipse Ditto in the overall Eclipse IoT landscape [52]

2.3.1 Eclipse Ditto

The IoT is an amalgamation of two worlds of software development: classical embedded software development and web development. Both worlds have a very different culture. In embedded software development often the focus is on reliability and safety whereas in web application the development speed and feature richness is key. Integrating both worlds introduces several problems. In order to cope with them, the digital twin metaphor is proposed, which means that for each physical device that is to be connected to the internet a corresponding digital representation exists. The digital twin offers the possibility to access and alter the state of a device in a controlled manner.

Eclipse Ditto provides a platform to realize the digital twin metaphor. It provides functionality to realize

- Device as a Service: higher level API to access devices
- State management of digital twins including notification of state changes
- Digital Twin Management: provides meta-data based support to search and select digital twins

The API of Eclipse Ditto is realized as REST-API. By that a backend-less realization of IoT applications is possible. IoT application developers can concentrate on the business logic and user experience without the hassle to integrate different protocols and device types. The architectural setting of Eclipse Ditto looks like in Figure 2.4.

2.3.2 Mihini

The Mihini project[4] provides an application environment for the Things in the IoT. It is based on the Lua API[5] which is a lightweight scripting language for developing portable M2M applications. It implements e. g. functionalities like I/O management, data management, device management, application management and application settings management. It runs on top of Linux systems like the Raspberry Pi. Figure 2.5 depicts the Mihini architecture.

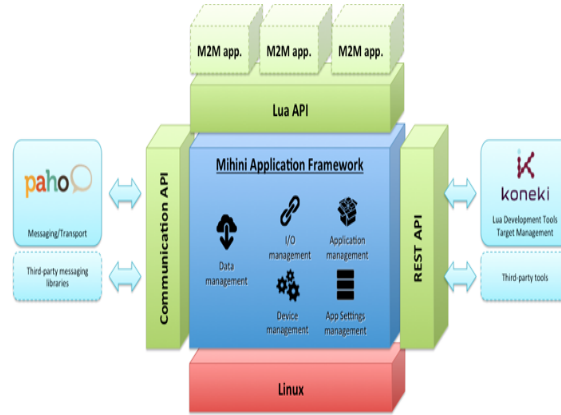

Figure 2.5: Mihini Architecture

2.3.3 Appstore

Eclipse Marketplace

The Eclipse Foundation operates a website –called Eclipse Marketplace– that provides Eclipse-based solutions. It allows solution providers to specify a P2 repository for their solution. P2 repositories combine the artefacts (bundles), the meta data and eclipse features. Within Eclipse marketplace, Eclipse users have a central catalogue to find Eclipse solutions but the install process is still not tightly integrated with the Eclipse workspace.

Marketplace Client is a rich client solution for installing solutions listed on Eclipse Marketplace directly from an Eclipse Installation. It provides the tight install integration between the Eclipse workspace and Eclipse Marketplace, plus other third party solution listings.

2.3.4 Eureka

The service registry and middle-tier load-balancer framework Eureka[6] is available at the Netflix Open Source Software Center[7] under Apache License 2.0. It represents a REST-based services discovery with consideration for load balancing and fail-over of middle-tier servers (instance/server/host level) [61]. In particular, Eureka allows microservices to register with an unique name and an appropriate URL, which can be afterward used to identify the service to

[4]https://eclipse.org/proposals/technology.mihini/
[5]https://www.lua.org/
[6]https://github.com/Netflix/eureka
[7]https://netflix.github.io/

make remote calls although the services may be deployed on different instances (cf. Figure 2.6). Thereby, Eureka follows a client/server approach:

The Eureka server runs as independent service within a MSA and act as central communication point for the Eureka clients. It manages the service dynamically, i. e. services that don't send a heartbeat within a certain interval are considered as unavailable and thus are removed from registry. Furthermore, the Eureka servers have a built-in resiliency to prevent a large scale outage [61]. As per convention each Eureka server is also a client, several Eureka server can be merged to a cluster.

An Eureka client is further divided into an *Application Client* and *Application Service* as shown in Figure 2.6. The former one represents services which rely on the communication with other services, while the latter one offers their functionality to other services and is responsible to register at the Eureka Server via REST. However, an Eureka client can be also both an *Application Client* and an *Application Service*. To handle failure of one or more Eureka servers, Eureka clients have the registry cache information in them, which is frequently updated via a poll mechanism. Servo, an interface for exposing and publishing application metrics in Java, is used within Eureka for performance monitoring and alerting.

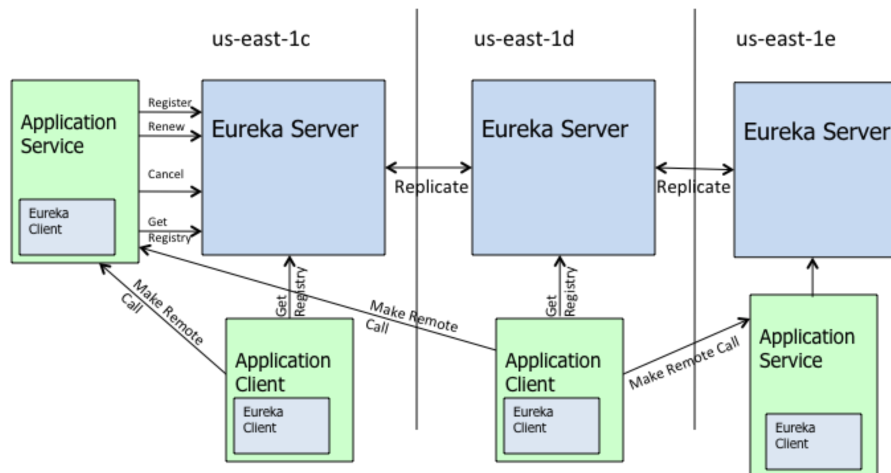

Figure 2.6: Exemplary Eureka architecture including different cluster [61]

2.3.5 Apache Camel

Apache Camel is an open source framework for system integration that aims at simplifying the integration process[8]. Its source code, which is written in Java, is available at GitHub and

[8]http://camel.apache.org

released under version 2.0 of the Apache License[9].

Apache Camel enables the integration of various systems by passing messages[10]. In this regard, it is agnostic regarding the data that is transferred within a message. The framework does not require that data is converted in any form. Apache Camel's core is a *routing and mediation engine* that transfers messages on routes between the systems to integrate. The routes are configured based on *enterprise integration patterns* and defined using one of multiple Domain-Specific Languages (DSLs). The concept of enterprise integration patterns has been introduced by Gregor Hohpe and Bobby Woolf in the similarly titled book in 2003. While investigating existing integrated systems, they derived a set of 65 messaging patterns that are grouped into the following classes[11]:

Messaging endpoints: Connecting a system to a message channel.

Message construction: Wrapping the data to exchange within a message.

Messaging channels: Writing information such that in can be read by a consumer.

Message routing: Routing a message to its destination based on filters.

System management: Operating a messaging system.

Message transformation: Transforming the content of a message between data formats.

These patterns represent proven solutions to existing problems regarding system integration. Apache Camel uses DSLs based on Java, Scala, Groovy, and XML to specify rules for message routing. Using existing programming languages allows to leverage existing development environments, such as the Eclipse IDE. The framework provides support for more than 80 protocols and data types as well as more than 150 data type converters.

Apache Camel claims to have a modular and extensible architecture, which should allow to easily add missing functionality, such as data type converters. In addition, it aims at reducing the effort for testing a self-build Apache Camel application. The so-called *TestKit*, which for example allows to mock endpoints, supports application development.

2.3.6 Spring Integration

Spring Integration is a framework for integration Spring-based applications that relies on message passing[12]. It heavily relies on the Enterprise Integration Patterns described in Sect. 2.3.5. For this purpose, Spring Integration extends the Spring programming model. It allows to integrate Spring applications providing diverse functionality, while maintaing the business logic in an isolated fashion. The source code of project is hosted on GitHub[13].

Spring Integration follows the concepts of *pipes*, responsible for transporting messages, and *filters*, producing and consuming messages. It comprises the following general components[14]:

[9] `https://github.com/apache/camel`

[10] `https://manning-content.s3.amazonaws.com/download/f/737b721-0f60-4ba9-bb1f-7a27c4a4532b/chapter1sample.pdf`

[11] `http://www.enterpriseintegrationpatterns.com/patterns/messaging/index.html`

[12] `http://projects.spring.io/spring-integration/`

[13] `https://github.com/spring-projects/spring-integration`

[14] `https://docs.spring.io/spring-integration/docs/4.3.12.RELEASE/reference/html/overview.html`

Message: It is composed of a header and the payload. The header contains meta information related to the payload, such as time-of-creation.

Message Channel: It refers to the concept of a *pipe*, as mentioned above. Messages, which have been produced, are send via a channel to the corresponding consumers.

Message Endpoint: It refers to the concept of a *filter*, as mentioned above. Endpoints connect the individual applications to the message channels.

A message may contain any kind of information, provided as arbitrary objects. The framework comprises a set of message channel implementations, including the following Java classes:

`PublishSubscribeChannel`: It broadcasts a message that has been sent by one provider to all of its subscribers. This channel type is often used for event propagation.

`QueueChannel`: It considers only a single provider as well as single consumer. Unless its capacity is not exceeded, messages are stored within a channel queue. Messages are send to the consumer according to the *first in, first out* (FIFO) principle.

`PriorityChannel`: It orders messages according to a priority, in contrast to `QueueChannel`. This information is provided as an additional header attribute.

`RendezvousChannel`: It refers to a synchronous implementation of `QueueChannel`. Hence, the execution of the sender is blocked, until the message has been received.

`DirectChannel`: It corresponds to the `PublishSubcribeChannel`, while sending messages of a publisher only to a single subscriber.

Note that the list above is not complete. Spring Integration distinguishes several types of message endpoints. These types overlap with concepts from the enterprise integration patterns:

Transformer: It transforms a given message, e.g., converting the format of the payload.

Filter: It decides, whether a message is accepted by an output channel or not.

Router: It decides, to which message channel a message is sent next.

Splitter: It splits a given message, usually equipped with a composed payload, into multiple messages and sends them to a set of output channels.

Service Activator: It connects a service entity to the messaging system, given an input channel and zero or more output channels.

Channel Adapter: It connects a message channel to a specific inbound or outbound transport or system.

Note that Spring Integration may provide multiple implementations of the different message endpoint types. Each message endpoint implementation has different semantics and exposes different behavior.

2.4 Data Analytic and Visualization

The advancing digitalization in the automotive domain, driven by more and more connected devices and an increasing number of sensors [33], leads to a large amount of heterogeneous data from various stakeholder [14]. For example, already in 2013, an average about 480 TB data from more than 26 million cars has been collected by the automotive manufacturers [10]. Beside the large data volume, the data exhibit a high variety [3] in terms of the used data format as well as the sources they come from, e. g. different types of sensors. Depending on the use case, it may be further necessary to process data in real-time or nearly real-time. These requirements in terms of volume, variety, and velocity [28], commonly characterized as big data, makes the processing and analysis of vehicle data a challenge [65]. However, as big data provides the basis for optimized product and after-sale service as well as personalized online service, e. g. smart services like a car insurance adaptation based on driving analysis or a smart traffic navigation based on aggregated traffic information, this section reviews appropriate data processing and analytic technologies. In particular, we focus on big data processing software approaches that allows to deal with large data sets in real-time, while supporting various data scheme.

2.4.1 Apache Storm

Apache Storm[15] denotes an open source distributed stream processing computation framework available under the Apache License, Version 2.0. In contrast to Apache Hadoop[16], Apache Storm focus on real-time processing rather than batch processing [2]. Therefore, Apache Storm offers two types of nodes to realize a cluster: The master node (run by a daemon called *Nimbus*) distributes code around the cluster, assign tasks to machines, and monitor for failures. On the other hand, worker nodes (run by a daemon called *Supervisor*) are responsible to start and stop worker processes, which contain the processing logic. Although both node types are stateless, a fault-tolerant, stable, and parallel manipulation of data is realized in conjunction with Apache Zookeeper[17] [2]. Figure 2.7 depict the coordination between master and slave nodes via Apache Zookeeper.

Within Apache Storm, the real-time manipulation of big data is established through a topology. In particular, a topology act as data transformation pipeline by providing a graph of computation (directed acyclic graph) where each node contains processing logic, while links between nodes represent streams and indicate how data should be passed around between nodes [2]. The vertices in such a topology can be either a spout or a bolt (cf. Figure 2.8). While a spout represents a data source, a bolt consumes any number of input streams, does some processing, and possibly emits new streams [2]. Apache Storm supports different kinds of data source like various Java Messaging Service (JMS) provider or Apache Kafka[18].

Vertical scalability is realized through the addition of worker nodes. Furthermore, Apache Storm also provides a load-balancing between similar nodes. As shown in [6], Apache Storm performs quite well and at a low latency with real-world big data benchmarks. Further features of Apache Storm are:

- Continuous Computation: Allows to distribute results whilst processing other data.

[15] http://storm.apache.org/
[16] http://hadoop.apache.org/
[17] https://zookeeper.apache.org/
[18] https://kafka.apache.org/

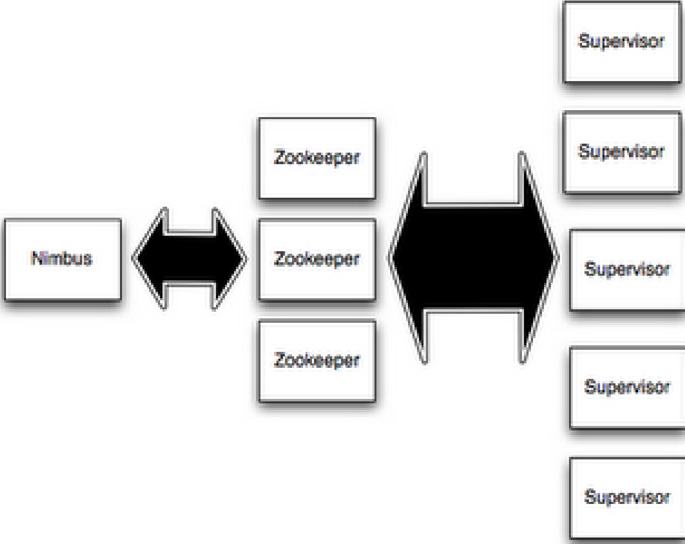

Figure 2.7: Coordination between master and slave nodes via Apache Zookeeper [2]

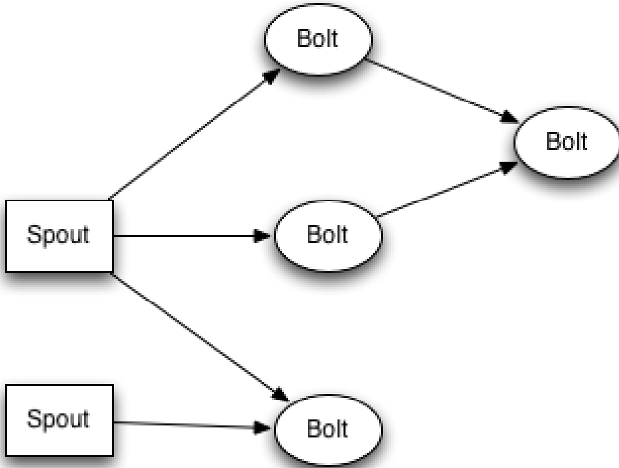

Figure 2.8: Topology in Apache Storm [2]

- Integration: Apache Storm allows to use any type of programming and seamless integrate with existing database technologies [29].
- Parallelization: Remote Procedure Calls (RPCs) allows to parallelize computation-intensive requests.

2.4.2 Apache Flink

Apache Flink[19] is an open source stream processing framework. The core is a distributed streaming dataflow engine written in Java and Scala. Flink executes arbitrary dataflow programs in a data-parallel and pipelined manner. Flink's pipelined runtime system enables the execution of bulk/batch and stream processing programs.

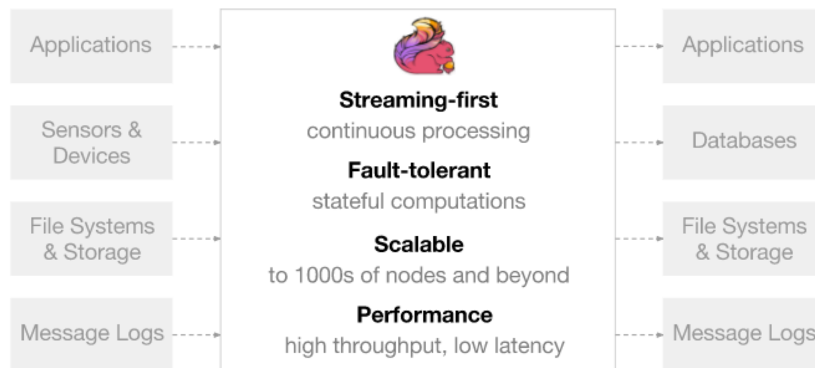

Figure 2.9: The Apache Flink

Apache Flink can:

- provide results that are accurate, even in the case of out-of-order or late-arriving data
- be stateful and fault-tolerant and can seamlessly recover from failures while maintaining exactly-once application state
- perform at large scale, running on thousands of nodes with very good throughput and latency characteristics

Apache Flink focusses explicitly on stream processing by introducing a streaming execution model[20]. The processing of bulk data is considered as a special case of stream processing, internally treating corresponding data sets as finite streams. This is a major difference to alternative solutions. Apache Flink provides the DataStream API[21] for stream processing and the DataSet API[22] for batch processing. While applications that process streaming data can only be written in Java or Scala, applications processing batch data can also be written in Python[23].

[19] https://flink.apache.org
[20] https://flink.apache.org/introduction.html
[21] https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/datastream_api.html
[22] https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/batch/index.html
[23] https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/batch/python.html

Apache Flink applications define transformations on data that originates from data sources and is returned to data sinks. In addition to defining custom transformations, Apache Flink does provide libraries regarding complex event processing, machine learning and graph processing. It furthermore provides compatbility to Apache Storm. Applications are executed on a distributed streaming dataflow engine, which processes data streams in an event-at-a-time fashion.

Apache Flink claims to have a superior performance regarding latency and throughput in comparison to Apache Storm[24] and Apache Spark[25]. One aspect that supports this assertion is a custom memory management that stores data in pre-allocated memory segments using a binary representation instead of storing Java objects on the heap[26]. Maintaining these off-heap memory segments idependently allows Apache Flink to reduce the load on the garbage collector. Furthermore, the memory overhead for storing Java objects is reduced considerably.

2.4.3 Eclipse BIRT

Eclipse BIRT[27] is an open source platform (available under the Eclipse Public License (EPL)) for the visualization of data and generation of reports. It is designed for the usage in rich client or web applications and therefore consist of two main components as shown in Figure 2.10:

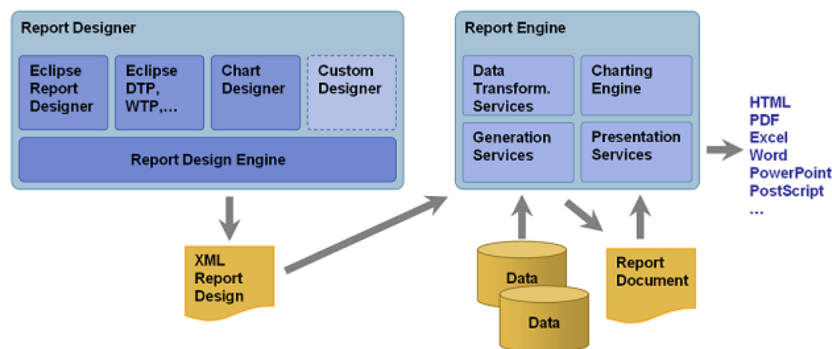

Figure 2.10: The architecture of Eclipse BIRT [51]

The visual report designer is based on Eclipse and allows to create BIRT Designs. Thus, such designs are created in a specific Eclipse perspective and stored in an open XML format. Among other things, the BIRT designer offers a component-based model for reusing design elements, integrates different types of reports and layouts, e. g. various chart types such as a bar or a pie chart, and supports a wide range of data sources like Java Database Connectivity (JDBC), Web Services, XML, MongoDB, or Excel [11]. The expression builder in BIRT further enables conditional report processing, i. e. flexible representation of information depending on the available data.

The runtime component (*Report Engine*) can be deployed to any Java environment and allows to generate reports, for example HTML, PDF, XLS, DOC, etc., based on the XML Report Design file [51]. The *Charting Engine* further provides facilities to generate charts, which can be used either standalone or embedded within BIRT reports. An additional viewer

[24] https://flink.apache.org/introduction.html#features-why-flink
[25] https://jobs.zalando.com/tech/blog/apache-showdown-flink-vs.-spark/?gh_src=4n3gxh1
[26] https://flink.apache.org/news/2015/05/11/Juggling-with-Bits-and-Bytes.html
[27] https://eclipse.org/birt/

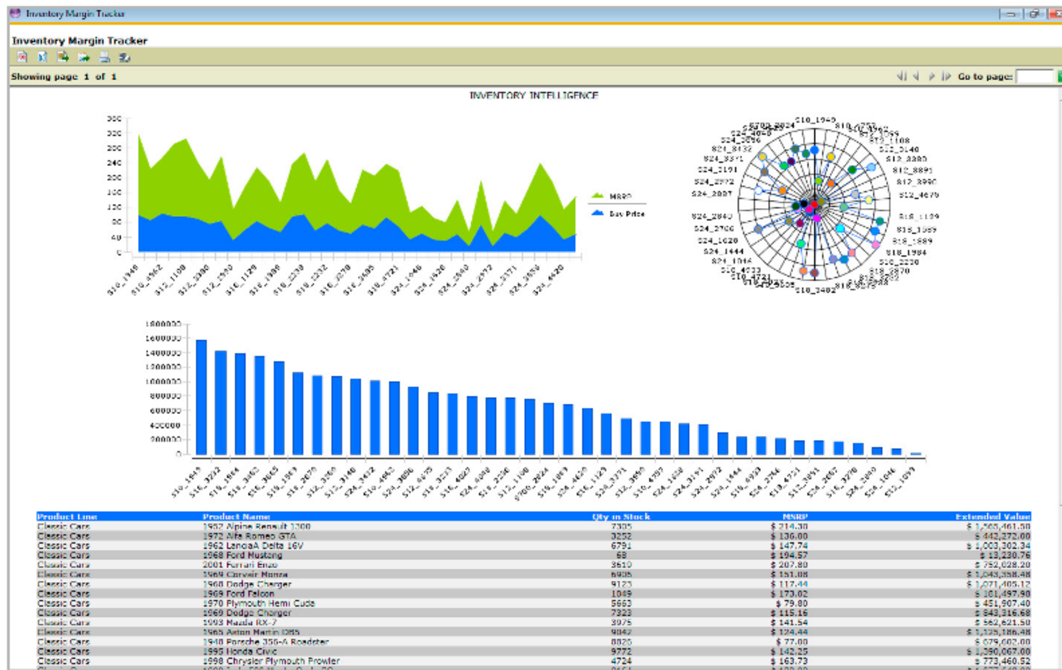

Figure 2.11: The BIRT viewer [51]

Plug-in enables the preview of reports (cf. Figure 2.11) within Eclipse and is also available as standalone Java EE application [11].

2.4.4 Grafana

Grafana[28] is an open source metric analytics & visualization platform published under the Apache License, Version 2.0. The main purpose of Grafana is the visualization of time series data via customized dashboards [18]. Grafana offers a specific query editor for each supported data source, e. g. InfluxDB or Graphite. In this way, features and capabilities of the according data source can be exposed. Different data sources can also be combined within a single dashboard.

Panels are the main building block of a dashboard as they allow to explore the existing data by processing the extracted data into a suitable visualization via several configuration parameters. They are comprised within the different rows of a dashboard as shown in Figure 2.12. The following panel types are currently available in Grafana:[29]

- Graph: The graph-based visualization is the main panel type in Grafana. It provides various options for customizing the visualization for the sake of the user. For example, the x-axis can either group the data by time, by series, or by history ranges, while data can be plotted as bar charts, line graphs, or points. In addition, graph panels allows to display tooltips, legends, or thresholds and panels can be customized regarding the

[28] https://grafana.com/

[29] The following website demonstrate the respective panels and their application: http://play.grafana.org/dashboard/db/grafana-play-home?orgId=1

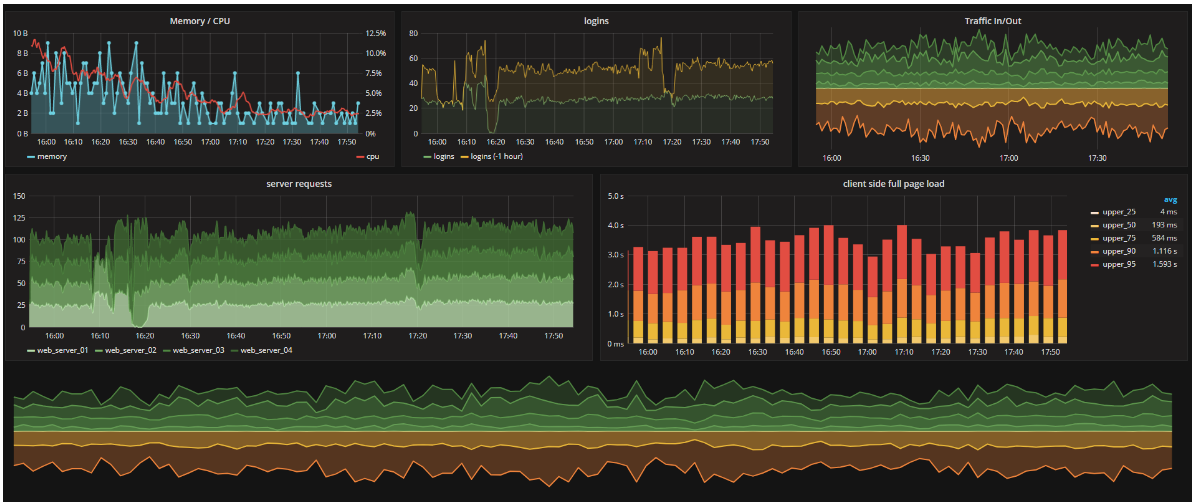

Figure 2.12: Example Grafana dashboard with different panels [18]

time range (the dashboard time range is used by default), axis label, data units, scale, rendering etc.

- Singlestat: Via a singlestat panel, the main summary stat of a single series, e. g. the maximum, minimum, average, or sum of values, can be displayed. Thresholds can be used to color the panel background according to the actual number. The representation of a single stat can be a simple number with pre- and postfix text, but also a coloring, spark lines, gauges, or a value to text mapping.

- Table: As the name suggest, the table panel allows one to display values by means of columns and rows in a very flexible way. In its simplest form, time series are displayed per column. However, it is also possible to aggregate values within rows by time series or to display annotations.

- Heatmap: A heatmap panels is similar to a histogram, but it features value distributions over specific time ranges to detect trends etc. Therefore, the frequency how often a value occurs for a specific point in time is colored proportional to the overall numbers of values.

- Text: The text panel allows to provide information and descriptions in markdown, HTML, or plain text.

In general, Grafana provides a great flexibility for the customization of dashboards and navigation. Through snapshots, panels can also be shared among stakeholder with no access to the dashboard. A snapshot denotes a static and interactive JSON document with all currently viewed data encoded [18]. Additional features of Grafana are templating for dynamic dashboards, annotations, export/import via JSON, or an alerting engine.

Authentication is realized in Grafana on different levels of privileges and in variety ways including its own integrated database, from an external Structured Query Language (SQL) server, or from an external LDAP server [18]. The access to dashboards can be tailored to users and organizations.

2.5 Data Storage and Management

One of the most important aspect for the realization of the cloud infrastructure is the storage and management of vehicle data by an appropriate DBMS. Although persisting and managing data is a central point in every software architecture, especially the various stakeholder and large amount of heterogeneous data in the automotive domain requires a well-defined data storage and management concept. Thus, the DBMS has to consider, among others, the following requirements and challenges:

- *Big Data*: The DBMS has to be Big Data compatible, i. e. the DBMS needs to be aware of large, complex, and unstructured data sets.
- *Data Management*: The DBMS has to provide data access to different user groups and on different levels of granularity. This requires a holistic access management and the ability to aggregate data of different hierarchies.
- *Performance*: The DBMS should process data in appropriate velocity by supporting certain approaches like in-memory storage.
- *Scalability*: The DBMS should provide mechanisms for horizontal scalability, i. e. increasing capacity of the database on demand.
- *Security*: As the exchanged vehicle may contain sensitive data, the DBMS should include different levels of security.

Over the last decades, different types of DBMSs, each following a specific approach for storing and managing data, have been evolved. Relational databases, for example MySQL, are matured and provide good protection for transactions as well as flexible mechanism for accessing data.

With the advent of Web 2.0, NoSQL databases have become more and more popular as they are suitable to handle large data sets in an efficient way and allow the storage of heterogeneous data, e. g. in terms of different JSON structures, in a generic manner [42]. Therefore, NoSQL databases rely on a non-relational model for data storage. Examples for such models are key-store, column-oriented, or document-based [3]. Various surveys and evaluations for NoSQL databases are available in literature. For example, Han et al. [20] conducted a survey on NoSQL databases and their advantages and limitations for the application in the cloud, while Lourenço et al. present in [32] quality attributes for choosing a suitable NoSQL database. Document-oriented databases like MongoDB are suitable for storing unstructured data. Graph databases, for example Neo4j, focus on the performant linkage of data to realize an efficient querying on large data sets. A Time Series Database (TSDB) like InfluxDB again is designed for the massive storage of time series data such as data collected at regular intervals from sensors etc.

As relational DBMSs already exists for a long time and thus have been often described and evaluated in literature [24], we focus in the following on the most popular and matured implementations for the other types of DBMSs.

A multi-model database [26], such as ArangoDB, OrientDB, or Couchbase, comprises different types of DBMSs within a single incorporated database engine.

Figure 2.13: MVCC concept in CouchDB [49]

2.5.1 Apache CouchDB

The document-oriented NoSQL database CouchDB[30] is an open source project hosted by Apache and available via Apache License, Version 2.0. Each CouchDB database comprises a set of uniquely named, semi-structured, and independent documents (indexed by their name and a *Sequence ID*). Documents can accessed via HTTP-based REST[31] for reading and updating (add, edit, delete) [49]. Each document defines its own schema and basically consists of both metadata like revision information and JSON objects for storing the data. JSON[32] objects represents field/value pairs of different data types like String, Date or even structures like an ordered list.

Due to its realization as peer-based distributed database system, CouchDB allows a bi-direction replication and synchronization of documents [49]. In this way, clients can perform offline operations on replicated documents and sync the changes later back to the database in a efficient way. Thus, CouchDB is also suitable for applications that have no guaranteed network connection. Another feature of CouchDB is the view model, which supports different views, i. e. aggregation and filtering of data, for a document. By default, querying is done via JavaScript, but there is also support for other languages like PHP or Phyton. Furthermore, views provide the basis for applying MapReduce Dean2008 operations.

As a DBMS, CouchDB provides all atomicity, consistency, isolation und durability (ACID) semantic. Thus, the database never contains partially saved or edited documents as updates on documents are either successful or fail completely [49]. When conducting an update operation, documents are first loaded, then the changes are applied, and finally saved to the database. A Multi-Version Concurrency Control (MVCC) model realizes the concurrent access to the database (cf. Figure 2.13). Thereby, the CouchDB document update model allows clients to resolve update conflicts, e. g. when two clients updating the same document at the same time. As the data access and manipulation can be extended by custom security models, CouchDB supports individualized security and validation [49].

2.5.2 MongoDB

The open source DBMS MongoDB[33] is available under GNU AGPL v3.0 License and denotes another type of NoSQL database. As such, it is designed for building and running data-driven

[30] http://couchdb.apache.org
[31] http://www.json.org/json-de.html
[32] http://www.json.org/json-de.html
[33] https://www.mongodb.org

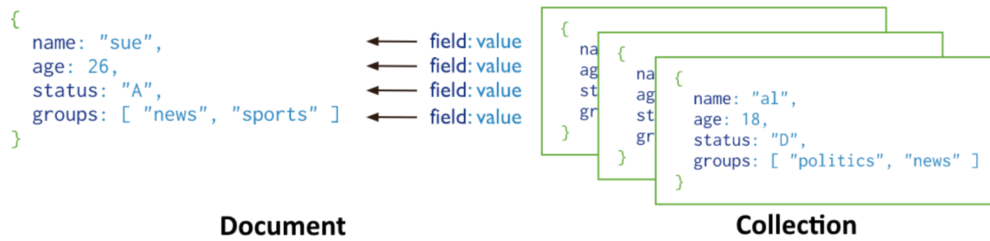

Figure 2.14: Data representation in a MongoDB database [38]

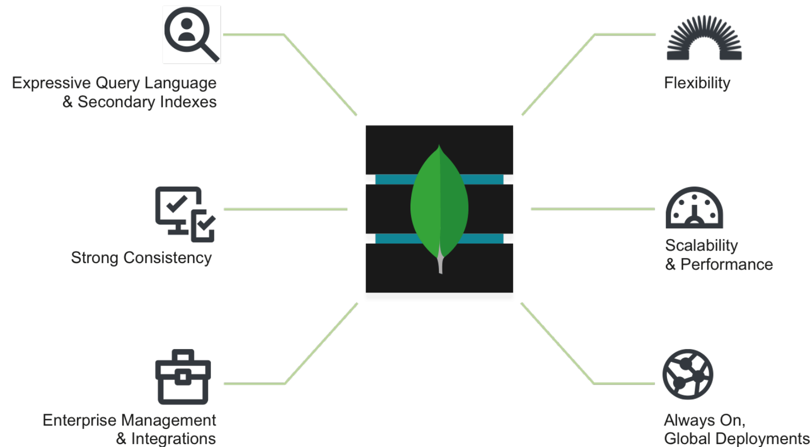

Figure 2.15: Architecture of MongoDB [37]

applications delivered as services that must be always-on, accessible from many different devices on any channel, and scaled globally to millions of users [37]. Similiar to CouchDB, MongoDB stores data in a flexible way via field/value pairs in domain-specific structured documents. Such dynamic schema allows further the validation of the document structure, value ranges, data types etc. Related documents are again aggregated in collections (collections are analogous to tables in relational databases [7]). As shown in Figure 2.14, a MongoDB database represents a set of document collections. Each document of a collection is identifiable through a 12-byte `ObjectID`, which can be either manually managed or generated by MongoDB (in case the _id field is empty). In contrast to CouchDB, MongoDB stores data in a binary representation called Binary JSON (BSON)[34], which is more lightweight and has a broader support for data types (e. g. date or floating point) than JSON [38].

MongoDb's architecture Nexus, shown in Figure 2.15, combines the critical capabilities of relational databases, i. e. expressive query languages, strong consistency, and enterprise management and integrations [37], with the innovations of NoSQL technologies like a flexible data model, scalability and performance, and availability [7].

MongoDB offers a rich query language for applying Create, Read, Update, and Delete (CRUD) operations in documents as well as data aggregation, text search, or geospatial queries [38]. Also graph traversals and complex data processing via MapReduce queries is enabled by MongoDB. Thereby, the access and manipulation of data is supported on various levels and

[34]http://bsonspec.org/

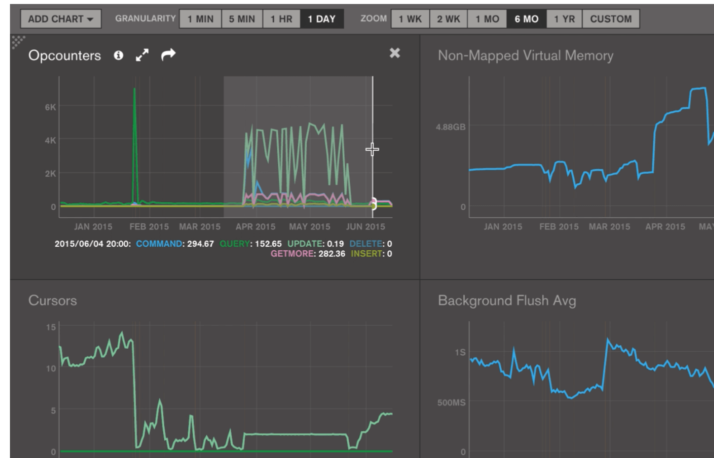

Figure 2.16: Monitoring facilities of MongoDB [38]

ways by different native drivers for programming languages and frameworks like Java, Phyton, or Apache Spark [38].

Data availability, i. e. increased data durability and protection against database downtime, is realized in MongoDB by replications. In particular, a replica set comprises multiple replica where one replica denotes the primary member and the others secondary member. If the primary one fails, one of the secondary replicas is automatically elected. To offer a flexible storage architecture, MongoDB allows multiple storage engines within a single deployment. For example, the current version 3.4 of MongoDB ships with the default WiredTiger storage engine as well as engines for the protection of highly sensitive data via encryption or In-Memory storage for high performance [37].

The interaction with the database is further supported by additional tools like MongoDB Compass[35] for schema exploration and management or MongoDB Cloud Manager[36] with facilities for comprehensive monitoring as shown in Figure 2.16. However, such tools and APIs may require a different license model, e. g. Apache License.

More features of MongoDB include Auto-Sharding for horizontal scale-out databases, a query router, ACID compliance at document level, and various security features such as authentication, access control, or encryption (TLS cf. Section 2.8).

2.5.3 Neo4j

The Java-based and open-source graph database Neo4j[37] is designed for fast management, storage, and traversal of nodes and relationships [39]. The Neo4j Community Edition is licensed under GPL v3, while the Neo4j Enterprise Edition is dual licensed under Neo4j commercial license as well as under AGPL v3. According to DB-Engines[38], Neo4j denotes one of the most popular graph database. Such database types are characterized through the generic storage of data in a graph.

[35] https://www.mongodb.com/products/compass
[36] https://www.mongodb.com/cloud/cloud-manager
[37] https://neo4j.com
[38] https://db-engines.com/de/ranking/graph+dbms

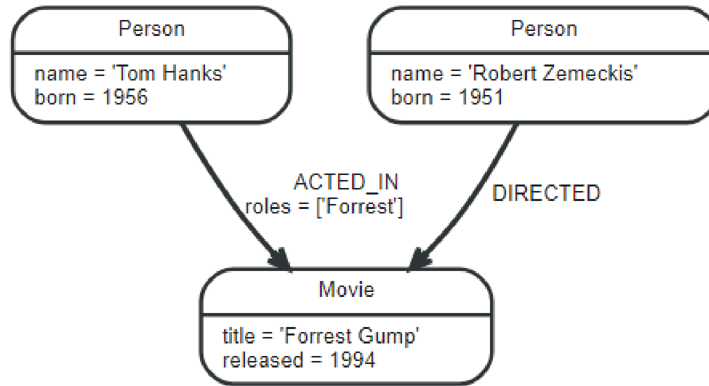

Figure 2.17: Example for a graph database

More precisely, Neo4j uses Cypher[39], a graph query language which operates on a property graph. In such a property graph, a node denotes a vertex, while a relationship denotes an edge. As shown in Figure 2.17, uniquely identifiable nodes likely represent entities along with their properties. A property is a field/value pair where the value is typed by either a String, integer, float, or Boolean. A relationship connects exactly two nodes, the source and the target node, to organize them in a specific structure like a list, a tree, a map, or a compound entity [39]. Relationships are directed, typed, and may be further described by a property. Although relationships are directed, they are equally well traversable in either direction and thus there is no need for duplicate relationships for performance reasons. Such semantically enriched relationships between nodes allows to find related data in a efficient way, which is the main distinction towards other database types [35]. To enhance the data query facilities and speed up the processing, labels can be attached to nodes in order to group them into sets. Labels also provide the basis for the definition of constraints, e. g. to ensure that property values are unique for all nodes with a specific label [39].

Neo4j provides database access via language-specific drivers. In particular, APIs for C#, Java, JavaScript, Python are shipped with Neo4j. Further features of Neo4j are ACID compliance, export of query data to JSON and XLS format, a REST API, and database indexing.

2.5.4 InfluxDB

The open source InfluxDB[40], available under MIT License, is a Time Series Database (TSDB) written in Go with the focus on high write and query loads. A TSDB is particular suitable for data which are more useful when they are aggregated, e. g. for a smart navigation application, but also tasks like monitoring, sensor data measurements, or real-time analytics benefit from time series data sets. Those tasks allows to visualize trends over time which typically require fast, high-availability storage and retrieval of time series data.

Since the root of TSDB is time, each InfluxDB database consists of a column for storing RFC3339-based[41] timestamps including the date and time. Additional columns allows to store non-time data, whereby each data value is always associated with a timestamp [62]. Figure 2.18

[39] https://github.com/opencypher/openCypher/blob/master/docs/property-graph-model.adoc
[40] https://www.influxdata.com/
[41] https://www.ietf.org/rfc/rfc3339.txt

name: census

time	butterflies	honeybees	location	scientist
2015-08-18T00:00:00Z	12	23	1	langstroth
2015-08-18T00:00:00Z	1	30	1	perpetua
2015-08-18T00:06:00Z	11	28	1	langstroth
2015-08-18T00:06:00Z	**3**	**28**	**1**	**perpetua**
2015-08-18T05:54:00Z	2	11	2	langstroth
2015-08-18T06:00:00Z	1	10	2	langstroth
2015-08-18T06:06:00Z	8	23	2	perpetua
2015-08-18T06:12:00Z	7	22	2	perpetua

Figure 2.18: Example schema for a InfluxDB [62]

depicts an example for a InfluxDB database. *Field* columns are used to store the actual data. Therefore, such a column consists of a *Field* key (`butterflies`, `honeybees`) as well as the according *Field* values, which denote concrete data of type String, float, integer, or boolean. *Fields* are not indexed and thus not suitable for queries. In contrast to *Fields*, *Tags* such as `location` and `scientist` are indexed and thus are more performant for querying. *Tags* denote a key-value pair of type String with a *Tag* key and *Tag* values. A *Measurement* acts as container for both *Fields* and *Tags* and is conceptually similar to a table in SQL. Via a *Retention Policy*, the duration for keeping data as well as the amount of *Measurement* copies for a cluster can be defined. By default, InfluxDB uses the `autogen` *Retention Policy*, which has an infinite duration and a replication factor set to one [62]. Finally, data which share the same *Measurement*, *Tag* sets, and *Retention Policy* belong to a distinct *Series*.

InfluxDB is a schemaless database and thus it is easy to add new measurements, tags, and fields at any time. With InfluxQL, InfluxDB provides an SQL-like query language to interact with the data stored in a InfluxDB database. Among other things, InfluxQL supports time-centric functions, regular expressions, arithmetic in expressions etc.[42] In general, time series data are written once and rarely updated. For this reason, InfluxDB is not a full CRUD database as it prevents some update and destroy behaviors to make the create and read operations more performant [62]. Data can be written to InfluxDB via HTTP, Transmission Control Protocol (TCP), and UDP. It has to be noted that the component for enabling horizontal scalability is not open source.

2.5.5 ArangoDB

ArangoDB is a native multi-model database system. It supports three data models which are key/value, documents and graphs with one database core. ArangoDB has its own query language - ArangoDB Query Language (AQL). AQL can be used to retrieve and modify data which are stored in ArangoDB and allows the combination of different data access patterns in a single query. ArangoDB is a NoSQL database system and syntax of AQL is different from SQL, but similar to many ways to SQL. Complex queries can be done with AQL from different data models. ArangoDB allows you to use existing or run your own data-centric microservices with Foxx. Foxx is a javascript framework. In this way, it can be defined user specific routes that

[42] cf. `https://docs.influxdata.com/influxdb/v1.3/query_language/functions/`

actual data doesn't have to be sent to the client. By this method, it improves security when the amount of data sent is reduced. ArangoDB is a distributed database supporting multiple data models. In horizontal distribution model, it provides many replication and automatic fail-over. An ArangoDB cluster consists of a number of ArangoDB instances which talk to each other over the network. These instances can be one of different roles (Agents, Coordinators, Primary, etc.).

2.5.6 OrientDB

OrientdDB has NoSQL database management system written in Java. OrientDB Community Edition is open-source and supports multi-model database. It is supporting graph, document, key-value, and object modules, but the relationship is managed as in graph databases with direct connections between records which are document, recordbytes, vertex and edge. Document is the most flexible record type in database. It can be imported or exported such as JSON format. BLOB record type was called recordbytes before OrientDB v2.2. BLOB can be loaded and stored as binary data. Vertex and edge types are used by graph database. Edge connects one vertex to another. OrientDB's Query language is similar to SQL so it can be transformed without more effort. OrientDB uses three algorithms SB-tree, Hash index, Lucene for indexing. OrientDB uses the Hazelcast Open Source project for auto-discovering of nodes, configuring the runtime cluster and synchronizing certain operations between nodes. Hazelcast provides many specific features such as queues for request and responses, storing metadata in distributed maps, distributed locks, etc. OrientDB Enterprise Edition provides Query profiler, Distributed Clustering configuration, Metrics Recording, Live Monitoring, etc. in addition to OrientDB Community Edition.

2.5.7 Couchbase

Couchbase Server is a multi-model database which natively manipulates data in key-value form or in JSON documents. Couchbase Server is a distributed and open source NoSQL database engine. Couchbase Server supports different platforms which are server and mobile. Using the SDKs, it can be written applications in the language such as Java, node.js,.NET, or others. Couchbase Server has its own query language - N1QL. N1QL syntax is similar to SQL Syntax and, when it is used, it operates JSON documents. There are many security capabilities in Couchbase Server, but Couchbase Community Edition (Open Source) not has most capabilities. It has only one capability that is built-in account managenment. Built-in account management provides password protection administration and data access. Couchbase is a distributed system and performs read/write operations, and performs queries with low latencies and high throughput with cluster structure. However, Couchbase Community Edition does not have many capabilities related to distributed system.

2.6 Device Management

According to a recently published Gartner forecast [17], 8.4 billion connected things will be in use worldwide in 2017 and it is expected that the number of devices reach 20.4 billion by 2020. Device management allows to handle the resulting heterogeneity and diversity of connected and remote devices within the IoT by exposing an open contract towards the devices with no assumption on the software stack of the device [12]. In general, device management is

responsible for several tasks like the authentication and provisioning, configuration and control, or monitoring and diagnostics. Another important aspect for the device management are software updates and maintenance. Although the amount of software in a vehicle is increasing more and more to realize complex functions like a parking assist system or a lane departure warning system, updating or maintaining the vehicle software has still to be done offline in a car service station. Such recalls typically affect a broad range of vehicles and leads to high costs for the company. For example, Daimler recently had to recall 3 million vehicles in Europe due to problems with the emissions performance, which will cost the company about 220m €.[43] However, one approach to overcome the current situation is to roll out updates over-the-air (OTA) to increase vehicle reliability and facilitate cost savings. OTA updates thus bypass the need of driving to car service stations or mass car recalls caused by defective software.

In the following, this section reviews technologies for device management with a particular focus on the establishment of OTA updates.

2.6.1 Eclipse hawkBit

Eclipse hawkBit[44] [53] is a backend framework released under the EPL to roll out software updates to constrained edge devices. With the challenge of a safe and reliable remote software update process in mind, the project aims to provide an uniform update process. This allows to avoid duplicate work for implementing mechanisms separately in each software component. Therefore, hawkBit provides a backend server that can be deployed in any cloud infrastructure (cf. Figure 2.19). It helps managing roll out campaigns, e.g. by defining deployment groups, cascading deployments, emergency stop of rollouts, and progress monitoring. Further, it offers several device management interfaces on which management messages and updates can be exchanged. However, hawkBit does not provide a client for edge devices by default. To connect certain devices, an adapter implementation that understands the protocols is needed. Those protocols in the Device Management Federation (DMF) API are AMQP, ODA-DM, and LWM2M. Also software can be delivered to edge devices through a REST API. At the cloud side, hawkBit ships a web-based UI for management purposes. Within the UI, all management functionalities are ready to use with a few clicks. In regard of the rising IoT cloud service infrastructure also interfaces for integrating hawkBit into other applications are accessible. Currently, a REST API exposes the functionality of the backend server towards other applications. For a more convenient use, hawkBit also helps managing roll out campaigns, e. g. by defining deployment groups, cascading deployments, emergency stop of rollouts, and progress monitoring. The source code for the spring based application is hosted at Github [45] and released in version 0.2.0M3. One of the features on the roadmap is the integration of Eclipse Hono as DMF provider.

2.6.2 Eclipse Leshan

Eclipse Leshan [56] is an implementation of the LWM2M protocol in Java released under the EPL. It provides implementations for the client as well as for the server side. Leshan uses Eclipse Californium to do communication via CoAP and relies on Eclipse Scandium for

[43] https://www.theguardian.com/world/2017/jul/19/mercedes-recalls-3m-diesel-cars-emissions-concerns
[44] https://eclipse.org/hawkbit
[45] https://github.com/eclipse/hawkbit

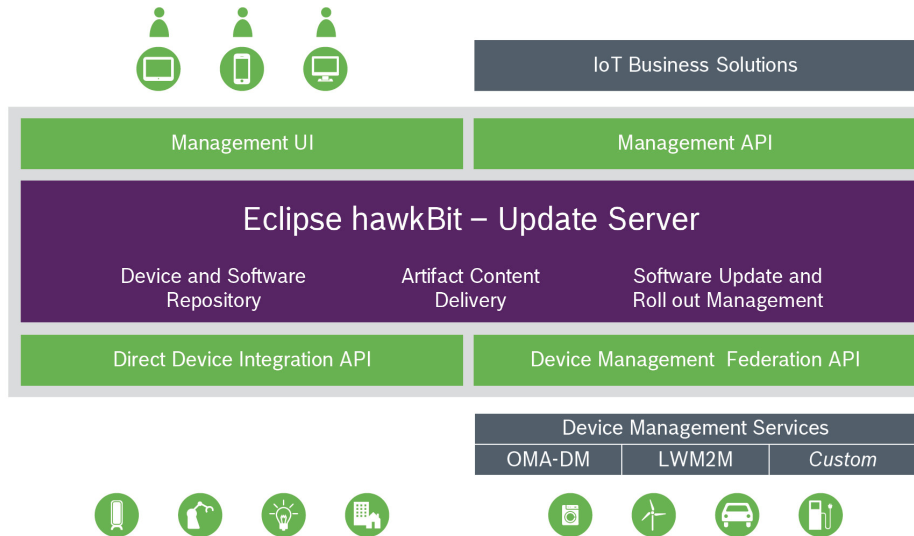

Figure 2.19: Architecture of hawkBit

Datagram Transport Layer Security (DTLS). Furthermore, there are libraries available that help people to implement their own LWM2M server and client side [56].

2.6.3 Eclipse Wakaama

Eclipse Wakaama [60] is released under the EPL and denotes another LWM2M implementation from the Eclipse IoT Working Group. In contrast to Eclipse Leshan, Wakaama is not a Java implementation for clients and servers but purely C-based and designed to be portable on Portable Operating System Interface (POSIX) compliant systems. This helps implementing systems for constrained devices which usually do not have a Java runtime environment. From the server perspective, Wakaama provides APIs to send commands to registered LWM2M Clients. On the client side, Wakaama checks received commands for syntax and access rights and then dispatches them to the relevant objects [60].

2.6.4 Eclipse Vorto

Eclipse Vorto[46] is an Eclipse IDE plugin and accompanying tools that allow to specify device capabilities in a technology agnostic way [59]. It provides a generic meta model that specifies a device as a set of function blocks – a set of status properties and action elements – comprised in a concise information model.

Besides the Vorto plugin, the project also provides a repository to share information models. This fosters collaboration between device vendors and solution providers by providing a central place to share information about devices in a machine readable way.

Having a consistent description of devices is good for communication between different parties. However, in order to actually use a device within an application real source code to connect to this device is required. Vorto provides this with a set of code generators, which can generate code for device access for various platforms such as Eclipse Smart Home, Kura etc.

[46] https://www.eclipse.org/vorto/

2.6.5 OGC SensorThings API

The OGC SensorThings API[47] denotes an open, unified, and geospatial-enabled standard to interconnect IoT devices, data, and applications over the web [48]. It is specified by the Open Geospatial Consortium (OGC) and is available as a non-proprietary, platform-independent, and perpetual royalty-free framework. From a technical perspective, the SensorThings API follows the REST principles and make use of, among other things, an efficient JSON encoding and the MQTT protocol.

In general, the OGC SensorThings API comprises two main functionalities: (i) the *Sensing* part and (ii) the *Tasking* part [48]:

- (i) The *Sensing* part describes a standard way to manage and retrieve observations and metadata from heterogeneous IoT sensor systems
- (ii) The *Tasking* part provides a standard way for parametrization of task-able IoT devices, such as sensors or actuators.

However, the *Tasking* part has not been specified yet and thus only the *Sensing* part will be in the following. The *Sensing* part and its underlying data model (cf. Figure 2.20) comprises different entities, whereby each entity has a unique identifier. By relying on REST, SensorThings entities can be added, updated, and deleted. While a `Thing` represents an object of the physical world or the information world, the `Location` entity allows to locate `Things`. Depending on the context, `Things` can be geo-referenced in different ways via different protocols. Currently, only GeoJSON is supported by SensorThings API, but there are attempts to provide multiple encoding types like IndoorGML for the future. `Observations` are responsible to measure or otherwise determine the value of a property. A `Datastream` allows to groups a collection of `Observations` that measure the same `ObservedProperty` produced by the same `Sensor` [48].

2.7 Message Gateway

Within an IoT-context, an IoT platform receives and sends different kinds of messages from and to various sources. In general, telemetry messages depict data that stem from devices, e. g. from its sensors, and flow from devices to the according components within the cloud. Telemetry data are meant to be further stored and processed in the IoT cloud platform. *Events* in this context depict telemetry messages that absolutely must be delivered. In contrast to telemetry messages, command and control refers to data that flow from the IoT cloud platform to distinct devices, for example, to control actors. Accordingly, command and control messages are dedicated to the device management component. Furthermore, some messages may have to be processed immediately, whereas other messages should be stored long term. Thus, messages have to be automatically routed to the appropriate consumer via a customer-defined routing logic. This section investigate suitable technologies for establishing a real-time message routing in the context of IoT.

2.7.1 Eclipse Hono

The Eclipse Hono project provides a platform for the scalable messaging in the IoT. It is realizing that by introducing a middleware layer between backend services and devices. Com-

[47] `https://github.com/opengeospatial/sensorthings`

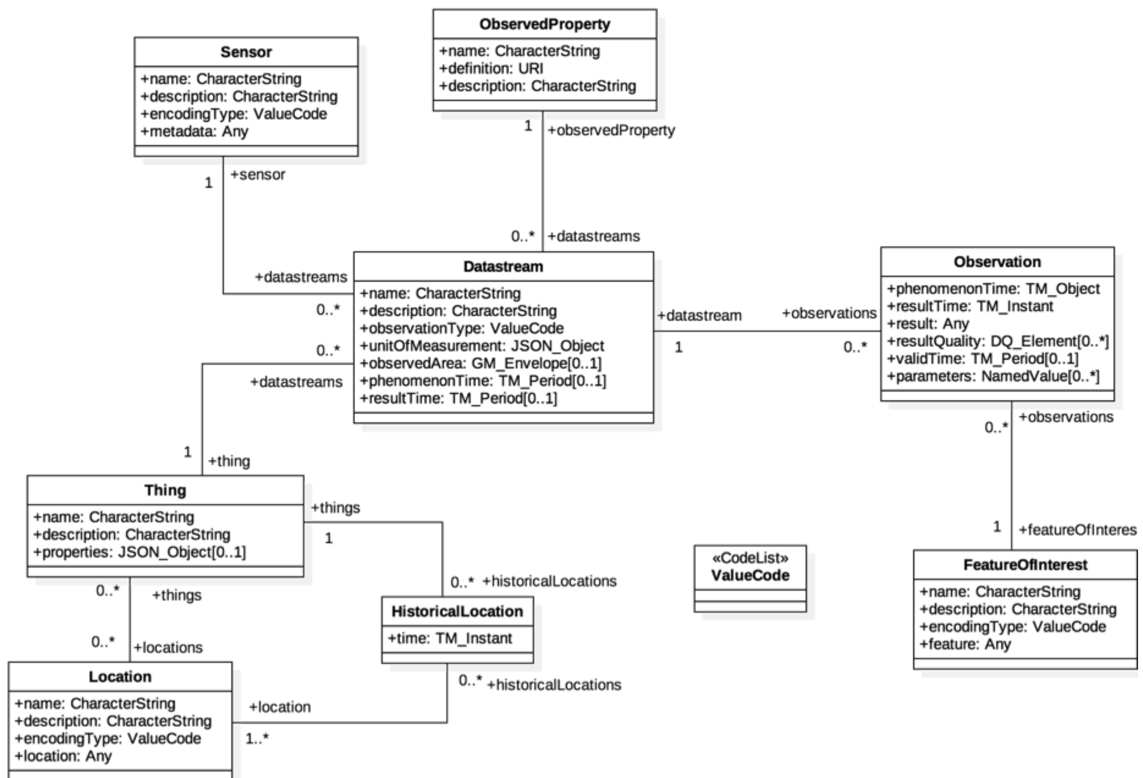

Figure 2.20: Sensing Entities of the OGC SensorThings API

munication to backend services takes place via the AMQP protocol. If devices can speak this protocol directly, they can transparently connect to the middleware. Otherwise Hono provides so called protocol adapters, which translate messages from the device protocol to AMQP. In this way, Hono's core services are decoupled from the protocols that certain applications are using.

Via AMQP 1.0 endpoints, Hono provides APIs that represent four common communication scenarios of devices in the IoT:

- Registration
- Telemetry
- Event
- Command & Control

Eclipse Hono consists of different building blocks. The first are the protocol adapters. Adapters are required to connect devices that do not speak AMQP natively. Currently, Hono comes with two protocol adapters: One for MQTT and the other for HTTP-based REST messages. Custom protocol adapters can be provided by using Hono's API.

The Hono server is the central piece to which devices can either connect directly via AMQP or via protocol adapters. Dispatch router handles the proper routing of AMQP messages within Hono between producing and consuming endpoints. The dispatch router in Hono is based on the Apache Qpid project[48] and designed with scalability in mind so that it can handle potentially connections from millions of devices. As such it do not takes ownership of messages, but rather passes AMQP packets between different endpoints. This allows a horizontal scaling to achieve reliability and responsiveness. Event and commands messages, which need a delivery guarantee, can be routed through a broker queue. The broker dispatches messages that need some delivery guarantees. Typically, such messages are mainly from the command & control API. The broker is based on the Apache ActiveMQ Artemis project[49]. While devices are connecting to the Hono server component, backend services connecting via subscribing to specific topics at the Qpid server [54].

Among the routing of messages, Hono consists of a device registry for the registration and activation of devices and the provision of credentials as well as an *Auth Server* to handle authentication and authorization of devices. By using an InfluxDB and a Grafana dashboard, Hono comes also along with some monitoring infrastructure. Due to its modular design, also other AMQP 1.0-compatible message broker than the Apache ActiveMQ Artemis can be used.

2.7.2 Apache Kafka

Apache Kafka[50] is a distributed platform that used for building real-time data pipelines and streaming apps. It is an open source, distributed, scalable, Pub/Sub messaging system that often is used as a kernel for architectures that deal with data streams. It has been designed with features such as persistent messaging, high performance, easy distribution, the ability to support multiple customers and perform real-time processing. It comprises three main elements:

[48] `https://qpid.apache.org/`
[49] `https://activemq.apache.org/artemis/`
[50] `https://kafka.apache.org/`

topics, producers and consumers. A topic is an abstract place where the messages are published; producers are processes that publish or create messages for a topic and finally consumers access a topic and process the messages posted in.

Kafka integrates information of producers and consumers without blocking the producers of the information and without letting producers know who the final consumers are. It provides an API similar to a messaging system and allows applications to consume log events in real time [63]. Figure 2.21 shows the Kafka architecture.

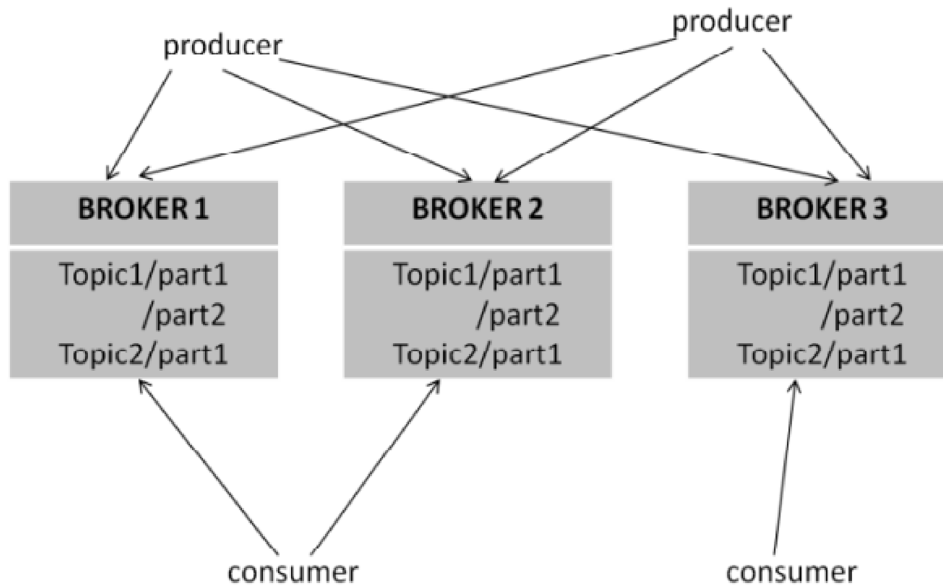

Figure 2.21: Kafka Architecture [63]

2.7.3 Eclipse Mosquitto & Paho

Eclipse Mosquitto

Eclipse Mosquitto[51] is a lightweight server implementation of the MQTT protocol available under EPL. As such an MQTT broker, it relies on the publish/subscribe model and realize the messages distribution between the client applications. Due to its efficient and simple implementation in C and the low network overhead [30], Eclipse Mosquitto is suitable for constrained devices with limited resources such as Sensors and actuators. According to [57], the current implementation of Mosquitto has an executable in the order of 120kB that consumes around 3MB RAM with 1000 clients connected.

In general, the Eclipse Mosquitto project consists of three parts [30]:

- The main mosquitto server
- The mosquitto_pub and mosquitto_sub client utilities that are one method to communicate with an MQTT server
- An MQTT client library written in C, with a C++ wrapper

[51] https://projects.eclipse.org/projects/technology.mosquitto

In addition to the MQTT specification, Mosquitto provides (i) an MQTT bridge to connect several MQTT server including other Mosquitto instances; (ii) a secured communication based on SSL/TLS; and (iii) an user authorization to restrict the access to certain MQTT topics [57].

Eclipse Paho

Eclipse Paho[52] provides another open-source client implementations of MQTT and MQTT-SN. Paho supports clients for various languages [58]:

- C: The fully fledged Paho MQTT C client is implemented in American National Standards Institute (ANSI) standard C and comprises two APIs for either synchronous or asynchronous calls.
- C++: The Paho MQTT C++ client realize an additional layer on top of the Paho C client.
- Java: For applications that run on the Java Virtual Machine (JVM), the Paho MQTT Java client can be used. Similarly to C, there is also support for synchronous as well as asynchronous
- Embedded C/C++: The Paho MQTT embedded C/C++ client is more resource-efficient than its C counterpart due to the fact that it uses very limited resources and do not rely on any particular libraries for networking, threading, or memory management. It is primarily intended for environments such as mbed[53], Arduino[54], or FreeRTOS[55].
- JavaScript: The browser-based client library for JavaScript relies on websockets for connecting to an MQTT broker.

Further supported clients are Phyton, C#, Go, and Android. Among other features, all of the previously mentioned Paho clients support SSL/TLS as well as MQTT 3.1.1.

2.8 Security

Ensuring security and data privacy protection is a significant concerns when processing data in the cloud. According to the IoT reference model in Figure 2.22 [23], security must be considered for all layers in the IoT Cloud Platform architecture. However, as security is a cross-cutting concern which affects various aspects of the APPSATCLE ecosystem on different level of granularity and thus should not be treated in isolation for each work package, a separate document will be provided that gives a holistic view on security aspects in a work package spanning manner. The work on this security document is currently in progress.

Nevertheless, as Eclipse Keti and Keycloak may be highly relevant for the architecture specification, they are described in more detail in the following.

[52] https://www.eclipse.org/paho/
[53] https://www.mbed.com
[54] https://www.arduino.cc/
[55] http://www.freertos.org

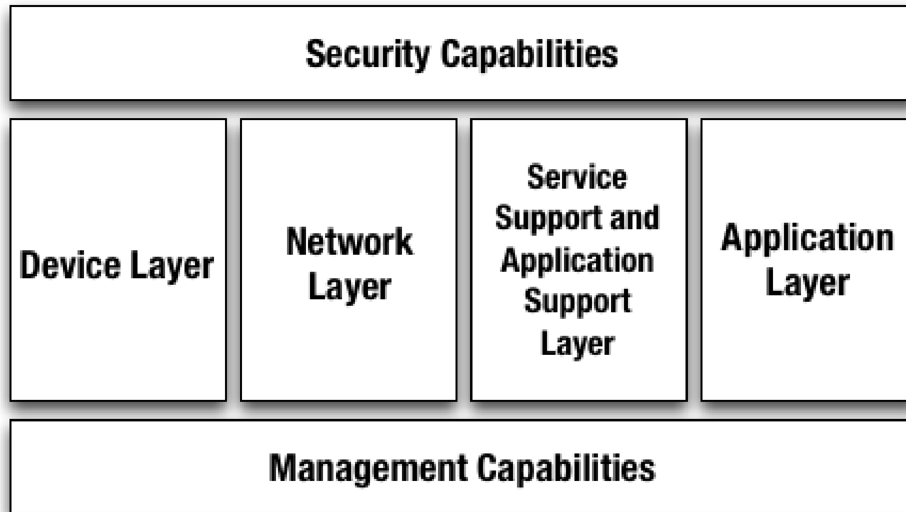

Figure 2.22: IoT Reference Model [48]

2.8.1 Eclipse Keti

Eclipse Keti is an application that allows to authorize the access of users to resources by applying the *Attribute Based Access Control* (ABAC) principle[56]. It is used to secure the communication of RESTful APIs. The project has its root in the component *ACS* that was originally a part of Predix, an industrial IoT platform developed by GE Digital. It has been made publicly available as open source in March 2016[57]. The Eclipse Keti project started in April 2017. Until now, the source code of ACS has not been moved to the new Eclipse repository[58]. Meanwhile, the development seems to continue in the old repository. The source code of ACS is released under version 2.0 of the Apache License. Eclipse Keti is also released under version 1.0 of the Eclipse Distribution License (EDL).

Whether a user can access a resource is decided based on evaluating *rules* that require attribute values as input. An attribute either refers to a user, resource or environment variable. Multiple rules can be combined into a *policy*. A policy comprises[59]:

- Target: Matches a given request to corresponding policy.
- Condition: Contains the specific authorization logic.
- Effect: Specifies the impact an authorization decision.

Eclipse Keti is a *Spring Boot* application. It consists of three major components:

- Policy evaluation: Conducts the evaluation of every authorization request.
- Policy management: Allows to update and maintain the set of policies.

[56] https://projects.eclipse.org/proposals/eclipse-keti
[57] https://github.com/predix/acs
[58] https://github.com/eclipse/keti
[59] https://www.youtube.com/watch?v=BJPN7W65c4U

- Attribute store: Maintains the user and resource permissions captured using attributes.

In addition, Eclipse Keti relies on *spring-security-oauth* to secure its own endpoints.

2.8.2 Keycloak

Keycloak is an application that allows to manage the identity of users (authentication) and their access to resources (authorization)[60]. Its source code is hosted at GitHub and released under version 2.0 of the Apache License[61]. The project has its roots in the JBoss ecosystem and its development is driven by RedHat.

Building an application, Keycloak allows to delegate the user authentication process. This substantially reduces the related implementation overhead for the application. It provides a user with single sign on to access all applications registered at a particular Keycloak instance. In addition to serving as an identity provider, Keycloak is also capable of incorporating existing providers, e.g., social networks. It supports authentication processes that rely on the protocols OpenID Connect and version 2.0 of the Security Assertion Markup Language (SAML). Authorization is conducted using version 2.0 of the Open Authorization (OAuth) protocol. Keycloak furthermore allows to link user information from other identity managment systems using Keberos, such as Lightweight Directory Access Protocol (LDAP) or Active Directory servers.

Keycloak organizes authentication and authorization within so-called *realms*. A realm comprises a set of registered users as well as applications. The authentication within a realm is either realized by supplying a seperate identity service or enabling the access to a set of exisiting identity providers. Users that are part of the realm may be federated with other data by attaching additional identity management systems. The access to resources is steered by defining roles and assigning them to users. An admin console provides means to modify the realm settings. Users can manage their accounts using a separate interface.

There are a set of Keycloak *client adapters*, which enable the communication employing the protocols mentioned above[62]. Depending on the protocol there exist implementations in various programming languages, such as Java and JavaScript [63]. Some of them provide tight integration with specific platforms, for example *Spring Boot* and *WildFly*.

[60] `http://www.keycloak.org/`
[61] `https://github.com/keycloak/keycloak`
[62] `http://www.keycloak.org/docs/latest/securing_apps/index.html`
[63] `http://www.keycloak.org/docs/latest/securing_apps/index.html`

3 Automotive User Stories

This chapter introduces some relevant automotive user stories with regard to cloud connectivity, security, OTA upgrade methods, functional tooling, and smart move components. The various user stories describe the usage of the APPSTACLE ecosystem from different points of view and give further details on the technical requirements for the APPSTACLE platform and in particular for the architecture of the IoT cloud platform.

Due to the amount and heterogeneity of the following user stories, it is later on necessary to filter the user stories to obtain a subset of user stories that (i) are most relevant to WP3; (ii) go into different aspects of the cloud architecture; and (iii) are appropriate for evaluation and the implementation in distinct demonstrators.

Stakeholders

The envisioned APPSTACLE ecosystem consists of different stakeholders with different requirements. This section lists the basic stakeholders considered for the user stories.

Car driver/owner: This stakeholder represent the end-customer who eventually paid the cost of the system. He or she wants to select apps that run on the in-vehicle platform. The vehicle owner also has a strong interest in controlling access to data from his vehicle.

Application developer: The application developer creates apps and solutions for the in-car platform and/or the cloud back-end. The apps may be offered to the end customer via an open platform (app store) or can be part of a specific service functionality from the Original Equipment Manufacturer (OEM), suppliers, or third parties. Consequently, the user of an app might be the end-customer, an OEM, or service provider. The app developer requires an open, powerful development environment that gives easy access to vehicle data, the car, and related cloud services. An app developer further expects good integration with existing ecosystems.

Car manufacturer/In-vehicle platform vendor: This entity is responsible for developing or customizing the in-vehicle platform and install it into a vehicle. This can either be an OEM offering the platform in his cars or an after-market product. It is expected that this stakeholder also operates or manages operation of some additional infrastructure like an app store or a billing system. The main goal of this stakeholder is generating revenue by operating and controlling the ecosystem.

Service provider: This stakeholder operates connected services that may include in-vehicle platform apps or cloud-based apps and offers them to the car driver/owner. The service provider may be identical to the car manufacturer/in-vehicle platform vendor or just use the infrastructure provided by them.

3.1 User Story 01: Roadside Assistance

The technology developed in APPSTACLE allows to support drivers in case the vehicle breaks down. The basic idea is to have a roadside assistance app installed on the vehicle that allows a roadside assistance provider to diagnose and potentially repair the vehicle remotely.

Development phase: The app developer engineers a *Roadside Assistance* application most likely in cooperation with the car manufacturer/in-vehicle platform vendor and the service provider. Additionally, a cloud instance for the roadside assistance provider is set up and appropriate cloud services for controlling the app remotely and analyzing the retrieved vehicle data are developed by the application developer.

Setup phase: The app might already be installed by the OEM. Alternatively the car driver/owner downloads and installs a *Roadside Assistance* application to its in-vehicle platform by using the corresponding app store. This may or may not be part of a contract between the car driver/owner and a service provider (e.g. roadside assistance provider such as ADAC, RAC, AAA, or as part of an insurance policy).

Usage phase: The driver of a vehicle detects a problem with his or her car (e.g. indicated by warning lights) and pulls over or otherwise the car brakes down. The driver contacts his roadside assistance service provider (e.g. by a smart phone app or by phone call, or through a head unit in the car) to request support. During this interaction the service provider is able to remote control the installed *Roadside Assistance* app in order to:

- ...retrieve data from the vehicle
- ...execute diagnostic workflows and retrieve the results
- ...control actuators within the vehicle

As a result, the service provider may be able to assist the driver by giving accurate information on the problem occurred, provide instructions for next steps (e.g. drive to the closest workshop or wait for help), solve the issue remotely (e.g. by re-setting fault memories), or dispatching a roadside assistance associate (e.g. by sending the right expert with the needed spare parts).

Technical requirements:

- Runtime environment for Apps on the in-vehicle platform
- APPSTACLE API which grants access to the data and procedures needed
- APPSTACLE API which can be enriched by OEM specific extensions
- Roles and rights management controlled by the car driver/owner

3.2 User Story 02: Vehicle Tracking

The owner of a vehicle or a third party needs to track the position of a specific vehicle. Such scenario may occur for several reasons, e.g. for fleet management, stolen vehicle tracking, pay-per-drive insurance tariffs, car sharing, or social networks: Let friends track your vehicle.

Development phase: The application developer implements tracking services for the cloud instance of the service provider and further provides a *Vehicle Tracking* app for the in-vehicle platform to forward time and position information.

Setup phase: The car driver/owner downloads the *Vehicle Tracking* app from the app store to install it on the in-vehicle gateway and allows the service provider to activate the forwarding of the relevant information to its cloud server.

Usage phase: The *Vehicle Tracking* app forwards position information alongside with a time stamp periodically to the cloud server of the service provider. Within the cloud, this information is used for enhanced services.

Technical requirements:

- Runtime environment for Apps on the in-vehicle platform
- APPSTACLE API granting access to location
- Roles and rights management controlled by the car driver/owner to protect security and privacy of such sensitive information

3.3 User Story 03: Wrong Way Driver Warning

A vehicle takes part in a wrong way driver warning system in order to increase its own and other vehicle's safety. This is done by enabling numerous cars to forward position and direction data to a central server instance which matches them to a map and detects wrong way drivers. In this case, all vehicles that might be at risk are warned by the service.

Development phase: The app developer implements the wrong way driver warning service for the cloud. Additionally, an in-vehicle application is created to forward time, position, and speed vector information.

Setup phase: The car driver/owner downloads the specific app from the app store to install it on his in-vehicle platform and allows access to position information, while the app registers with the server instance.

Usage phase: The app forwards position and speed vector information alongside with a timestamp periodically to the cloud server of the service provider. Within the cloud, this information is used for map matching and risk analysis. In the case a wrong way driver is detected, the originator of the warning and the vehicles at risk are warned about the situation through the APPSTACLE Human-Machine Interface (HMI) connector.

Technical requirements:

- Runtime environment for Apps on the in-vehicle platform
- APPSTACLE API granting access to location and speed vector

- Roles and rights management controlled by the car driver/owner to protect security and privacy of such sensitive information
- Interface to HMI (in-car or BYOD) to display warnings

3.4 User Story 04: Augment vehicle functionality

This user story describes a scenario where a vehicle will be enhanced by a specific functionality in order to adjust to special equipment. For example, adding a roof rack or a trailer to a vehicle might come with some software modifying brake booster, suspension, and ESP settings. For commercial vehicles such as semi-trailers, different trailer variants might want to extend and adapt functionalities of the tractor unit: When transporting livestock, the capability of having a live camera feed from the trailer might be added. When using a cooling trailer, temperature and state of refrigeration system can be communicated to the driver and additionally stored and analyzed in the cloud back-end.

Development phase: The app developer implements some in-vehicle application that provides additional functionality or makes functionalities from added components available. Additionally, cloud service for analyzing distinct parameters, e.g. temperature values, may be implemented.

Setup phase: The car driver/owner downloads the specific app from the app store to install it on the in-vehicle platform.

Usage phase: The app is provided in the cloud ecosystem and the owner can download or use the functionality based on the business model (rent or buy). The specific software will be downloaded and installed in the in-vehicle platform.

Technical requirements:

- Runtime environment for apps on the in-vehicle platform
- APPSTACLE API granting access to data and functions from in-car Electronic Control Units (ECUs) and externally connected equipment

3.5 User Story 05: Data Collection Fleet Learning

A lot of data is available at a fleet of vehicles that can be used to enhance performance of autonomous driving systems, improve understanding of component aging, and enabling predictive maintenance algorithms. As a single car cannot provide all generated data, the cloud back-end needs to orchestrate which individual cars deliver what information.

Development phase: The application developer implements a configurable data acquisition app for the car that delivers data to the cloud according to the current connectivity situation and requirements from the back-end. Furthermore, cloud-based services needs to be developed to collect and access the data. This enables applications such as predictive maintenance or machine learning.

Setup phase: Most likely such an application is pre-installed by an OEM. However, installing an app from a third party via the app store is possible if the car driver/owner grants the required access.

Usage phase: The app collects data and sends it to the cloud. It can be used to improve the cars software, keeping unplanned maintenance down, and improving future revisions of the same car. In the predictive maintenance use case, the driver can be notified in case of expected problems.

Technical requirements:

- Runtime environment for Apps on the in-vehicle platform
- APPSTACLE API granting access
- Security and privacy of the data

3.6 User Story 06: IoT Data concentration

This user story describes a scenario where the technology developed in APPSTACLE could help to reduce bandwidth usage on the mobile internet connection. Many experts expect that cloud-related data generated by future vehicles exceeds the bandwidth being available by far (or would cause significant costs respectively). The APPSTACLE in-vehicle platform could support in such cases by hosting domain specific apps that reduce the amount of data forwarded to the cloud (e.g. by preprocessing, detection of irrelevant data, data compression).

Development phase: The App developer engineers a specific data concentration application most likely in cooperation with the service provider. Furthermore, specific service within the cloud for unpacking the data have to be established.

Setup phase: The car driver/owner downloads and installs a data concentration application for a specific domain/use case to the in-vehicle platform by using the corresponding app store. This may or may not be part of a contract between the car driver/owner and a service provider.

Usage phase: During the usage of the vehicle, the data concentration app constantly reads the relevant data using the in-vehicle connectivity of the APPSTACLE platform. This data is then concentrated in any of the forms explained earlier and the result of this procedure is forwarded to a connected cloud instance that unpack the transmitted data for further processing.

Technical requirements:

- Runtime environment for Apps on the in-vehicle platform
- APPSTACLE API including the data and procedures needed
- Roles and rights management controlled by the car driver/owner

3.7 User Story 07: Driver Seat Configuration

Vehicles, which are used by several drivers, can store the seat configuration of each driver. Car fleets (e.g. bus or truck companies) may use a cloud service to store the configuration for each driver, independent of the current car (truck, bus).

Development phase: The application developer implements the seat configuration by using the access to the driver seat ECU. He or she also implements the corresponding cloud service to exchange driver configuration data with the cloud.

Setup phase: The car driver/owner downloads the specific app from the app store to install it on the in-vehicle platform and allows access to the driver seat. The app registers with the server instance.

Usage phase: The car driver identifies himself by an Near Field Communication (NFC) tag or fingerprint reader. Afterward, the app downloads the appropriate driver configuration from the cloud.

Technical requirements:

- Runtime environment for apps on the in-vehicle platform
- APPSTACLE API granting access to driver seat and personal identification system

3.8 User Story 08: Parking Space Finder

The driver of a car needs to find a nearby parking space - as quick as possible and as close as possible. Traffic or street surveillance systems may help to solve this frequent problem and thereby also minimizing the parking search traffic.

Development phase: The app developer implements the *Parking Space Finder* app, while the appropriate cloud service, probably realized by a second party, provides the *Intelligent Street* system data and cloud services to find free parking spaces.

Setup phase: The car driver/owner downloads the specific app from the app store to install it on the in-vehicle platform and allows access to the HMI and navigation system. The app registers with the server instance.

Usage phase: The car driver enters a "find parking space" command to the app using the HMI. The *Parking Space Finder* app provides location information to the cloud instance and receives data of available parking spaces. The app will display this information on the HMI.

Technical requirements:

- Runtime environment for apps on the in-vehicle platform
- APPSTACLE API providing access to car location (GPS data) and HMI

3.9 User Story 09: Improved Carpooling System

Current carpooling systems require medium term registration (at least one day ahead usually) and much administrative effort. Automated data service may minimize the time necessary to find a driver or passenger. As security risks are involved, it may be necessary to provide a secure authentication for both driver and passenger.

Development phase: The app developer implements the *Carpooling* app. The carpooling provider implements the cloud service, which finds out appropriate passenger/driver pairings.

Setup phase: The car driver/owner downloads the specific app from the app store to install it on the in-vehicle platform and allows access to the HMI. The app registers with the server instance. The driver proves his or her identity by password, fingerprint reader, or iris scan.

Usage phase: The car driver enters a "find passenger" command to the app using the HMI. Afterward, the *Carpooling* app provides location information to the cloud instance to receive data of possible passengers. The app will then display this information on the HMI. This app may well be augmented by a chat service.

Technical requirements:

- Runtime environment for apps on the in-vehicle platform
- APPSTACLE API providing access to car location (GPS) and HMI as well as to the authentication system (fingerprint reader, iris scanner) of the car

3.10 User Story 10: Car Accident Registration by Video

"Dash Cam (Crash Cam)" videos may upload the most current seconds preceding an accident to the appropriate cloud service.

Development phase: The application developer implements the *Accident Registration* app as well as the cloud service receiving the video data.

Setup phase: The car driver/owner downloads the specific app from the app store to install it on the in-vehicle platform and allows access to the car's camera(s). The app registers with the server instance.

Usage phase: This app does not usually need an HMI, but runs in the background, invisible to the driver. However, it may need configuration and test functions requiring access to the HMI.

Technical requirements:

- Runtime environment for apps on the in-vehicle platform
- APPSTACLE API providing access to car camera

3.11 User Story 11: Car Theft Registration & Car Vandalism Registration

"Dash Cam" videos may upload the current seconds preceding and during a car theft to the cloud. Microphone and other sensor data may provide further valuable information to identify thieves and vandals.

Development phase: The app developer implements the *Theft Registration* app as well as the cloud service receiving the video and sensor data.

Setup phase: The car driver/owner downloads the specific app from the app store to install it on the in-vehicle platform and allows access to the car's camera(s), microphones, and shock/vibration sensors. The app registers with the server instance.

Usage phase: This app does not usually need an HMI, but runs in the background, invisible to the driver. However, it may need configuration and test functions requiring access to the HMI.

Technical requirements:

- Runtime environment for apps on the in-vehicle platform
- APPSTACLE API providing access to car camera and microphone

3.12 User Story 12: Traffic Jam Warning & Traffic Jam Avoidance

Traffic surveillance systems provide information on current traffic jams and calculate routes to minimize the travel time by considering the current traffic situation.

Development phase: The application developer implements the *Traffic Jam Warning* app. The traffic jam warning system relies on a global system and will be realized as a cloud service provided by a second party.

Setup phase: The car driver/owner downloads the specific app from the app store to install it on the in-vehicle platform and allows access to the car's GPS and navigation system. The app registers with the cloud service.

Usage phase: The app downloads current traffic information from the appropriate cloud service and displays warnings on the HMI if necessary. It may also connect to the navigation system to calculate alternative routes in order to avoid traffic jams.

Technical requirements:

- Runtime environment for apps on the in-vehicle platform
- APPSTACLE API providing access to the HMI and navigation system

3.13 User Story 13: Chat Service for Car Drivers

The idea behind this user story is to provide a chat service with nearby car drivers in order to get acquainted, share traffic information (e.g. traffic jams, construction sites, speed cameras), or admonish car drivers.

Development phase: The application developer implements the *Car Chat* app, which must be able to spot nearby cars. A cloud service will be implemented for interconnection of car drivers.

Setup phase: The car driver/owner downloads the specific app from the app store to install it on the in-vehicle platform and allows access to the car's GPS system, HMI, microphone and speaker. The app registers with the cloud service.

Usage phase: The app provides information about nearby drivers who are prepared to communicate. A visual inquiry may be sent to the HMI of a nearby car, thus providing the chance to start communication.

Technical requirements:

- Runtime environment for apps on the in-vehicle platform
- APPSTACLE API providing access to HMI, microphone, and speaker

3.14 User Story 14: Traffic Enforcement Camera Warning

A cloud service providing information on the position of speed cameras or red light cameras may increase traffic security by reminding car drivers to follow traffic rules.

Development phase: The application developer implements the *Speed Camera Warning* app. An appropriate cloud service will be provided by a second party.

Setup phase: The car driver/owner downloads the specific app from the app store to install it on the in-vehicle platform and allows access to the car's GPS system and HMI. The app registers with the cloud service.

Usage phase: The app uses the cloud service to get information about nearby speed cameras and the like. If such traffic control is detected, a warning message will be displayed on the HMI. An useful extension may provide a method to share information about recently spotted cameras via a cloud service.

Technical requirements:

- Runtime environment for apps on the in-vehicle platform
- APPSTACLE API providing access to HMI and speaker

3.15 User Story 15: Advertising Services for Drivers

Providing information to travelers about points of interest such as local shopping facilities, restaurants, public events etc.

Development phase: The app developer implements the *Advertising Service* app. An appropriate cloud service will be provided by a second party.

Setup phase: The car driver/owner downloads the specific app from the app store to install it on the in-vehicle platform and allows access to the car's GPS system and HMI. The app registers with the cloud service.

Usage phase: The app uses the cloud service to get information about nearby advertising partners. Additionally, the app may also filter these information according to the wishes of the driver.

Technical requirements:

- Runtime environment for apps on the in-vehicle platform
- APPSTACLE API providing access to GPS, HMI, and speaker

3.16 User Story 16: Social Media

Everybody wants to stay connected all the time. The integration of social media within the vehicle offers new ways to share your life with others and let others follow you on your way of life. The idea is going to a wide spread of interpreting due to the possibilities Twitter, Facebook, Blogs, YouTube (live streaming) and other kinds of social media offer.

Development phase: The application developer implements the *Social Media app*.

Setup phase: The car driver/owner downloads the specific app from the app store to install it on the in-vehicle platform and allows access to the car's GPS system, Cameras and HMI. The app registers with the cloud service.

Usage phase: The app uses the cloud service to get information about nearby social media partners or "friends". For example, the driver will be informed about Facebook friends on the same road or nearby, while another idea would be to share beautiful landscapes with one click while driving. The app may also filter these information according to the wishes of the driver.

Technical requirements:

- Runtime environment for apps on the in-vehicle platform
- APPSTACLE API providing access to GPS, Camera, mic, HMI and speaker

3.17 User Story 17: Ambulance Assist

Normally, a path is made for the ambulance when vehicle drivers hear the ambulance's siren and move away to the next lane or so. This is easier when the path of the ambulance has fewer vehicles traveling in its direction or when the ambulance is not at a traffic signal. But if there is a traffic jam or there is a red signal at a busy junction, making way for the ambulance is difficult and time consuming. In order to avoid this, it would be better if the ambulance could communicate with the traffic signals well in advance so as to have a clear path.

Development phase: The application developer either develops a cloud specific app that communicates with the cloud and in turn communicates with the traffic system or develops an in-vehicle app that communicates directly with traffic signals in the vicinity.

Setup phase: This will be a pre-installed app which could be a part of the contract between the platform provider (OEM/supplier/3rd party), the organization(s) that provide the ambulance service, and the traffic authority of the region/state/country.

Usage phase: Once the app is switched on, it communicates the GPS information of the vehicle to the cloud along with the source and destination locations. A specific cloud services track the ambulance and communicates with the next immediate 1-2 traffic signals to make a clear path for the ambulance. Additionally, the cloud instance could also receive the traffic information in this path to communicate, perhaps, with traffic signals that are much ahead in the ambulance's path to ensure a continuous free path for the ambulance.

Additional use case 1: The cloud instance also communicates with the vehicles in the region of the ambulance informing them to make way in advance.

Additional use case 2: The app also provides a means to communicate the patient's conditions to the hospital authorities so that the hospital has the necessary environment ready to treat the patient with no delays.

Technical requirements:

- Runtime environment for apps on the in-vehicle platform
- The platform has necessary APIs and protocols to communicate with a system outside its ecosystem
- APPSTACLE API providing access to GPS etc.

3.18 User Story 18: System Surveillance and Maintenance

Diagnosis services (in form of either internal or external components) need to get access to sensor, actuator, status, state, or further parameters of diverse ECUs and components. New technology allows collecting such properties remotely. The need of this may occur when the system has issues in performing its tasks. Diagnosis services could even have the possibility to perform sophisticated analysis of problems via collected information over time and live data. This information includes QoS relevant data as the correctness or timeliness of data and transmissions. The range of tracked information may be extended.

Development phase: An application developer implements an independent app for collecting and sending this information to the cloud, while the cloud has to provide services for aggregated views of the information.

Setup phase: The related monitoring app can be installed by an authorized user (may also be the OEM).

Usage phase: While the vehicle is in usage, the monitoring app collects and sends information about the systems functionality to the cloud. The maintainer of the car reacts to performance issues. Based on the collected monitoring data, reports generated by the cloud's services gives an outline on the circumstances and possible areas of conflict.

Technical requirements:

- In-vehicle app runtime environment
- Roles and Rights management
- Definition of interfaces for monitoring
- QoS mechanisms and QoS-monitoring

3.19 User Story 19: Pool car management

An organization has a pool of cars to be used by employees. These employees can make reservations for pool cars and then unlock and start the car via an app on his or her mobile phone.

Development phase: The app developer develops an app for the car, a mobile phone app for the user, and appropriate cloud services that allows those apps to work together. The vehicle app needs to be able to unlock the car and start the engine.

Setup phase: Initially, the in-vehicle app is installed on the vehicle and the car is added to a pool. The user installs the mobile phone app so that the pool owner can grant access to vehicles in the pool.

Usage phase: The user makes a reservation for a vehicle, walks to the vehicle, and requests the car to be unlocked. Afterward, the authorization for the user is verified and the vehicle is unlocked.

Technical requirements:

- Runtime environment for apps on the in-vehicle platform
- APPSTACLE API allowing to unlock vehicle and start engine
- Roles and rights management controlled by the car driver/owner to protect security

3.20 User Story 20: In-vehicle behavior learning

Due to the complex architecture of the in-vehicle system, car manufacturers may find it quite challenging to configure the communication between ECUs and calibrate each ECU individually in an optimal manner. Additionally, technicians as well as car manufacturers often require network analytics to perform in-vehicle diagnostics.

Development phase: A web interface will be developed to provide a network/asset inventory map to the technician/car manufacturer which may give real-time information about misconfiguration or potential threats to the vehicle.

Setup phase: A network monitoring device is connected on the gateway or the OBDII port of the vehicle and the car driver/owner can access the web application from any device that is present inside the car, such as a smart phone or tablet.

Usage phase: The car driver authenticates in the web application to observe the real-time communication between ECUs in the form of an in-vehicle network map as well as the detailed messages that are sent. The web application notifies the user when it detects:

- Misconfiguration in the in-vehicle architecture
- Potential threats from abnormal behavior in the in-vehicle network

Technical requirements:

- Network monitoring device, e. g. Intrusion Detection System (IDS), connected to the vehicle gateway or diagnostic port

3.21 User Story 21: Secure Car2X data exchange

Data from the vehicle's physical environment are often received or transmitted from/to nearby stations (Car2Infrastructure) or vehicles (Car2Car) to enhance awareness about road conditions (e.g. traffic jams). As the exchanged data may contain sensitive information, security aspects should be considered to avoid that they are spoofed or malformed.

Development phase: A web interface application will be developed to provide access to the messages exchanged in the Car2X network as well as information about potential threats that will impose message leakage.

Setup phase: A network monitoring (IDS) device is connected on the gateway of a target vehicle and the car driver/owner can access the web application from any device smart phone or tablet device that is present inside the car, but also from remotely.

Usage phase: The car driver identifies himself/herself in the app to observe the communication between the target vehicle and the nearby stations or vehicles in real-time. A real-time network map can also be shown with the current connected devices in the wireless network. The app notifies the user when it detects:

- Addition or withdrawal of devices in the network
- Potential threats during Car2X data exchange

Additionally, the app can perform detailed logging and reproduction of the secured data received by the target vehicle, derive only the required data for analysis, and transmit them further to the car manufacturer or the nearest station for inferring information about road conditions.

Technical requirements:

- Network monitoring device (IDS) connected remotely or directly to the vehicle gateway

3.22 User Story 22: Emergency Braking & Evading Assistance System (EBEAS)

The basic idea of the EBEAS app is to automate reactions in the case of detected obstacles in front of a vehicle. For this purpose, EBEAS coordinates different in-car subsystems.

Development phase: The application developer engineers an EBEAS application, most likely in cooperation with the car manufacturer/in-vehicle platform vendor, service provider, and suppliers that developed the participating automotive subsystems. As an EBEAS encompasses highly safety-critical functionality, app developers have to be trusted, e.g. by utilizing certification etc.

Setup phase: The app might already be installed by the OEM. Alternatively the car owner downloads the EBEAS app from the app store to install it on the in-vehicle platform.

Usage phase: The basic version of EBEAS has three functions. The radar sensors of the vehicle are used to detect obstacles in front of the vehicle, while the EBEAS logic component decides whether to evade the obstacle or to brake and performs according actions, e.g. by using the adaptive cruise control functionality. Furthermore, the detected obstacle is reported to a service deployed in the automotive IoT cloud to inform other vehicles about the detected obstacle.

An extended versions of EBEAS could include

- ...a pre-crash system to prevent serious damage from the driver and other passengers in cases an accident is not avoidable
- ...a car-to-car communication system to negotiate on the reaction to a dangerous situation with other vehicles in range

Technical requirements:

- Runtime environment for apps on the in-vehicle platform
- Real-time scheduling
- In-vehicle connectivity and coordination of participating ECUs

4 Architecture Specification

This chapter specifies the initial draft of the cloud platform architecture that shall be established within the APPSTACLE project including the message flow as well as the generic building block such as data management, device management, and possible big data concepts. In particular, the mutual interactions between the various building blocks are described. Note that there are important interdependencies regarding components that are part of the in-vehicle platform. Therefore, the particular vehicle-to-cloud interactions are addressed as well.

In general, the cloud platform is distinguished into three layers (cf. Fig. 4.1):

1. the *Core Layer*,
2. the *Data Analytics & Visualization Layer*,
3. and the *Application Layer*.

Core Layer The *Core Layer* captures the foundations of the cloud platform. In this regard, the *Message Gateway* represents a core functionality to enable the communication between a large number of vehicles and the cloud back-end. Therefore, the corresponding component has to fulfill a set of requirements. For example, the infrastructure has to scale with the number of vehicles that send and receive messages. Furthermore, it is assumed that the vehicles communicate by using different protocols, e.g. MQTT. The *Message Gateway* component converts each message into a common format to provide a consistent interface to the cloud back-end applications. It also maintains a separate device registry to manage the individual vehicles. To enable authentication and authorization, the delivery of messages to entities within the cloud platform as well as the set of vehicles is controlled by a separate component. Messages may be forwarded to different components within the *Core Layer*.

Firstly, messages may be stored within a *Data Management* component that persists information, e.g. telemetry data provided by the vehicles. This allows to create a historical view on each vehicle connected to the cloud platform. The information is provided by *Apps* that run within the in-vehicle *App Runtime*.

Secondly, messages may be forwarded to a *Device Representation* component that maps the state of physical entities (i.e. vehicles) to a digital representation allowing to programmatically interact with the physical entity as it is a digital one. This provides a consistent programming model to App developers.

Thirdly, messages can be further transferred to a *Core Services* component that realizes some generic functionality and is further responsible to send and receive information from domain-specific services.

Furthermore, the *Message Gateway* is connected to a *Device Management* component, which is responsible for the provisioning and modification of in-vehicle functionality. Via the *Message Gateway*, this component interacts with the *In-Vehicle Gateway* that is part of the host system provided by the in-vehicle platform. In addition to transferring in-vehicle *Apps*, the *Device*

Management component also allows to deliver updates of the host system. The communication with the in-vehicle platform may rely on different protocols, e.g. LWM2M or OMA-DM.

Via the component for *Identity Management*, access to distinct functionality can be granted for eligible devices and users.

Data Analytics & Visualization Layer The *Data Analytics & Visualization Layer* provides services that are built on top of the collected vehicle data stored within the *Data Management* component of the *Core Layer*. This includes the analysis of the presumably large amounts of persisted data by using *Big Data* technologies. Such analyses may refer to individual as well as groups of vehicles. The time series data may also be accessible via a *Visualization* component, e.g. for depicting the development of telemetry variables. Depending on the use case, a *Report Generator* allows to generate appropriate business reports.

Application Layer The *Application Layer* exposes services that are specific to certain use cases. Thus, they access the functionality provided by the underlying layers. This includes consuming the messages that are routed via the *Device Representation* component or accessing data generated using *Big Data Analysis*, but also the component for *Identity Management* and the functionality provided by the *Core Services*. A majority of those services is assumed to provide functionality to *Apps* that are running on the in-vehicle platform. In addition, the *Application Layer* comprises the *Web User Interface* that allows to obtain *Apps* and transfer the corresponding packages to the in-vehicle platform.

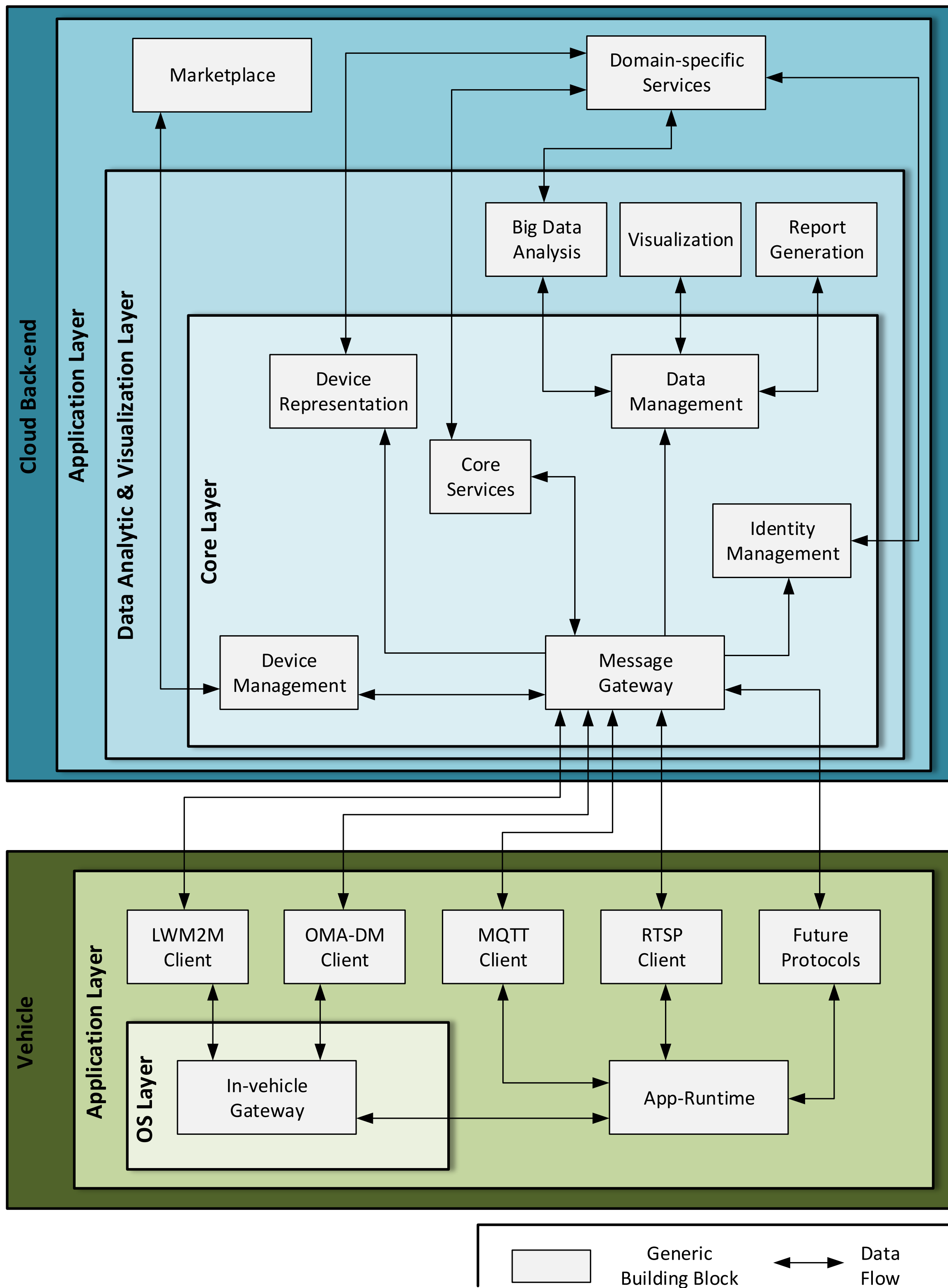

Figure 4.1: The cloud platform architecture with generic building blocks

5 Architecture Evaluation

Based on the state of the art analysis from Chapter 2 and the insights from Chapter 3, this chapter evaluates technology for each generic building block of the architecture specification in Chapter 4. In particular, the evaluation is based on the evaluation criteria from the following section as well as additional literature like surveys or secondary studies, e. g. literature review or systematic mapping study.

5.1 Evaluation criteria

In the following, the evaluation criteria (**C**) used to assess specific technologies regarding their suitability in the context of the cloud platform architecture are presented. The criteria are determined based on work presented in deliverable D1.1. There, a thorough investigation regarding the requirements of the technologies employed is conducted within the state-of-the-art chapter. A subset of criteria has been selected for this deliverable, based on their applicability to the cloud platform architecture. Note that specific criteria are added for individual sub topics, if required. Each of the following base criteria is attached with an explanation that defines its scope:

- **C1** *Adaptability*: Is it possible to adapt the technology to specific needs like customer-specific requirements?
- **C2** *Community*: Is there a living community? (Activity in Gits, mailing lists, discussion boards etc.).
- **C3** *Compatibility*: Does the technology exhibit any standardized interfaces or does it rely on common exchange format or specifications to foster a compatibility with other technology from related aspects?
- **C4** *Documentation*: Is there a well-maintained documentation? Is help available in terms of tutorials, videos etc.?
- **C5** *Scalability*: Does the technology exhibit any mechanisms to scale with the number of devices or the amount of data transferred?
- **C6** *Software License*: What is the license? What other components are integrated (e.g. Linux kernel). Copyleft yes or no, strong or weak? In case of non-FOSS software: Costs.
- **C7** *Up-to-dateness*: Is the technology well maintained and does it incorporate novel features?

5.2 Technology Evaluation

For each component of the cloud architecture, the set of available technologies are compared regarding their suitability according to the defined criteria. The structure of this section is aligned to Chap. 2.

5.2.1 IoT Cloud Platform

This section evaluates Eclipse Kapua that are used as an IoT Cloud platform (cf. Sec. 2.2.1)

Table 5.1: Eclipse Kapua (cf. Sec. 2.2.1).

Criterion	Value(s)
C1	Generic IoT Cloud platform that can be changed for different circumstances.
C2	Project website: `https://www.eclipse.org/kapua/` GitHub: `https://github.com/eclipse/kapua`, 3020 commits (till 1.2.2018), 18 contributors Forum: `https://www.eclipse.org/forums/index.php/f/340/`, 28 discussed topics Reporting Issues: `https://github.com/eclipse/kapua/issues`, 250 open and 552 closed issues Mailing list (developer): `https://accounts.eclipse.org/mailing-list/kapua-dev`
C3	Kapua connects to IoT devices via MQTT and other protocols. Its services integrate with IT applications through flexible message routing and REST API.
C4	General Documentation: `https://www.eclipse.org/kapua/documentation.php` Getting Started: `https://www.eclipse.org/kapua/getting-started.php` Developer Guide: `https://www.eclipse.org/kapua/docs/0.3.0/developer-guide/en/index.html`
C5	strong relationship with scalable projects such as Hono; however, they have not been integrated yet.
C6	Eclipse Public License (EPL)
C7	Latest Releases: 0.3.2 (22.12.2017), few releases (6)

5.2.2 Application and Service Integration

This section covers the following technologies that are used regarding application and service integration (cf. Sec. 2.3):

- Eclipse Ditto (cf. Tab. 5.2),
- Mihini (cf. Tab. 5.3),
- Eureka (cf. Tab. 5.4),
- Apache Camel (cf. Tab. 5.5), and
- Spring Integration (cf. Tab. 5.6).

Table 5.2: Eclipse Ditto (cf. Sec. 2.3.1).

Criterion	Value(s)
C1	The services use an externally provided MongoDB as database which is a must for users. It uses Eclipse Hono for the message exchange with devices.
C2	Project website: `https://www.eclipse.org/ditto/` GitHub: `https://github.com/eclipse/ditto`, 612 commits (till 1.2.2018) Forum: `https://www.eclipse.org/forums/index.php/f/364/`, no topics! blog: `https://www.eclipse.org/ditto/blog.html`, 4 posts Mailing list (developer): `https://accounts.eclipse.org/mailing-list/ditto-dev`
C3	consumes AMQP 1.0 interface and provides REST-like HTTP API for CRUD and search operations on Things.
C4	General Documentation: `https://www.eclipse.org/ditto/intro-overview.html` Sandbox: `https://ditto.eclipse.org/` Architecture: `https://www.eclipse.org/ditto/architecture-overview.html` Installation: `https://www.eclipse.org/ditto/installation-building.html`
C5	On increasing load still provide a scalable, robust and high-performance implementation; however, the technology is not still matured.
C6	Eclipse Public License (EPL)
C7	Latest Releases: 0.1.0-M3 (12.01.2018), 0.1.0-M1 (18.12.2017), small numbers of releases (2)

Table 5.3: Eclipse Mihini (cf. Sec. 2.3.2).

Criterion	Value(s)
C1	developing portable M2M applications easily
C2	Project website: Archived `https://www.eclipse.org/projects/archives.php` GitHub: `https://github.com/nim65s/mihini-repo` Forum: no active forum Mailing list (developer): No subscriber
C3	LUA and REST API
C4	General Documentation: `https://wiki.eclipse.org/Mihini` Development documentations: `https://wiki.eclipse.org/Mihini/Development` Installation: `https://wiki.eclipse.org/Mihini/mihinifeatures`
C5	—
C6	Eclipse Public License (EPL), one LGPL library
C7	Latest Releases: 0.9 (June 2013)

Table 5.4: Eureka (cf. Sec. 2.3.4).

Criterion	Value(s)
C1	Eureka fills the need for mid-tier load balancing. Used for specific purposes such as aiding Netflix Asgard and cassandra deployments.
C2	Introduction: https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance GitHub: https://github.com/Netflix/eureka, 1495 commits (till 1.2.2018), 57 contributors wiki: https://github.com/Netflix/eureka/wiki Mailing list (developer): no subscriber
C3	REST API
C4	General Documentation: https://github.com/Netflix/eureka/wiki Configuring: https://github.com/Netflix/eureka/wiki/Configuring-Eureka Building Client and Server: https://github.com/Netflix/eureka/wiki/Building-Eureka-Client-and-Server Running the Demo APP: https://github.com/Netflix/eureka/wiki/Running-the-Demo-Application
C5	It addresses services to be stateless (non-sticky).
C6	Apache License 2.0
C7	Latest Releases: 1.8.6 (15.11.2017), large number of releases (223)

Table 5.5: Apache Camel (cf. Sec. 2.3.5).

Criterion	Value(s)
C1	supports most of the Enterprise Integration Patterns
C2	Project website: http://camel.apache.org/ GitHub: https://github.com/apache/camel, 31187 commits (till 1.2.2018), 374 contributors Issue tracker: https://issues.apache.org/jira/projects/CAMEL/issues, 12184 reported issues, 11716 closed Contributing: https://github.com/apache/camel/blob/master/CONTRIBUTING.md mailing lists: http://camel.apache.org/mailing-lists.html
C3	MQTT and some other interfaces
C4	General Documentation: http://camel.apache.org/documentation.html Developer Documentation: http://camel.apache.org/developers.html Building Documentation: http://camel.apache.org/building.html
C5	offers scalability, transaction support, concurrency and monitoring
C6	Apache License 2.0
C7	Latest Releases: 2.20.2 (26.01.2018), 2.19.4 (05.11.2017), good number of releases (120)

Table 5.6: Spring Integration (cf. Sec. 2.3.6).

Criterion	Value(s)
C1	No hard coded process flows and capability to easily change and expend these flows according to application requirement.
C2	Project website: `http://projects.spring.io/spring-integration/` GitHub: `https://github.com/spring-projects/spring-integration`, 9046 commits (till 1.2.2018), 101 contributors Issue tracker: `https://jira.spring.io/browse/INT/?selectedTab=com.atlassian.jira.jira-projects-plugin:summary-panel`, 4314 reported issues, 3654 closed Contributing: `https://github.com/spring-projects/spring-integration/blob/master/CONTRIBUTING.adoc` Communication: `https://gitter.im/spring-projects/spring-integration`
C3	MQTT, AMQP
C4	General Documentation: `https://docs.spring.io/spring-integration/docs/4.3.12.RELEASE/reference/html/overview.html` Contributor Documentation: `https://github.com/spring-projects/spring-integration/blob/master/CONTRIBUTING.adoc` Sample Projects: `https://github.com/spring-projects/spring-integration-samples`
C5	possible but via technology configuration
C6	Apache License 2.0
C7	Latest Releases: 5.0.1 (29.01.2018), good number of releases (150)

5.2.3 Data Analytic and Visualization

This section covers the following technologies that are used regarding data analytics and visualization (cf. Sec. 2.4):

- Apache Storm (cf. Tab. 5.7),
- Apache Flink (cf. Tab. 5.8),
- Eclipse BIRT (cf. Tab. 5.9), and
- Grafana (cf. Tab. 5.10).

Table 5.7: Apache Storm (cf. Sec. 2.4.1).

Criterion	**Value(s)**
C1	- Supports only stream processing - Fine-grained API for composing computations
C2	Project website: `https://storm.apache.org/` GitHub: `https://github.com/apache/storm` Issue tracker: `https://issues.apache.org/jira/projects/STORM/issues/STORM-2904?filter=allopenissues` Project Management and Contributors: `https://storm.apache.org/contribute/People.html`) Project By-laws: `https://storm.apache.org/contribute/BYLAWS.html`
C3	- Since version 1.0.3, Apache Storm is capable of consuming data from Apache Kafka 0.8.x (`http://storm.apache.org/releases/1.0.3/storm-kafka.html`)
C4	General Documentation: `https://storm.apache.org/releases/<RELEASE>` Developer Documentation: `https://github.com/apache/storm/blob/master/DEVELOPER.md` Implementation Documentation: `http://storm.apache.org/releases/current/Implementation-docs.html` Additional Information: `https://storm.apache.org/talksAndVideos.html`
C5	Apache Storm has been developed as a scalable system to process large amounts of streaming data.
C6	Apache License 2.0
C7	Latest Releases: 1.1.1 (01.08.2017), 1.0.5 (15.09.2017)

Table 5.8: Apache Flink (cf. Sec. 2.4.2).

Criterion	**Value(s)**
C1	- Supports stream as well as batch processing - Provides corresponding DataStream and DataSet APIs - High-level contructs for composing computations
C2	Project website: `https://flink.apache.org/` GitHub: `https://github.com/apache/flink` Mailing lists: `https://flink.apache.org/community.html#mailing-lists` Issue tracker: `https://issues.apache.org/jira/projects/FLINK/issues` Meetups: `https://www.meetup.com/de-DE/topics/apache-flink/` Conference (Flink Forward): `https://flink-forward.org/` Company backing: `https://data-artisans.com/`
C3	- It is possible to execute Apache Storm Topologies on Apache Flink (`https://flink.apache.org/news/2015/12/11/storm-compatibility.html`) - BETA: Apache Flink is compatible to Hadoop Map Reduce interfaces (`https://ci.apache.org/projects/flink/flink-docs-release-1.4/dev/batch/hadoop_compatibility.html`)
C4	General Documentation (comprehensive): `https://ci.apache.org/projects/flink/flink-docs-release-<RELEASE>/` Apache Flink Training: `http://training.data-artisans.com/`
C5	Apache F link is a system for the distributed processing of large amounts of data.
C6	Apache License 2.0
C7	Latest Stable Release: 1.4 (12.12.2017)

Table 5.9: Eclipse BIRT (cf. Sec. 2.4.3).

Criterion	Value(s)
C1	- Reports can be customized - Report Item Extension API allows to define additional report items - Open Data Access (ODA) Extension API allows to create custom data source drivers
C2	Project website: `https://www.eclipse.org/birt/` GitHub: `https://github.com/eclipse/birt` Forum: `http://www.eclipse.org/forums/?group=eclipse.birt` Mailing list (news): `http://dev.eclipse.org/mhonarc/lists/birt-news/maillist.html` Mailing list (developer): `https://accounts.eclipse.org/mailing-list/birt-dev` OpenText Analytics Developer Community: `https://www.opentext.com/what-we-do/products/analytics/analytics-developer-community`
C3	- Integration of the report designer within the Eclipse IDE - Reports are stored as XML files - Reports can access a variety of data sources, including SQL databases, Web Services and XML files
C4	General Documentation: `https://www.eclipse.org/birt/documentation/` Demos: `https://www.eclipse.org/birt/demos/` Open Text Documentation: `https://www.opentext.com/what-we-do/products/analytics/analytics-resources/documentation`
C5	—
C6	Eclipse Public License (EPL)
C7	Latest Stable Release: 4.7.0 (22.06.2017)

Table 5.10: Grafana (cf. Sec. 2.4.4).

Criterion	Value(s)
C1	- Dashboards can be customized - Plugins allow to augment the functionality (`https://grafana.com/plugins`)
C2	Project website: `https://grafana.com/` GitHub: `https://github.com/grafana/grafana` Twitter: `https://twitter.com/grafana` Conference (GrafanaCon): `http://grafanacon.org/`
C3	- Connectivity to a variety of data sources is provided - Additional data sources can be added via plugins
C4	General Documentation: `http://docs.grafana.org/` FAQ: `https://community.grafana.com/c/howto/faq` Tutorials: `docs.grafana.org/tutorials/` YouTube Channel: `https://www.youtube.com/channel/UCYCwgQAMm9sTJv0rgwQLCxw`
C5	- Grafana is designed to visualize data that is stored in scalable data storage deployment
C6	Apache License 2.0
C7	Latest Stable Release: 4.6.3 (14.12.2017)

5.2.4 Data Storage and Management

In the following, the given criteria are applied to the technologies from Section 2.5) for realizing a data storage and management:

- Apache CouchDB (cf. Tab. 5.11),
- MongoDB (cf. Tab. 5.12),
- Neo4j (cf. Tab. 5.13),
- InfluxDB (cf. Tab. 5.14)
- ArangoDB (cf. Tab. 5.15)
- OrientDB (cf. Tab. 5.16)
- and Couchbase (cf. Tab. 5.17)

Table 5.11: Apache CouchDB (cf. Sec. 2.5.1).

Criterion	Value(s)
C1	Security & validation via custom security models
C2	Project website: `http://couchdb.apache.org/` GitHub: 129 contributors, 10.962 commits `https://github.com/apache/couchdb` Issue Tracker: 67 open, 253 closed issues `https://github.com/apache/couchdb/issues` Mailing list (`https://couchdb.apache.org/#mailing-list`), Chat (IRC), Slack (5,128 tagged questions)
C3	HTTP-based REST interface; data stored as JSON objects
C4	Apache CouchDB 2.1 Documentation: `http://docs.couchdb.org/en/2.1.1/` Wiki, blog (`https://blog.couchdb.org/`)
C5	Replications, read requests (load balancing), write requests (clustering), and data (clustering) (`http://guide.couchdb.org/draft/scaling.html`)
C6	Apache License Version 2.0
C7	Latest Stable Release: 2.1.0 (04.08.2017); frequent releases (47)

Table 5.12: MongoDB (cf. Sec. 2.5.2).

Criterion	Value(s)
C1	Querying, data model
C2	Project website: `https://www.mongodb.com` GitHub: 327 contributors, 40.377 commits `https://github.com/mongodb/mongo` Issue Tracker: 70.501 reported issues `https://jira.mongodb.org` Google groups (32.458 threads), frequent user events
C3	HTTP-based REST interfaces (Python, Java, Ruby, Node.jS); data stored as Binary JSON (BSON) objects
C4	MongoDB Manual 3.6: `https://docs.mongodb.com` White papers, datasheets, webinars, presentations, blog (`https://www.mongodb.com/blog`)
C5	Replications, Auto-Sharing for horizontal scale-out databases Cluster, performance, and data scale proven in industrial settings (`https://www.mongodb.com/mongodb-scale`)
C6	Database Server and Tools: GNU AGPL v3.0 Drivers: Apache License v2.0. for mongodb.org; Varying licenses for third party drivers
C7	Latest Stable Release: 3.6.2 (10.01.2018); frequent releases (496)

Table 5.13: Neo4j (cf. Sec. 2.5.3).

Criterion	Value(s)
C1	—
C2	Project website: `https://neo4j.com/` GitHub: 151 contributors, 52.992 commits `https://github.com/neo4j` Issue Tracker: 308 open, 1873 closed issues `https://github.com/neo4j/neo4j/issues` Slack (14.897 tagged questions), user events, meetups, training days, online training, social media
C3	REST API, language-specific drivers (C#, Java, JavaScript, Python), export of query data to JSON and XLS format
C4	Neo4j documentation 3.3: `https://neo4j.com/docs/` API documentation for Java, JavaScript, Python, and .NET Blog (https://neo4j.com/blog/)
C5	Cluster management component with full replication for each instance, In-Memory Sharing
C6	Community edition: GPL v3 license Different licensing models for enterprise edition (commercial, developer, evaluation, and educational license)
C7	Latest Stable Release: 3.3.2 (22.01.2018); frequent releases (220)

Table 5.14: InfluxDB (cf. Sec. 2.5.4).

Criterion	Value(s)
C1	Support for custom dashboards and reporting interfaces
C2	Project website: `https://www.influxdata.com/` GitHub: 288 contributors, 13.176 commits `https://github.com/influxdata/influxdb` Issue Tracker: 636 open, 4741 closed issues `https://github.com/influxdata/influxdb/issues` Active forum (`https://community.influxdata.com/`), Google groups (3.057 threads)
C3	Write and query via HTTP APIs
C4	InfluxDB 1.4 documentation `https://docs.influxdata.com/influxdb/v1.4/` Blog (https://www.influxdata.com/blog/), technical papers, videos, webinars
C5	Scalability through clustering (not open source and only supported in Influx-Enterprise)
C6	MIT License
C7	Latest Stable Release: 1.4.2 (15.11.2017); frequent releases (206)

Table 5.15: ArangoDB (cf. Sec. 2.5.5).

Criterion	Value(s)
C1	Support for custom data structures as well as security and performance
C2	Project website: `https://www.arango.com/` GitHub: 76 contributors, 41.880 commits `https://github.com/arangodb/arangodb` Issue Tracker: 330 open, 2.359 closed issues `https://github.com/arangodb/arangodb/issues` Active forum (`https://www.arangodb.com/community/`)
C3	HTTP-based REST interface (Python, Java, Ruby, Node.js, etc.); VelocyPack: data stored in JSON and Binary format
C4	ArangoDB manual 3.3 `https://docs.arangodb.com/3.3/Manual/index.html` Documentation (https://www.arangodb.com/documentation/) Blog (https://www.arangodb.com/blog/), technical papers, videos, webinars
C5	Distributed database supporting multiple data model Replication and automatic fail-over Capacity dynamically up and down ArangoDB cluster with distinct roles: Agents, coordinators, primary and secondary servers
C6	Apache License Version 2.0
C7	Latest Stable Release: 3.3.3 (20.12.2017)

Table 5.16: OrientDB (cf. Sec. 2.5.6).

Criterion	Value(s)
C1	Support for custom data structures (Java API)
C2	Project website: `https://orientdb.com/` GitHub: 114 contributors, 16.513 commits `https://github.com/orientechnologies/orientdb` Issue Tracker: 1073 open, 6.399 closed issues `https://github.com/orientechnologies/orientdb/issues` Active community (`https://orientdb.com/community/`)
C3	Three kinds of drivers: Native binary remote, HTTP-based REST/JSON, and Java-wrapped Support for Java, Node.js, PHP, Python, C, etc. Data stored as JSON, BLOB, Vertex, Edge
C4	OrientDB 2.2 manual `https://orientdb.com/docs/2.2.x/` Blog (https://orientdb.com/blog/), technical papers, videos, webinars
C5	Distributed database supporting multiple data model Replication and automatic fail-over Clustering via the Hazelcat open source project: Auto-discovery (Multicast, TCP-IP, Amazon), queues for request and responses, storage of metadata in distributed maps, distributed locks
C6	Apache License Version 2.0
C7	Latest Stable Release: 2.2.22 (05.02.2018)

Table 5.17: Couchbase (cf. Sec. 2.5.7).

Criterion	Value(s)
C1	Support for custom data structures as well as security and performance
C2	Project website: `https://www.couchbase.com` GitHub: 137 repositories `https://github.com/couchbase` Active community (`https://forums.couchbase.com/`)
C3	General access via N1QL queries Support for Java, .NET Node.js, PHP, Python, Ruby, Go, C, etc. Data stored as JSON and binary
C4	Couchbase documentation `https://developer.couchbase.com/documentation-archive` Blog (https://blog.couchbase.com/), technical papers, videos, webinars
C5	Distributed database supporting multiple data model Replication and automatic fail-over Load balancing Granularity of write locks Scalable request processing architecture
C6	Apache License Version 2.0
C7	Latest Stable Release: 5.1 (February 2018)

5.2.5 Device Management

This section covers the following technologies that are used regarding device management (cf. Sec. 2.6):

- Eclipse hawkBit (cf. Tab. 5.18),
- Eclipse Leshan (cf. Tab. 5.19),
- Eclipse Wakaama (cf. Tab. 5.20),
- Eclipse Vorto (cf. Tab. 5.21), and
- OGC SensorThings API (cf. Tab. 5.22).

Table 5.18: Eclipse hawkBit (cf. Sec. 2.6.1).

Criterion	Value(s)
C1	- Adaptation by adjusting the source code cotained in the GitHub repository
C2	Project website: `https://eclipse.org/hawkbit/` GitHub: `https://github.com/eclipse/hawkbit` Gitter: `https://gitter.im/eclipse/hawkbit` Hudson build: `https://hudson.eclipse.org/hawkbit/` Issue tracker: `https://github.com/eclipse/hawkbit/issues` Mailing list: `https://accounts.eclipse.org/mailing-list/hawkbit-dev`
C3	- The Direct Device Integration API is based on HTTP and JSON - Integration with Eclipse Hono is planned for the future - Usage of the existing clients SWupdate and rauc-hawkbit
C4	General Documentation: `https://eclipse.org/hawkbit/documentation/overview/introduction.html` YouTube Tutorial: `https://www.youtube.com/watch?v=g-dhKMaaanE`
C5	- Allows to update a large number of devices in paralell (Technical scalability) - Enhances the feature set of the set of devices (Functional scalability)
C6	Eclipse Public License 1.0
C7	Latest Stable Release: 0.2.0M3 (03.04.2017)

Table 5.19: Eclipse Leshan (cf. Sec. 2.6.2).

Criterion	Value(s)
C1	- Adaptation by adjusting the source code cotained in the GitHub repository
C2	Project website: `https://www.eclipse.org/leshan/` GitHub: `https://github.com/eclipse/leshan` Issue tracker: `https://github.com/eclipse/leshan/issues` Mailing list: `https://accounts.eclipse.org/mailing-list/leshan-dev`
C3	- Client and server implementation of the OMA Lightweight M2M protocol in Java - Based on Eclipse Californium and Eclipse Scandium
C4	General Documentation: `https://github.com/eclipse/leshan/wiki` Getting Started: `https://github.com/eclipse/leshan/wiki/Getting-started`
C5	- LWM2M is desgined for the usage by devices equipped with limited resources - It is expected that thousands of devices can be connected to a single Leshan instance
C6	Eclipse Distribution License 1.0 (BSD) Eclipse Public License 1.0
C7	Latest Stable Release: 1.0.0-M5 (19.01.2018)

Table 5.20: Eclipse Wakaama (cf. Sec. 2.6.3).

Criterion	Value(s)
C1	- Adaptation by adjusting the source code cotained in the GitHub repository
C2	Project website: `https://eclipse.org/wakaama/` GitHub: `https://github.com/eclipse/wakaama` Issue tracker: `https://github.com/eclipse/wakaama/issues` Mailing list: `https://accounts.eclipse.org/mailing-list/wakaama-dev`
C3	- Client and server implementation of the OMA Lightweight M2M protocol in C
C4	Getting Started: `https://eclipse.org/wakaama/index.html#getting-started`
C5	- LWM2M is desgined for the usage by devices equipped with limited resources
C6	Eclipse Distribution License 1.0 (BSD) Eclipse Public License 1.0
C7	Latest Release: Initial Commit (08.07.2014)

Table 5.21: Eclipse Vorto (cf. Sec. 2.6.4).

Criterion	Value(s)
C1	- Adaptation by adjusting the source code cotained in the GitHub repository - New IoT device descriptions can be added to the Vorto Repository (http://vorto.eclipse.org/#/)
C2	Project website: https://www.eclipse.org/vorto/ GitHub: https://github.com/eclipse/vorto Issue tracker: https://github.com/eclipse/vorto/issues/ Blog: https://www.eclipse.org/vorto/blog/index.html Twitter: https://twitter.com/VortoIoT Mailing list: https://accounts.eclipse.org/mailing-list/vorto-dev Forum: http://www.eclipse.org/forums/index.php?t=thread&frm_id=303 Wiki: https://wiki.eclipse.org/Vorto
C3	- Integration into the Eclipse IDE - Information models are defined using a domain-specific language
C4	General documentation (comprehensive): https://www.eclipse.org/vorto/documentation/
C5	- An information model can be applied to every IoT device of the same type.
C6	Eclipse Public License 1.0
C7	Latest Release: 0.10.0.M1 (05.07.2017)

Table 5.22: OGC SensorThings API (cf. Sec. 2.6.5).

Criterion	Value(s)
C1	- It targets the semantics of messages transmitted using exisiting IoT protocols - Up to this point, no official implementation has been released
C2	Project website: https://ogc-iot.github.io/ogc-iot-api/ GitHub: https://github.com/opengeospatial/sensorthings Issue tracker: https://github.com/opengeospatial/sensorthings/issues
C3	- It complements exisiting IoT data transfer protocols
C4	General documentation: http://www.opengeospatial.org/standards/sensorthings OGC SensorThings API Part 1 (Sensing): http://docs.opengeospatial.org/is/15-078C6/15-078C6.html
C5	- Up to this point, only the first part of the corresponding standard that targets the sensing has been relseased
C6	License agreement: see http://docs.opengeospatial.org/is/15-078C6/15-078C6.html
C7	Latest Release: 1.0 (26.07.2016)

5.2.6 Message Gateway

In the following, the given criteria are applied to the technologies from Section 2.7 for realizing a message gateway. An additional criteria for the message gateway is the *real-time* capability (C8), which is relevant for distinct use cases, such as the User Story 03 "'*Wrong Way Driver Warning*"' (cf. Section 3.3) or User Story 22 "'*Emergency Braking & Evading Assistance System (EBEAS)*"' (cf. Section 3.22). For such use cases, a real-time processing of forwarded data from multiple vehicles may be necessary.

- Eclipse Hono (cf. Tab. 5.23),
- Apache Kafka (cf. Tab. 5.24),
- and Eclipse Mosquitto (cf. Tab. 5.25)

Table 5.23: Eclipse Hono (cf. Sec. 2.7.1).

Criterion	**Value(s)**
C1	Custom protocol adapters, seamless integration of any AMQP 1.0-compatible message broker
C2	Project website: `https://www.eclipse.org/hono` GitHub: 18 contributors, 1.308 commits `https://github.com/eclipse/hono` Issue Tracker: 34 open, 214 closed issues `https://github.com/eclipse/hono/issues` Mailing list (`https://accounts.eclipse.org/mailing-list/hono-dev`), Chat (IRC), Forum (0 threads)
C3	Built upon AMQP, protocol adapters for MQTT and REST and custom protocol adapters via Hono's API
C4	User and developer guide (`https://www.eclipse.org/hono/user-guide/`) and API documentation Presentations
C5	Horizontally scalable microservice architecture, EnMasse[1] for horizontal scalability of dispatch routers and brokers
C6	Eclipse Public License 1.0
C7	Latest stable release: 0.5-M10 (24.10.2017); some releases (10)
C8	—

Table 5.24: Apache Kafka (cf. Sec. 2.7.2).

Criterion	Value(s)
C1	Capabilities to configure broker, topics, producer, consumer, and connections/streams
C2	Project website: `https://kafka.apache.org/` GitHub: 374 contributors, 4.584 commits `https://github.com/apache/kafka` Issue Tracker: 6476 issues at all `https://issues.apache.org/jira/projects/KAFKA/issues` Different mailing lists (`https://kafka.apache.org/contact`), Google groups (92 threads), IRC, use cases, meetups, summits
C3	Binary protocol over TCP with Java and Scala API implementation
C4	Kafka 1.0 documentation (`https://kafka.apache.org/documentation/`)
C5	Designed for high volume publish-subscribe messages and streams: scaling writes and reads by sharing topic logs into partitions
C6	Apache License v2
C7	Latest stable release: 0.11.0.2 (16.11.2017); frequent releases (64)
C8	Capabilities for real-time stream processing in mind

Table 5.25: Eclipse Mosquito (cf. Sec. 2.7.3).

Criterion	Value(s)
C1	Broker configuration: custom authentication and access control, options to configure bridges
C2	Project website: `https://projects.eclipse.org/projects/technology.mosquitto` GitHub: 25 contributors, 646 commits `https://github.com/eclipse/mosquitto` Issue Tracker: 179 open, 328 closed issues `https://github.com/eclipse/mosquitto/issues` Mailing list (https://accounts.eclipse.org/mailing-list/mosquitto-dev), forum (85 threads)
C3	C-based API (*libmosquitto*) for MQTT
C4	Mosquitto documentation (`https://mosquitto.org/documentation/`) Blog (`https://mosquitto.org/`)
C5	Efficient C implementation (1000 clients == 3MB RAM), support for horizontal scalability through a MQTT bridge
C6	Eclipse Public License 1.0
C7	Latest stable release: 1.4.14 (29.05.2017); some releases (19)
C8	—

5.3 Technology Selection

In order to rate the suitability of the respective technology for the generic building blocks of Figure 4.1, the following scores depicted in Table 5.26 are applied to each criterion, if possible.

Table 5.26: Evaluation criteria scores

Value	Qualitative
-1	Strongly contradicts the criterion
0	Partly fulfills the criterion
1	Strongly fulfills the criterion

Each technology is assigned with an aggregated score based on the individual scores v, which reflects its suitability in the context of the cloud platform architecture. For this purpose, a specific aggregation function s is applied to cumulate the scores for the individual criteria (cf. Equ. 5.1).

$$s = v_{C1} + v_{C2} + v_{C3} + v_{C4} + v_{C5} + v_{C7} + v_{Cx} \tag{5.1}$$

Note that the criterion **C6** *Software License* is assigned with a binary scale, reflecting that the license applied to a technology either can be used by third party code redistributed by Eclipse Foundation Projects or not. As a result, this criterion is represented by a separate function f, that may override the value of s (cf. Equ. 5.2).

$$f(v_{C6}, s) = \begin{cases} s, & \text{if } v_{C6} = 0, 1 \\ -\infty, & \text{if } v_{C6} = -1 \end{cases} \tag{5.2}$$

Note that the value $-\infty$ will prevent that a corresponding technology is selected.

5.3.1 IoT Cloud Platform

Table 5.27 represents the evaluation of the only technology for managing IoT Cloud platforms. It considers results from 5.1. In this section, we have variety of other existing IoT Cloud platforms that have been used partially in designing our reference architecture.

Table 5.27: The evaluation of IoT Cloud Platform technologies

	Eclipse Kapua
v_{C1}	1
v_{C2}	0
v_{C3}	1
v_{C4}	1
v_{C5}	0
v_{C7}	-1
s	2
v_{C6}	1
$f(v_{C6}, s)$	2

5.3.2 Application and Service Integration

Table 5.27 represents the evaluation of the existing technologies for the *Application and Service Integration*. It considers results from Table 5.2 to 5.6. In this section, we have variety of technologies from matured technologies such as Apache Camel to an archived technology such as Eclipse Mihini. Apache Camel and Spring Integration technologies can support most of the user specific needs and are can be adopted to the context. All technologies (except Mihini) support standardized and common interfaces ranging from MQTT, AMQP to the REST API. Our analysis shows that Apache Camel and Spring Integration have a good and reliable community in support containing large number of contributors and commits in GitHub; however, the contribution towards Eclipse Ditto and Eureka is increasing. In addition, Apache Camel and Eureka provide mechanisms of scalability on increasing load of data or amount of connected devices. Spring Integration needs adoption of other technologies to scale. Most of the technologies tried to establish basic documentations including general description, developers and users' guides, as well as installation, demo and deployment. Technologies are based on two major open source licenses as Eclipse Public License (EPL) and Apache License 2.0. Eclipse Mihini has one LGPL library that should be considered in future development. Considering the interval of the releases Apache Camel, Spring Integration and Eureka have been maintained regularly and are stable considering the number of releases. According to our evaluation, we propose that Apache Camel is an appropriate technology for APPSTACLE as it meets all evaluation criteria; however, other matured solutions can be adopted with vital consideration regarding their shortages.

Table 5.28: The evaluation of Service Integration technologies

	Eclipse Ditto	**Mihini**	**Eureka**	**Apache Camel**	**Spring Integration**
v_{C1}	-1	0	-1	1	1
v_{C2}	0	-1	0	1	1
v_{C3}	1	0	1	1	1
v_{C4}	1	-1	1	1	1
v_{C5}	0	-1	1	1	0
v_{C7}	-1	-1	1	1	1
s	0	-4	3	6	5
v_{C6}	1	1	1	1	1
$f(v_{C6}, s)$	0	-4	3	6	5

5.3.3 Data Analytics and Visualization

Based on the assessment in Tab. 5.29, Apache Flink and Grafana are the preferable choices regarding the analysis of large amounts of data as well as the corresponding visualization. Apache Storm is only assessed with the value 0 regarding adaptibility, because it does not allow the analysis of batch data sets. This drawback is mitigated by Apache Flink, which allows to analyse streaming as well as batch data sets. The community around Eclipse BIRT appears to be limited, which leads to an assignment of 0 of the corresponding criterion.

Table 5.29: Application of evaluation criteria to the data analytics and visualization technologies.

	Apache Storm	**Apache Flink**	**Eclipse BIRT**	**Grafana**
v_{C1}	0	1	1	1
v_{C2}	0	1	0	1
v_{C3}	1	1	1	1
v_{C4}	1	1	1	1
v_{C5}	1	1	0	1
v_{C7}	1	1	1	1
s	4	6	4	6
v_{C6}	1	1	1	1
$f(v_{C6}, s)$	4	6	4	6

5.3.4 Data Storage and Management

Table 5.30 depict the assessment of the according DBMS technologies based on the results from Table 5.11 to 5.14. In general, all of the considered technologies are matured, well documented, and maintained by an existing community. Furthermore, they are accessible through standardized and common interfaces as well as partly exhibit capabilities for adaptability. In the nature of NoSQL databases, they further features horizontal scaling. This features make them appropriate for the usage in an IoT cloud platform. However, the usage of a distinct NoSQL databases heavily depends on the according use case. For example, document-oriented databases focus on the storage of heterogeneous data in a generic manner, which make them suitable for use cases that produces unstructured data and require some flexibility, e. g. User Story 04 "‘*Augment vehicle functionality*"’ (cf. Section 3.4) may affect various configuration parameters and require access to different sensors etc. By contrast, a graph database like Neo4j allows for an efficient querying of related data in large data sets, e. g. User Story 16 "‘*Social Media*"’(cf. Section 3.16). A TSDB focus on massive storage of time series data and is therefore suitable for use cases like vehicle tracking (cf. User Story 02 in Section 3.2).

Thus, we propose to use a multi-model database and select the database based on the according use case instead of relying on a specific DBMS solution. All of the considered multi-model databases in Table 5.31 support one or more data model, but using only one data model for the main database structure. Thereby, the most commonly used data model is a document-oriented data model, which allow to stores different data in a generic manner. This enables applications to store different data and query the data with one method. In general, the query method should not be completely different conventional methods. All considered multi-model databases have their own query languages, but all query languages are not very different from SQL. In this way, very flexible structure is provided in the queries. One of the most important features is scalability in the storage of very large data. Scalability should be focused on consistency, availability and partition tolerance for distributed systems.

Table 5.30: Application of the evaluation criteria on DBMS technologies

	CouchDB	**MongoDB**	**Neo4j**	**InfluxDB**
v_{C1}	0	0	-1	0
v_{C2}	1	1	1	1
v_{C3}	1	0	1	1
v_{C4}	1	1	1	1
v_{C5}	1	1	0	-1
v_{C7}	0	1	1	1
s	4	4	3	3
v_{C6}	1	1	0	1
$f(v_{C6}, s)$	4	4	3	3

Table 5.31: Application of the evaluation criteria on multi-model databases

	AnrangoDB	**MongoDB**	**Neo4j**
v_{C1}	0	0	0
v_{C2}	1	1	0
v_{C3}	0	1	1
v_{C4}	0	1	1
v_{C5}	0	1	0
v_{C7}	0	1	1
s	2	5	3
v_{C6}	1	1	1
$f(v_{C6}, s)$	2	5	3

5.3.5 Device Management

Table 5.32: The evaluation of technologies for device management.

	hawkBit	**Leshan**	**Wakaama**	**Vorto**	**SensorThings API**
v_{C1}	1	1	1	1	0
v_{C2}	1	1	0	1	0
v_{C3}	1	1	1	1	1
v_{C4}	1	1	0	1	1
v_{C5}	1	1	1	1	0
v_{C7}	1	1	-1	1	0
s	6	6	2	6	2
v_{C6}	1	1	1	1	-1
$f(v_{C6}, s)$	6	6	2	6	$-\infty$

The development efforts within the Eclipse Wakaama project seem to be limited. There has no official release been made yet and the number of commits to the repository declined in the recent past. Furthermore, the amount of documentation available is rather small. This results

in a low assessment of the project. OGC SensorThings API is a standard that has not yet been implemented. The community around this standard appears to be limited. The documentation regarding OGC SensorThings API is represented by an formal standard description. Up to this point, only version 1.0 of the standard has been released. Due to its custom license agreement, the OGC SensorThings API may cause legal issues. For this reason the usage of this standard with respect to this research project is discarded. The remaining technologies, Eclipse hawkBit, Eclise Leshan, and Eclipse Vorto have a total score of 6, which implies their suitability.

5.3.6 Message Gateway

The message gateway is one of the central components in an IoT architecture and requires a thorough investigation regarding the applicability in the APPSTACLE cloud backend. Table 5.33 show the evaluation results for each technology. Apache Kafka is a platform for stream processing with particular focus on realtime capabilities. The project is matured and used in a lot of industry platforms, e. g. Netflix or eBay. Mosquitto depict a MQTT broker and is realized as Eclipse project with a small, but active community. In contrast to Kafka, Mosquitto is realized in C and do not exhibit any capabilities for real-time processing and has minor support for horizontal scalability and adaptability. The messaging infrastructure in Hono supports MQTT and HTTP out of the box and various protocols through custom adapters. Through its microservice architecture, Hono is designed with horizontal scalability in mind and allows also to integrate any AMQP 1.0-compatible message broker. To provide and manage identities of connected devices, Hono has also capabilities for device registration and authentication, while the telemetry and command & control API covers different aspects of device registry. This additional features in contrast to Kafka and Mosquitto makes Hono suitable for the application in the automotive domain. Although there is no support for real-time processing by default, a suitable message broker could be used to realize such a behavior. Thus, we propose to use Hono as message gateway as it is more than a pure messaging component, is maintained by an active community, and consists of an appropriate software license.

Table 5.33: Application of the evaluation criteria on message gateway technologies

	Eclipse Hono	**Apacha Kafka**	**Eclipse Mosquitto**
v_{C1}	1	0	0
v_{C2}	0	1	-1
v_{C3}	1	0	0
v_{C4}	1	1	0
v_{C5}	1	1	0
v_{C7}	0	1	0
v_{C8}	-1	1	-1
s	3	5	-2
v_{C6}	1	1	1
$f(v_{C6}, s)$	3	5	-2

6 Conclusion

Figure 6.1 shows the mapping of the evaluation results to the generic building blocks of architecture specification in Figure 4.1 and therewith the selection of specific tools. In general, the applied technologies are open source with an according software license, maintained by a community, include documentation, and are regularly updated.

The *Core Layer* basically consists of a message gateway, core services, a data management tool, and technologies for identifying, representing, and managing devices. As the message gateway is the single point of interaction with the environment, it has to support at least the protocols used by the in-vehicle gateway. Nevertheless, the message gateway should also be able to adapt future protocols, which may emerge due to the availability of new technologies. By using Eclipse Hono, not only a flexible and scalable message gateway is provided, but also capabilities for device registration and authentication. Thereby, the seamless interaction with the in-vehicle gateway is achieved through the protocol adapter concept of Eclipse Hono, which allows to map any messaging protocol to the underlying AMQP protocol via existing or custom protocol adapters. An important requirement within the automotive domain is the roll out of software updates to constrained edge devices OTA. The evaluation in Section 5.3.5 shows that Eclipse hawkBit is a suitable technology for realizing such a device management. Furthermore, hawkBit supports different protocols, e. g. REST and AMQP, and is thus compatible with Eclipse Hono as message gateway. Although Eclipse Ditto is a relatively new project that is subject to ongoing major development activities, it provides valuable information of devices for the realization of domain-specific services and thus is also incorporated into the APPSTACLE core layer. As depicted in Section 5.3.4, the data management should not rely on a single DBMS technology, but rather make use of SQL and different types of NoSQL databases that are integrated into a single backend. Multi-model databases are one way to achieve that. For example, OrientDB[1] represents an open source multi-model database written in Java with support for SQL as well as graph, document, key/value, and object models. Another promising option is to use Eclipse JNoSQL[2], which defines a set of APIss to interact with various NoSQL databases, such as mongoDB, neo4j, and even multi-model databases like OrientDB. However, a sophisticated integration requires more extensive investigations and evaluations.

The technologies for realizing an identity management will be defined within the security document. Nevertheless, Hono is already shipped with some functionality for device authentication whereas Keycloak (cf. Section 2.8.2) implements authentication and authorization mechanisms.

The *Data Analytic & Visualization Layer* mainly rely on the data stored in the data management and either focus on representing data in a valuable way or to process big data for further analysis. The evaluation in Chapter 5 has shown that Eclipse BIRT can be used to generate business reports, while Grafana allows to visualize data within flexible dashboards. In contrast, big data analysis are challenging and need holistic investigations. Apache Flink is a suitable

[1] `http://orientdb.com/`
[2] `http://www.jnosql.org/`

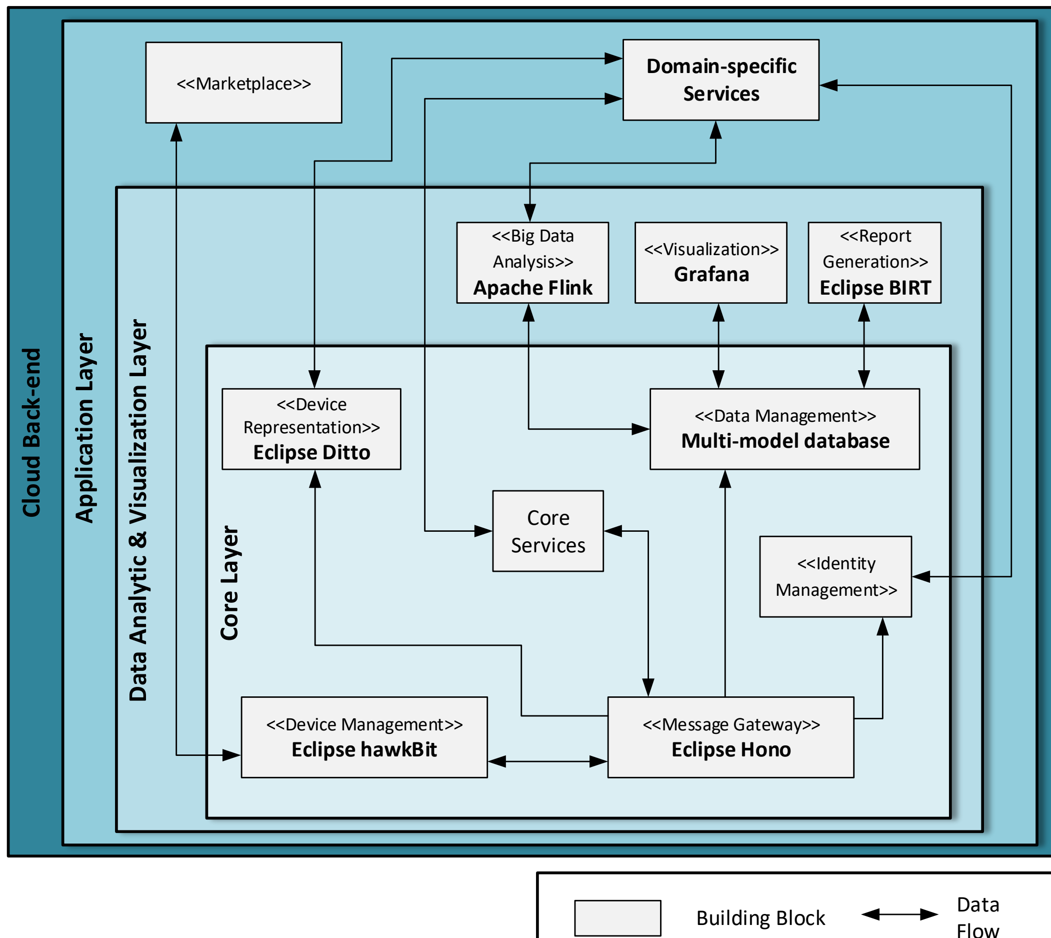

Figure 6.1: The cloud platform architecture with specific technology

technology for such tasks as it is scalable and allows to analyze data streams as well as batch sets.

The uppermost layer, i.e., *Application Layer*, includes domain-specific services that are developed according to the use case and a market place that offers applications to users. Due to the fact that no existing and open source technology for realizing such a market place could be found, this component will probably be a major custom implementation within APPSTACLE.

6.1 Subsequent activities

Based on this initial architecture specification, the next steps are the integration and implementation of the according technologies within a platform prototype to evaluate the chosen technologies, identify required extensions and additional components, and finally form a platform applicable to the various use cases defined in Section 3. Eclipse projects have been proven to be a successful and effective means to develop, maintain, and disseminate de-facto standard development tools. This includes not only the hosting of open source projects, but also best practices for continuous integration, as well as build and test environments to enable long term

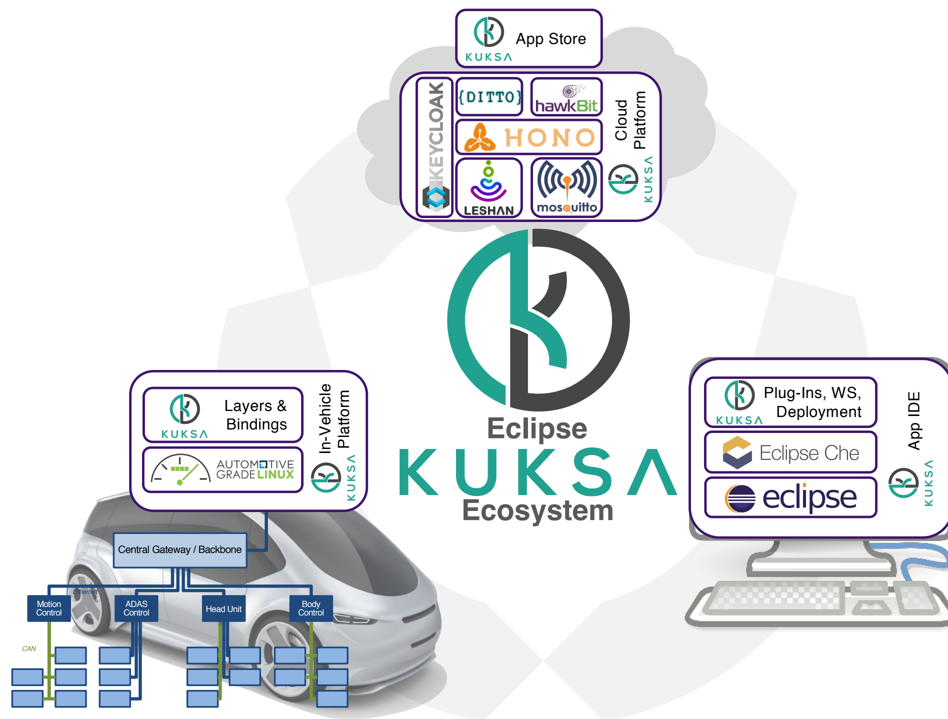

Figure 6.2: Proposed architecture of Eclipse Kuksa

support and maintenance. Accordingly, the Eclipse Kuksa[3] is already proposed as new Eclipse project for the results of APPSTACLE.

Figure 6.2 depicts the Eclipse Kuksa ecosystem including the in-vehicle platform from work package 1, the cloud platform specified in this deliverable document (D3.1), and the App development IDE, which allows to develop in-vehicle applications on top of Automotive Grade Linux as well as domain-specific cloud services in form of spring boot applications. Having the different aspects of APPSTACLE within a common platform allows to steer the development activities and fosters the interoperability.

[3]https://projects.eclipse.org/proposals/eclipse-kuksa

Bibliography

[1] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys Tutorials*, 17(4):2347–2376, 2015.

[2] Apache. Apache Storm documentation v1.1.1. http://storm.apache.org/releases/1.1.1/index.html, 2017. Accessed: 2017-08-22.

[3] M. D. Assunção, R. N. Calheiros, S. Bianchi, M. A. Netto, and R. Buyya. Big data computing and clouds: Trends and future directions. *Journal of Parallel and Distributed Computing*, 79:3–15, 2015.

[4] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Comput. Netw.*, 54(15):2787–2805, Oct. 2010.

[5] C. Bormann, A. P. Castellani, and Z. Shelby. Coap: An application protocol for billions of tiny internet nodes. *IEEE Internet Computing*, 16(2):62–67, Mar. 2012.

[6] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1789–1792, May 2016.

[7] K. Chodorow. *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 2013.

[8] D. Crockford. The application/json media type for javascript object notation (json). RFC 4627, IETF, 7 2006.

[9] M. Díaz, C. Martín, and B. Rubio. State-of-the-art, challenges, and open issues in the integration of internet of things and cloud computing. *Journal of Network and Computer Applications*, 67:99–117, 2016.

[10] J. Dorsey. Big data in the driver's seat of connected car technological advances. http://news.ihsmarkit.com/press-release/country-industry-forecasting/big-data-drivers-seat-connected-car-technological-advance, nov 2013. Accessed: 2017-06-11.

[11] Eclipse. Beginning BIRT: A Practical Introduction. https://www.eclipse.org/community/eclipse_newsletter/2015/september/article3.php, 2015. Accessed: 2017-08-26.

[12] Eclipse. Open source stack for IoT Cloud Platforms. https://iot.eclipse.org/cloud/, 2017. Accessed: 2017-07-18.

[13] Eclipse Foundation. Open Source Software for Industry 4.0. https://iot.eclipse.org/resources/white-papers/Eclipse%20IoT%20White%20Paper%20-%20Open%20Source%20Software%20for%20Industry%204.0.pdf, 2017. Accessed: 2017-09-15.

[14] Erik T. Heidt. Gartner: 2017 Planning Guide for the Internet of Things. https://www.gartner.com/binaries//content/assets/events/keywords/catalyst/catus8/2017_planning_guide_for_the__iot.pdf, 2016. Accessed: 2017-09-01.

[15] G. Fersi. Middleware for internet of things: A study. In *Distributed Computing in Sensor Systems (DCOSS), 2015 International Conference on*, pages 230–235. IEEE, 2015.

[16] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, UNIVERSITY OF CALIFORNIA, IRVINE, 2000. AAI9980887.

[17] Gartner, Inc. Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016. http://www.gartner.com/newsroom/id/3598917, 2017. Accessed: 2017-10-01.

[18] GrafanaLabs. Grafana documentation, OCtober 2017.

[19] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013.

[20] J. Han, E. Haihong, G. Le, and J. Du. Survey on nosql database. In *Pervasive computing and applications (ICPCA), 2011 6th international conference on*, pages 363–366. IEEE, 2011.

[21] Intel. The Intel IoT Platform: Architecture Specification White Paper. https://www.intel.de/content/dam/www/public/us/en/documents/white-papers/iot-platform-reference-architecture-paper.pdf, 2016. Accessed: 2017-09-12.

[22] ISO. Iso/iec 20922:2016 mqtt v3.1.1, 06 2016.

[23] ITU-T. Itu-t y.2060: Overview of the internet of things, June 2012.

[24] N. Jatana, S. Puri, M. Ahuja, I. Kathuria, and D. Gosain. A survey and comparison of relational and non-relational database. *International Journal of Engineering Research & Technology*, 1(6):1–5, 2012.

[25] V. Karagiannis, P. Chatzimisios, F. Vazquez-Gallego, and J. Alonso-Zarate. A survey on application layer protocols for the internet of things. *Transaction on IoT and Cloud Computing*, 3(1):11–17, 2015.

[26] S. Kozielski, D. Mrozek, P. Kasprowski, B. Malysiak-Mrozek, and D. Kostrzewa, editors. *Beyond Databases, Architectures and Structures. Towards Efficient Solutions for Data Analysis and Knowledge Representation - 13th International Conference, BDAS 2017, Ustroń, Poland, May 30 - June 2, 2017, Proceedings*, volume 716 of *Communications in Computer and Information Science*, 2017.

[27] S. Krčo, B. Pokrić, and F. Carrez. Designing iot architecture(s): A european perspective. In *2014 IEEE World Forum on Internet of Things (WF-IoT)*, pages 79–84, March 2014.

[28] D. Laney. 3D data management: Controlling data volume, velocity, and variety. Technical report, META Group, February 2001. Accessed: 2017-06-26.

[29] J. Leibiusky, G. Eisbruch, and D. Simonassi. *Getting Started with Storm*. O'Reilly Media, Inc., 2012.

[30] R. A. Light. Mosquitto: server and client implementation of the mqtt protocol. *Journal of Open Source Software*, 2(13), 2017.

[31] S.-W. Lin, B. Miller, J. Durand, G. Bleakley, A. Chigani, R. Martin, B. Murphy, and M. Crawford. The Industrial Internet of Things Volume G1: Reference Architecture V 1.8. `http://www.iiconsortium.org/IIC_PUB_G1_V1.80_2017-01-31.pdf`, 2017. Accessed: 2017-09-11.

[32] J. a. R. Lourenço, B. Cabral, P. Carreiro, M. Vieira, and J. Bernardino. Choosing the right nosql database for the job: a quality attribute evaluation. *Journal of Big Data*, 2(1):18, 2015.

[33] A. Luckow, K. Kennedy, F. Manhardt, E. Djerekarov, B. Vorster, and A. Apon. Automotive big data: Applications, workloads and infrastructures. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 1201–1210, Oct 2015.

[34] Microsoft Corporation. Microsoft Azure IoT Reference Architecture. `http://download.microsoft.com/download/A/4/D/A4DAD253-BC21-41D3-B9D9-87D2AE6F0719/Microsoft_Azure_IoT_Reference_Architecture.pdf`, 2016. Accessed: 2017-09-14.

[35] J. J. Miller. Graph database applications and concepts with neo4j. In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*, volume 2324, page 36, 2013.

[36] J. Mineraud, O. Mazhelis, X. Su, and S. Tarkoma. Contemporary internet of things platforms. *arXiv preprint arXiv:1501.07438*, 2015.

[37] MongoDB. Mongodb architecture guide. Technical Report MongoDB 3.4, MongoDB Inc., June 2017. Accessed: 2017-07-04.

[38] MongoDB. The MongoDB 3.4 Manual. `https://docs.mongodb.com/manual/`, 2017. Accessed: 2017-07-16.

[39] Neo4j. The Neo4j Developer Manual v3.2. `https://neo4j.com/docs/developer-manual/`, 2017. Accessed: 2017-07-17.

[40] OASIS. Oasis advanced message queuing protocol (amqp) tc.

[41] OMA. Lightweight machine to machine technical specification approved version 1.0.1 – 04 july 2017, 2017.

[42] J. Pokorny. Nosql databases: A step to database scalability in web environment. In *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services*, iiWAS '11, pages 278–283, New York, NY, USA, 2011. ACM.

[43] Qpid project. The apache qpid project, 2017.

[44] P. P. Ray. A survey of iot cloud platforms. *Future Computing and Informatics Journal*, 1(1):35 – 46, 2016.

[45] Z. Shelby. Embedded web services. *IEEE Wireless Communications*, 17(6):52–57, December 2010.

[46] Z. Shelby, K. Hartke, and C. Bormann. The Constrained Application Protocol (CoAP). RFC 7252, June 2014.

[47] A. M. C. Souza and J. R. A. Amazonas. A novel smart home application using an internet of things middleware. In *Smart SysTech 2013; European Conference on Smart Objects, Systems and Technologies*, pages 1–7, June 2013.

[48] Steve H. L. Liang and Chih-Yuan Huang and Tania Khalafbeigi. Ogc sensorthings api part i:sensing. OGC® Implementation Standard (`http://docs.opengeospatial.org/is/15-078r6/15-078r6.html`), 2016.

[49] C. Team. *CouchDB 2.0 Reference Manual*. Samurai Media Limited, United Kingdom, 2015.

[50] The ActiveMQ Project. The apacha activemq project, 2017.

[51] The Eclipse BIRT Team. Eclipse BIRT documentation. `https://www.eclipse.org/birt/documentation/`, 2017. Accessed: 2017-08-25.

[52] The Eclipse Ditto Team. Eclipse ditto project proposal. `https://projects.eclipse.org/proposals/eclipse-ditto`, 2017. Accessed: 2017-07-03.

[53] The Eclipse Hawkbit Team. Eclipse hawkbit documentation, August 2017.

[54] The Eclipse Hono Team. Eclipse hono documentation. `https://www.eclipse.org/hono/`, 2017. Accessed: 2017-07-03.

[55] The Eclipse Kapua Team. Eclipse Kapua documentation. `https://www.eclipse.org/kapua/documentation.php`, 2017. Accessed: 2017-10-08.

[56] The Eclipse Leshan Team. Leshan project page, 2017.

[57] The Eclipse Mosquitto Team. The eclipse mosquitto project page, 2017.

[58] The Eclipse Paho Team. The eclipse paho project page, 2017.

[59] The Eclipse Vorto Team. Vorto project documentation. `http://www.eclipse.org/vorto/`, July 2017. Accessed: 2017-07-17.

[60] The Eclipse Wakaama Team. Eclipse wakaama project page, 2017.

[61] The Eureka Team. Eureka 1.0 documentation. `https://github.com/Netflix/eureka/wiki`, August 2017. Accessed: 2017-09-5.

[62] The Influxdata Team. Influxdb version 1.3 documentation, August 2017.

[63] K. M. M. Thein. Apache kafka: Next generation distributed messaging system. *International Journal of Scientific Engineering and Technology Research*, 3(47):9478–9483, 2014.

[64] P. Waher. *Learning Internet of Things*. Packt Publishing, 2015.

[65] J. Wang, H. Xiong, Y. Ishikawa, J. Xu, and J. Zhou. Web-age information management - 14th international conference, WAIM, 2013.

[66] G. Wu, S. Talwar, K. Johnsson, N. Himayat, and K. D. Johnson. M2m: From mobile to embedded internet. *IEEE Communications Magazine*, 49(4):36–43, April 2011.