## DELIVERABLE

### D4.5 Context-Aware Actions and Self-Adaptation Framework v1

<table>
<thead>
<tr>
<th><strong>Project Acronym:</strong></th>
<th>bloTope</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Project title:</strong></td>
<td>Building an IoT Open Innovation Ecosystem for Connected Smart Objects</td>
</tr>
<tr>
<td><strong>Grant Agreement No.</strong></td>
<td>688203</td>
</tr>
<tr>
<td><strong>Website:</strong></td>
<td><a href="http://www.bloTope-project.org">www.bloTope-project.org</a></td>
</tr>
<tr>
<td><strong>Version:</strong></td>
<td>1.0</td>
</tr>
<tr>
<td><strong>Date:</strong></td>
<td>2017-07-03</td>
</tr>
<tr>
<td><strong>Responsible Partner:</strong></td>
<td>CSIRO</td>
</tr>
<tr>
<td><strong>Editor:</strong></td>
<td>Arkady Zaslavsky</td>
</tr>
<tr>
<td><strong>Contributing Partners:</strong></td>
<td>CSIRO, EPFL, Aalto, Fraunhofer, Uni.Lu</td>
</tr>
<tr>
<td><strong>Dissemination Level:</strong></td>
<td>Public</td>
</tr>
<tr>
<td></td>
<td>Confidential – only consortium members and European Commission Services</td>
</tr>
</tbody>
</table>
# Revision History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Author</th>
<th>Organization</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>23/02/2017</td>
<td>Arkady Zaslavsky</td>
<td>CSIRO</td>
<td>Initial ToC</td>
</tr>
<tr>
<td>0.2</td>
<td>8/05/2017</td>
<td>Arkady Zaslavsky</td>
<td>CSIRO</td>
<td>Draft#01</td>
</tr>
<tr>
<td>0.21</td>
<td>8/05/2017</td>
<td>Alexey Medvedev</td>
<td>CSIRO</td>
<td>Draft#01.1</td>
</tr>
<tr>
<td>0.22</td>
<td>8/05/2017</td>
<td>Ali Hassani</td>
<td>CSIRO</td>
<td>Draft#01.2</td>
</tr>
<tr>
<td>0.3</td>
<td>5/06/2017</td>
<td>Arkady Zaslavsky</td>
<td>CSIRO</td>
<td>Draft#02</td>
</tr>
<tr>
<td>0.3a</td>
<td>7/06/2017</td>
<td>Kristian Bäckström</td>
<td>CT</td>
<td>Review</td>
</tr>
<tr>
<td>0.4</td>
<td>7/06/2017</td>
<td>Alexey Medvedev</td>
<td>CSIRO</td>
<td>Draft02a</td>
</tr>
<tr>
<td>0.5</td>
<td>11/06/2017</td>
<td>Arkady Zaslavsky</td>
<td>CSIRO</td>
<td>Draft03</td>
</tr>
<tr>
<td>0.6</td>
<td>19/06/2017</td>
<td>Niklas Kolbe/Sylvain Kubler</td>
<td>Uni.Lu</td>
<td>Review</td>
</tr>
<tr>
<td>0.9</td>
<td>26/06/2017</td>
<td>Arkady Zaslavsky</td>
<td>CSIRO</td>
<td>Final draft</td>
</tr>
<tr>
<td>1.0</td>
<td>30/06/2017</td>
<td>Arkady Zaslavsky</td>
<td>CSIRO</td>
<td>Final deliverable</td>
</tr>
</tbody>
</table>

Every effort has been made to ensure that all statements and information contained herein are accurate, however the bioTope Project Partners accept no liability for any error or omission in the same.
Table of Contents

1. Introduction ........................................................................................................... 13
   1.1. Scope................................................................................................................ 13
   1.2. Audience......................................................................................................... 13
   1.3. Summary ........................................................................................................ 13
   1.4. Structure ....................................................................................................... 13

2. Context as a Service – architectural principles ....................................................... 14
   2.1. A motivating scenario..................................................................................... 16
   2.2. CDQL ............................................................................................................. 17
       2.2.1. CDM ...................................................................................................... 17
       2.2.2. CQL ...................................................................................................... 19
   2.3. CoaaS and CDQL Framework ....................................................................... 22
   2.4. CQL EBNF .................................................................................................... 26
   2.5. Development and execution environment for CDQL .................................... 28
       2.5.1. Development Environments and Tools.................................................... 28
       2.5.2. Architecture of the Proof-of-Concept Implementation.............................. 28
       2.5.3. Node-red Integration ............................................................................. 36

3. CSMS - Context Storage Management System ....................................................... 39
   3.1. Context storage requirements ....................................................................... 41
   3.2. Existing context representation and storage technologies .......................... 42
   3.3. Data management in existing IoT platforms ................................................. 45
       3.3.1. Overview of data management in existing IoT platforms......................... 45
       3.3.2. Commercial IoT platforms ..................................................................... 45
       3.3.3. Academic/research projects .................................................................... 47
       3.3.4. Benchmarking and Evaluation ................................................................ 49
       3.3.5. Discussion ............................................................................................... 50
   3.4. Proposed Context Storage Management System ......................................... 50
   3.5. 5.1 Conceptual architecture of Context Storage Management System .......... 52
   3.6. Components of CSMS Architecture ............................................................... 53
       3.6.1. Storage Query Execution Manager (SQEM). ......................................... 53
       3.6.2. Context Service Description Module (CSDM) ........................................ 53
       3.6.3. Subscription Module (SM) ...................................................................... 54
3.6.4. Performance Data Repository (PDR) ........................................................................55
3.6.5. Efficiency and adaptability of data management in CSMS ..................................56
3.7. Storage technologies .............................................................................................................. 62
3.8. Experimental and testing environment for CSMS ............................................................ 63
4. Conclusion .................................................................................................................................. 66
5. References ................................................................................................................................... 67
List of Tables

Table 1 Constraint Operators ........................................................................................................... 21
Table 2. Summary of context representation approaches and their intersections with CoaaS platform storage requirements ............................................................................................................. 44
Table 3. Underlying storage technologies of IoT platforms .................................................................. 49

List of Figures

Figure 1. Big picture of bioTope ecosystem ........................................................................................ 15
Figure 2. CoaaS architecture ................................................................................................................. 16
Figure 3 School scenario ......................................................................................................................... 17
Figure 4 Entities relationship ................................................................................................................ 19
Figure 5 CQL basic structure ............................................................................................................... 20
Figure 6 CoaaS query engine centric framework .................................................................................. 23
Figure 7 CDQL Framework ...................................................................................................................... 24
Figure 8. CDQL Manager Package Diagram .......................................................................................... 29
Figure 9. CDQL Grammar ....................................................................................................................... 30
Figure 10. Query Execution Plan Graph ................................................................................................. 31
Figure 11. CSDL Structure .................................................................................................................... 32
Figure 12. CSD for a Carpark service .................................................................................................... 33
Figure 13. constraint checking algorithm .............................................................................................. 36
Figure 14. Node-red based example ...................................................................................................... 37
Figure 15. CDQL class diagram ............................................................................................................ 38
Figure 16- Smart Parking ......................................................................................................................... 38
Figure 17 - Visualization of situation subspace in Context Spaces theory .............................................. 44
Figure 18 - OpenIoT data platform architecture .................................................................................... 48
Figure 19 - CoaaS architecture with focus on context storage management ......................................... 51
Figure 20 - Push-based CDQL query ..................................................................................................... 51
Figure 21 - Conceptual architecture of CoaaS storage management system (CSMS) ............................ 52
Figure 22 - Context Service Description Module .................................................................................. 54
Figure 23 - Subscription module ........................................................................................................... 55
Figure 24 - Four approaches to indexing with regards to query processing [57] ................................. 60
Figure 25. CSMS implementation environment ..................................................................................... 63
## Acronyms and Definitions

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>Anti-lock Braking System</td>
</tr>
<tr>
<td>ACID</td>
<td>Atomicity, Consistency, Isolation and Durability</td>
</tr>
<tr>
<td>ACL</td>
<td>Access Control List</td>
</tr>
<tr>
<td>AES</td>
<td>Advanced Encryption Standard</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ARM</td>
<td>Advanced RISC (Reduced Instruction Set Computing) Machine</td>
</tr>
<tr>
<td>AWS</td>
<td>Amazon Web Services</td>
</tr>
<tr>
<td>BDAaaS</td>
<td>Big Data Analytics as a Service</td>
</tr>
<tr>
<td>BDaaS</td>
<td>Big Data as a Service</td>
</tr>
<tr>
<td>BDIAaaS</td>
<td>Big Data Infrastructure as a Service</td>
</tr>
<tr>
<td>BDPaaS</td>
<td>Big Data Platform as a Service</td>
</tr>
<tr>
<td>BDXaaS</td>
<td>Big Data Everything as a Service</td>
</tr>
<tr>
<td>BIBA</td>
<td>Bremer Institute for Production and Logistic</td>
</tr>
<tr>
<td>bIoTope</td>
<td>Building an IoT OPen Innovation Ecosystem for connected smart objects</td>
</tr>
<tr>
<td>BMW</td>
<td>Bayerische Motoren Werke AG</td>
</tr>
<tr>
<td>CA</td>
<td>Certificate Authority</td>
</tr>
<tr>
<td>CAGR</td>
<td>Compound Annual Growth Rate</td>
</tr>
<tr>
<td>CAP</td>
<td>Consistency / Availability / Partition tolerance</td>
</tr>
<tr>
<td>CDH</td>
<td>Cloudera Distribution including Hadoop</td>
</tr>
<tr>
<td>CEP</td>
<td>Complex Event Processing</td>
</tr>
<tr>
<td>CIRB</td>
<td>Centre d'Informatique pour la Région Bruxelloise</td>
</tr>
<tr>
<td>CoaaS</td>
<td>Context as a Service</td>
</tr>
<tr>
<td>CoAP</td>
<td>Constrained Application Protocol</td>
</tr>
<tr>
<td>CPS</td>
<td>Certificate Practice Statement (alternatively Cyber Physical System)</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CSIRO</td>
<td>Commonwealth Scientific and Industrial Research Organisation</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Definition</td>
</tr>
<tr>
<td>--------------</td>
<td>------------</td>
</tr>
<tr>
<td>CSP</td>
<td>Certification Service Provider, a CA (Certificate Authority) in eIDAS context</td>
</tr>
<tr>
<td>CST</td>
<td>Context Spaces Theory</td>
</tr>
<tr>
<td>CSV</td>
<td>Comma Separated Values</td>
</tr>
<tr>
<td>CT</td>
<td>ControlThings</td>
</tr>
<tr>
<td>DaaS</td>
<td>Data as a Service</td>
</tr>
<tr>
<td>DATEX</td>
<td>DATa EXchange</td>
</tr>
<tr>
<td>DH</td>
<td>Diffie-Hellman (key agreement algorithm)</td>
</tr>
<tr>
<td>DHT</td>
<td>Distributed Hash Table</td>
</tr>
<tr>
<td>DIALOG</td>
<td>Distributed Information Architectures for coLlaborative loGistics</td>
</tr>
<tr>
<td>DNS</td>
<td>Domain Name Server</td>
</tr>
<tr>
<td>ECC</td>
<td>Elliptic curve cryptography</td>
</tr>
<tr>
<td>eID</td>
<td>Electronic identification</td>
</tr>
<tr>
<td>EJDB</td>
<td>Embeddable JSON DataBase</td>
</tr>
<tr>
<td>eLDS</td>
<td>eccenca Linked Data Suite</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modelling Framework</td>
</tr>
<tr>
<td>Encryption</td>
<td>Reversible transformation of data by a cryptographic algorithm to produce cipher text i.e. hide the information content of the data</td>
</tr>
<tr>
<td>EPSG</td>
<td>European Petroleum Survey Group Geodesy</td>
</tr>
<tr>
<td>ETL</td>
<td>Extraction, Transform, Loading</td>
</tr>
<tr>
<td>GCHQ</td>
<td>Government Communications Headquarters (British intelligence and security organisation)</td>
</tr>
<tr>
<td>GM</td>
<td>General Manager</td>
</tr>
<tr>
<td>GML</td>
<td>Geography Markup Language</td>
</tr>
<tr>
<td>GNU</td>
<td>GNU is Not Unix</td>
</tr>
<tr>
<td>GnuPG</td>
<td>GnuPG stands for GNU Privacy Guard, which is an open and free implementation of the OpenPGP standard, defined by RFC4880</td>
</tr>
<tr>
<td>GPL</td>
<td>GNU General Public License</td>
</tr>
<tr>
<td>GSM</td>
<td>Global System for Mobile communications</td>
</tr>
<tr>
<td>GSN</td>
<td>Global Sensor Networks</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HDF</td>
<td>Hortonworks Data Flow</td>
</tr>
<tr>
<td>HDFS</td>
<td>Hadoop Distributed File Systems</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>HDP</td>
<td>Hortonworks Data Platform</td>
</tr>
<tr>
<td>HMAC</td>
<td>Hash-based message authentication code</td>
</tr>
<tr>
<td>Host</td>
<td>A computing device that mediates access to information or functionality, or provides services to a network</td>
</tr>
<tr>
<td>HTML</td>
<td>HyperText Markup Language</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>IaaS</td>
<td>Information as a Service</td>
</tr>
<tr>
<td>IBM</td>
<td>International Business Machines Corporation</td>
</tr>
<tr>
<td>ICE</td>
<td>In Case of Emergency</td>
</tr>
<tr>
<td>ICO</td>
<td>Internet-Connected Objects</td>
</tr>
<tr>
<td>ICT</td>
<td>Information and Communication Technologies</td>
</tr>
<tr>
<td>ICT30</td>
<td>Internet of Things and Platforms for Connected Smart Objects</td>
</tr>
<tr>
<td>Identity</td>
<td>A mathematically unique identity bounded to an entity or object using a private key</td>
</tr>
<tr>
<td>IoAT</td>
<td>Internet of Anything</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>IoTBnB</td>
<td>IoT service publication and Billing</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
</tr>
<tr>
<td>ISP</td>
<td>Internet Service Provider</td>
</tr>
<tr>
<td>IT</td>
<td>Information Technology</td>
</tr>
<tr>
<td>JOSE</td>
<td>JSON Web Encryption</td>
</tr>
<tr>
<td>JS</td>
<td>JavaScript</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>JSON-LD</td>
<td>JavaScript Object Notation for Linked Data</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>JWE</td>
<td>JSON Web Encryption</td>
</tr>
<tr>
<td>JWT</td>
<td>JSON Web Token</td>
</tr>
<tr>
<td>KaaS</td>
<td>Knowledge as a Service</td>
</tr>
<tr>
<td>Key</td>
<td>Sequence of symbols that controls the operation of cryptographic transformation (encryption, decryption)</td>
</tr>
<tr>
<td>KMF</td>
<td>Kevoree Modelling Framework</td>
</tr>
<tr>
<td><strong>KML</strong></td>
<td>Keyhole Markup Language</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------------</td>
</tr>
<tr>
<td><strong>LGPL</strong></td>
<td>GNU Lesser General Public License</td>
</tr>
<tr>
<td><strong>LOD</strong></td>
<td>Linked Open Data</td>
</tr>
<tr>
<td><strong>M2M</strong></td>
<td>Machine-to-Machine</td>
</tr>
<tr>
<td><strong>MAC</strong></td>
<td>Message Authentication Codes</td>
</tr>
<tr>
<td><strong>MD5</strong></td>
<td>A widely used hash algorithm</td>
</tr>
<tr>
<td><strong>MQTT</strong></td>
<td>Message Queueing Telemetry Transport</td>
</tr>
<tr>
<td><strong>MQV</strong></td>
<td>Menezes–Qu–Vanstone is an authenticated protocol for key agreement based on the Diffie–Hellman scheme with stronger notion of security</td>
</tr>
<tr>
<td><strong>MyData</strong></td>
<td>MyData is a human centric personal information management. The MyData concept lets the individual decide exactly to whom and to what extent his personal data is exposed</td>
</tr>
<tr>
<td><strong>NAS</strong></td>
<td>Network-Attached Storage</td>
</tr>
<tr>
<td><strong>NAT</strong></td>
<td>Network Address Translation</td>
</tr>
<tr>
<td><strong>Node</strong></td>
<td>A computing device which is or can be included in a network</td>
</tr>
<tr>
<td><strong>NSA</strong></td>
<td>National Security Agency (in United States)</td>
</tr>
<tr>
<td><strong>NTRU</strong></td>
<td>A public-key cryptosystem</td>
</tr>
<tr>
<td><strong>OAuth</strong></td>
<td>An open standard authentication protocol</td>
</tr>
<tr>
<td><strong>O-DF</strong></td>
<td>Open Data Format</td>
</tr>
<tr>
<td><strong>OGC</strong></td>
<td>Open Geospatial Consortium</td>
</tr>
<tr>
<td><strong>O-MI</strong></td>
<td>Open Messaging Interface</td>
</tr>
<tr>
<td><strong>Open ID</strong></td>
<td>An identity layer on top of OpenAuth v2 that verifies the identity of end-user</td>
</tr>
<tr>
<td><strong>OpenTSDB</strong></td>
<td>Open Time Series Database</td>
</tr>
<tr>
<td><strong>OWL</strong></td>
<td>Web Ontology Language</td>
</tr>
<tr>
<td><strong>PaaS</strong></td>
<td>Platform as a Service</td>
</tr>
<tr>
<td><strong>PDA</strong></td>
<td>Personal Digital Assistant</td>
</tr>
<tr>
<td><strong>PDF</strong></td>
<td>Portable Document Format</td>
</tr>
<tr>
<td><strong>Peer</strong></td>
<td>A node that can act both as a server for and as a client to other peers. This makes the nodes in a peer-to-peer network more equally ranked from the communication perspective, compared to client / server communication.</td>
</tr>
<tr>
<td><strong>PGP</strong></td>
<td>Pretty Good Privacy, a program that provides cryptographic privacy and authentication for files, emails, etc. PGP implements the OpenPGP standard.</td>
</tr>
<tr>
<td><strong>PKC</strong></td>
<td>Public Key Cryptography</td>
</tr>
<tr>
<td>Acronym</td>
<td>Definition</td>
</tr>
<tr>
<td>---------</td>
<td>------------</td>
</tr>
<tr>
<td>PKI</td>
<td>Public Key Infrastructure on which the Public Key Cryptography is deployed for use.</td>
</tr>
<tr>
<td>QES</td>
<td>Qualified Electronic Signature</td>
</tr>
<tr>
<td>QSCD</td>
<td>Qualified Signature Creation Device</td>
</tr>
<tr>
<td>QTS</td>
<td>Qualified Trust Services, a part of the eIDAS regulation</td>
</tr>
<tr>
<td>QTSP</td>
<td>Qualified Trust Service Provider, a part of the eIDAS regulation</td>
</tr>
<tr>
<td>RA</td>
<td>Registration Authority</td>
</tr>
<tr>
<td>RBAC</td>
<td>Role Based Access Control</td>
</tr>
<tr>
<td>RC4</td>
<td>Rivest Cipher 4 is a stream cipher</td>
</tr>
<tr>
<td>RDBMS</td>
<td>Relational Database Management System</td>
</tr>
<tr>
<td>RDD</td>
<td>Resilient Distributed Dataset</td>
</tr>
<tr>
<td>RDF</td>
<td>Resource Description Framework</td>
</tr>
<tr>
<td>Relation</td>
<td>A performed public key exchange between two identities. A relation has also access control definitions for the relation</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>RFID</td>
<td>Radio Frequency Identification</td>
</tr>
<tr>
<td>RIF</td>
<td>Rule Interchange Format</td>
</tr>
<tr>
<td>RPM</td>
<td>Revolutions Per Minute</td>
</tr>
<tr>
<td>RSA</td>
<td>A public key crypto system, named after the designers Ron Rivest, Adi Shamir, and Leonard Adleman</td>
</tr>
<tr>
<td>SaaS</td>
<td>Software as a Service</td>
</tr>
<tr>
<td>SHA</td>
<td>Secure Hash Algorithm</td>
</tr>
<tr>
<td>SIAMU</td>
<td>Le Service d’Incendie et d’Aide Médicale Urgente de la Région de Bruxelles-Capitale</td>
</tr>
<tr>
<td>SME</td>
<td>Small Medium Enterprise</td>
</tr>
<tr>
<td>SOA</td>
<td>Service Oriented Architecture</td>
</tr>
<tr>
<td>SOAP</td>
<td>Simple Object Access Protocol</td>
</tr>
<tr>
<td>SPARQL</td>
<td>SPARQL Protocol And RDF Query Language</td>
</tr>
<tr>
<td>SPIN</td>
<td>SPARQL Inferencing Notation</td>
</tr>
<tr>
<td>SPOF</td>
<td>Single Point of Failure</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>SSL</td>
<td>Secure Sockets Layer</td>
</tr>
<tr>
<td>SSN</td>
<td>Semantic Sensor Networks</td>
</tr>
<tr>
<td>Acronym</td>
<td>Full Form</td>
</tr>
<tr>
<td>---------</td>
<td>-----------</td>
</tr>
<tr>
<td>STIB</td>
<td>Société des Transports Intercommunautaires de Bruxelles</td>
</tr>
<tr>
<td>STO</td>
<td>Situation Theory Ontology</td>
</tr>
<tr>
<td>SWRL</td>
<td>Semantic Web Rule Language</td>
</tr>
<tr>
<td>TCP</td>
<td>Transport Control Protocol</td>
</tr>
<tr>
<td>TCP/IP</td>
<td>Transmission Control Protocol / Internet Protocol (Protocol Stack)</td>
</tr>
<tr>
<td>TLS</td>
<td>Transport Layer Security</td>
</tr>
<tr>
<td>TTP</td>
<td>Trusted Third-Party</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>UL</td>
<td>University of Luxembourg</td>
</tr>
<tr>
<td>URI</td>
<td>Uniform Resource Identifier</td>
</tr>
<tr>
<td>USD</td>
<td>US Dollar</td>
</tr>
<tr>
<td>UTC</td>
<td>Coordinated Universal Time</td>
</tr>
<tr>
<td>VP</td>
<td>Vice President</td>
</tr>
<tr>
<td>WGS</td>
<td>World Geodetic System</td>
</tr>
<tr>
<td>WP3</td>
<td>Work Package 3</td>
</tr>
<tr>
<td>WP4</td>
<td>Work Package 4</td>
</tr>
<tr>
<td>X.509</td>
<td>An important standard for a public key infrastructure (PKI)</td>
</tr>
<tr>
<td>XMI</td>
<td>XML Metadata Interchange</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
<tr>
<td>ZIA</td>
<td>Zero-Interaction Authentication</td>
</tr>
</tbody>
</table>
Executive Summary

This deliverable falls within the scope of WP4 (Context-Aware Service Provisioning for IoT), which addresses challenges of context representation, validation and reasoning about context, as well as context storage and performance. The deliverable presents the bloTope context management system called Context-as-a-Service (CoaaS) and associated software components. Key functionality of the bloTope CoaaS will be presented such as i) real-time context query engine, ii) context storage management system which is intended to address scalability and real-time data stream monitoring for situation reasoning. A prototype/demonstrator will be presented together with the report.
1. Introduction

1.1. Scope

The main purpose of this deliverable is to provide insights into today’s context- and situation-awareness and propose a solution that can enable real-time context-awareness and provide context-as-a-service (CoaaS) to any bloTope entity, tool or component. The design of CoaaS will be presented and two major components of CoaaS – context query engine and context storage management system - will be described in detail along with examples. This is the version 1 of deliverable D4.5 and reflects the current state of CoaaS development.

1.2. Audience

The target audience of the deliverable includes groups within and outside the consortium, in particular:

- **Researchers, developers and integrators of the bloTope consortium**: This deliverable illustrates important aspects of the bloTope context service formulation and delivery and will therefore serve as a valuable input for stakeholders within the bloTope consortium, notably stakeholders that work on the design of the bloTope infrastructure and/or its implementation in the scope of the bloTope open source project.

- **Researchers within other IERC and IoT EPI projects**: The deliverable illustrates some of the core implementation concepts of bloTope and will therefore be of interest to researchers in other IERC and IoT EPI projects, notably researchers working on projects that interact closely with bloTope.

- **Researchers working on IoT**: The deliverable will be also of interest to broader groups of IoT researchers, since it provides new insights into IoT open innovation ecosystem (e.g., sensors / cloud computing) integration. As a public document, the deliverable will be accessible to such groups.

- **Open source community**: In the medium term, bloTope intends to build an open source community based on the bloTope IoT platform ecosystem. This deliverable may serve as a guide to some of the introductory, yet important topics and functionalities of bloTope.

1.3. Summary

This deliverable falls within the scope of WP4 (Context-Aware Service Provisioning for IoT), which addresses challenges of context representation, validation and reasoning about context, as well as context storage and performance. The deliverable presents the bloTope context management system called Context-as-a-Service (CoaaS) and associated software components. Key functionality of the bloTope CoaaS will be presented such as i) real-time context query engine, ii) context storage management system which is intended to address scalability and real-time data stream monitoring for situation reasoning. A prototype/demonstrator will be presented together with the report.

1.4. Structure

The deliverable is structured as follows: Section 2 describes the CoaaS architectural principles focusing on proposed context query engine, EBNF structure of the context query language, examples of push/pull queries. Section 3 describes the proposed context storage management system, its architecture and an implementation strategy. Section 4 concludes the deliverable.
2. Context-as-a-Service – architectural principles

The chapter describes the proposed context query language, implementation of a context query engine and examples of applying context queries in bIoTope use cases. Further implementation of context query engine will be pursued within the bIoTope project.

The Internet of Things (IoT) is fast evolving and is expected the overall spending on IoT will reach US $1.3 trillion by 2019 from US $696 billion in 2015 [1]. As IoT grows at a staggering pace, the need for contextual intelligence will be a fundamental factor for IoT sustainability. Context-awareness enables intelligent adaptation of IoT applications such that they can perform their tasks in an efficient, proactive and autonomous manner [2]. For example, consider a smart home scenario where a smart tumble dryer is tasked to dry a piece of clothing tagged with information (e.g. using RFID) regarding drying instructions. Assuming the clothing material is not suitable for tumble drying, without context-awareness, the smart dryer will dry the cloth unaware of this fact. Augmenting the IoT devices and application with context will enable it (the smart dryer) to reason over the data (the cloth) and arrive at the right decision, in this case, not to tumble dry the piece of cloth.

The “big picture” of the bIoTope ecosystem is illustrated in Figure 1. Big picture of bIoTope ecosystem incorporating core components and O-MI/O-DF wrapped partner platforms. Platforms P1, P2 and so on represent partner platforms and technologies (e.g., Warp10) which are wrapped into O-MI/O-DF compliant wrappers. O-MI/O-DF is the “glue” that binds the ecosystem together. Any new tool or platform can join the bIoTope ecosystem once it has an O-MI/O-DF compliant wrapper that can be developed manually or possibly in a semi-automated way as illustrated by P5 in the figure. The core components will be developed by the bIoTope consortium and are represented in the belt that encompasses the component platforms. The ecosystem will have multiple entry points and the service requests (SR) are represented by multi-section documents possibly incorporating SPARQL queries, O-MI/O-DF XML, web-services, SSNO (SSN Ontology) references and resources URIs. SRs can be generated by users using IDE interfaces or via APIs from applications, services and other IoT EPI ecosystems. CoaaS component consists of context query engine, context storage management system and context reasoning engine.

Nowadays, sources of context are not limited to external sensors and built-in sensors in smart devices such as accelerometers, GPS sensors, and light level sensors. Web APIs (e.g. linked open data [3]) and social networks also provide rich, useful and relevant information to deduce context. On the other hand, due to the proliferation of smart devices and popularity of other context’s sources such as social networks, the same contextual information might be offered by several context sources. Therefore, to use the maximum capacity of IoT applications (augmenting them with c-awareness), it is essential to provide an easy and standard approach to define, advertise, discover/acquire, and query context. A promising solution to address this need is encapsulating context related information as real-world services to cooperate them with other entities.

A similar concept was raised and studied in the realm of Semantic Web Services (SWS) [4] to add automation and dynamics to traditional web services. SWS aims at providing formal descriptions of requests and web services that can be exploited to automate several tasks in the web services usage process, including dynamic discovery of services [5]. However, SWS is not suitable for describing context relevant to IoT applications mainly due to the dynamic nature of IoT (i.e. unstable connections, mobility, and heterogeneity). SWS focuses on describing the service, its characteristics, and the service instantiations. However, in the case of context related services for IoT applications, besides the need to describe the service, its instantiations, and the characteristics of the service; it is also im-
important to describe the characteristics of service provider and the IoT entities (e.g. devices) producing the data, and the relationships between the entities (including people, devices, infrastructure etc.). Further, context can have different levels of abstraction. Context can be low level information such as a Celsius temperature value of 35 or high level context inferred from low level context such as ‘a fire threat’. The limitations of SWS makes its hard to represent different aspects of context and support the needs of IoT applications. Hence, a standard framework that supports various abstractions of context while recognizing the highly dynamic environment of IoT and its characteristics is needed. We refer to this framework as Context as a Service (CoaaS) (Error! Reference source not found.).

![Figure 1. Big picture of bIoTope ecosystem incorporating core components and O-MI/O-DF wrapped partner platforms](image)

**Error! Reference source not found.** illustrates the overview of CoaaS framework and shows its main components. CoaaS is a context management middleware which is responsible for providing a comprehensive method to allow a smart entity, namely context service consumers, to consume a context service provided by another entity that is a context service provider. The entity is a member of the IoT ecosystem and can include physical IoT devices, human beings, internet connected vehicles, and mobile smart phones to name a few. The CoaaS enables global standardization and interworking among different context providers and consumers in IoT environments. Context as a Service (CoaaS) forms an important part of the service offerings in the EU Horizon-2020 project bIoTope ([www.biotope-h2020.eu](http://www.biotope-h2020.eu)) – Building IoT OPen Innovation Ecosystem for connected smart objects.

CoaaS needs to address several challenges, such as modelling and representing context providers, expressing context requests, storing and indexing contextual information, describing context services, semantic service discovery, and dynamic selection of context services.
In this chapter we focus on one of the fundamental challenges in designing the CoaaS framework, namely, the need for a generic approach to define, represent, and query context in IoT applications. To achieve this goal, we propose a Context Definition and Query Language (CDQL). CDQL consists of two components, Context Definition Model (CDM) which is responsible for modelling and representing context, and CQL which is used to query and share contextual information.

### 2.1. A motivating scenario

In this section, we present a motivating scenario that explains the need for a flexible, dynamic, easy to use, and scalable representation and query language for context-aware applications. The scenario under consideration is about safety in smart cities, in particular school children safety. As depicted in Figure 3, the school safety scenario consists of several entities such as a smart bus, a smart car, mobile devices, a school server, and a smart gate (we have selected a few entities to exemplify a smart city use case. However, there is no limit to the number of entities). As a first step, it is essential to have a generic context model to acquire and represent the contextual information from IoT entities.

In the example depicted in Figure 3, a user called John wants to pick-up his daughter, Hannah, from her school. On his way to school, due to an unexpected traffic, he realizes that he cannot arrive at the school on time. Realizing this, a smart IoT system begins to determine alternatives to achieve the goal “pickup Hannah”. An option could be to request another trusted parent to pick-up Hannah from school on John’s behalf. In order to represent this context request, several factors should be considered, namely:

- The selected parent(s) for picking up Hannah should be trusted by John;
- The selected parent(s) should have a car with an extra seat for Hanna;
- The selected parent(s) should be close enough to the school;
- The child of the selected parent(s) should finish the school in the same time as Hannah;
- The child of the selected parent(s) should be currently at school.
Additionally, this process needs to be automated so John’s device can automatically trigger the same query, “pickup Hannah” whenever he is running late. Now consider Alice who is Bob’s mother. She is the parent matching all the above-mentioned criteria to pick up Hannah. She wants to know the fastest and safest location for picking up Bob and Hannah from school.

Based on this smart city IoT application scenario and the aforementioned considerations, we list the key requirements in designing CDQL as the following:

- To support complex context queries concerning various contexts entities and constraints;
- To provide transparency. In another word, it should hide technical details of context providers such as storage techniques;
- To conform to a context model that can be converted into different data models as required.
- To support both pull-based and push-based queries;
- To support continuous queries;
- To support aggregating and reasoning functions to query both low level and infer high-level context.

![School scenario](image)

**Figure 3 School scenario**

## 2.2. CDQL

This section describes our proposed Context Definition and Query Language (CDQL) to compose and answer context queries. As we mentioned, CDQL consists of two components: Context Definition Model (CDM) and Context Query Language (CQL). In the rest of this sub-section a detail explanation of our proposed CDM and CQL is presented.

### 2.2.1. CDM

One of the main motivation behind designing the CDQL is to enable IoT entities (e.g. machines and smart devices) to request, share and exchange context with each other. In other words, CDQL is designed to make it possible for an entity to retrieve a specific set of contextual information from an-
other entity(s) with particular characteristics. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves [6].

An IoT application could be considered as a collection of entities to serve a particular purpose. The context about an entity is generated/captured from the IoT data produced by the entity (context provider) and is made available/shared with other entities (context consumer). For example, consider an entity A that wants to know about the level of light at night in a certain bike path. To answer this query, we need to first find those entities (e.g. humans carrying mobile devices, fixed sensors etc. that are part of an environmental monitoring IoT applications) located in that area. Then, we need to filter the retrieved list based on the entity’s context e.g. in case of a smart phone, its owner activity (context) to determine relevance, the smart device capabilities such as equipped with a light sensor etc. In order to support such queries, there is a need to define different types of contextual information and the corresponding characteristics and capabilities using a generic and standard approach. Based on these considerations, we list four key requirements in designing CDM as the following:

- The CDM should define the context related capabilities of each device (mainly sensing, processing, and reasoning capabilities) and how they can be invoked;
- The CDM should define and capture the main characteristics of each device;
- The CDM should define the context entities (described later on in this section) and their relationship;
- The CDM should be generic and domain independent.

As illustrated in Figure 4, to capture all the required contextual information, in this research, we define five fundamental context entity types namely: Person, Organization, Location, Event, and Device. These types are designed as the basis of CDM for defining a common language between context providers and consumers to make the communication possible. However, since we considered objected-oriented concepts during the design of CDM, the aforementioned types can get extended to sub-types (based on the application requirements) for capturing and storing more sophisticated contextual information.

Person Entity: The person entity consists of a set of contextual information about a person such as name, age, gender, DOB, contact details, home address and so on.

Organization Entity: This entity defines the characteristics of an organization and consists of three components: organization type, general profile, and domain specific profile. The organization type provides information about the type of the organization. Some possible values for this attribute are educational organization, government organization, sporting organization, and local organization.

The general profile includes information about an organization which is independent of the organization type such as name and contact information. And the last component is the domain specific profile which provides more contextual information based on the type of the organization.

Location Entity: Location is a primary context. The location entity describes location details based on two aspects: Geographical Location and Semantic Location. The Geographical Location is used to provide information about the actual location of an entity. This information might be represented as latitude and longitude or Street Address. On the other hand, the semantic location focuses on the semantics of a location such as at home, or very close to school.

Event Entity: This entity is responsible for representing the activities that a person or organization are involved in. Two main information pieces that are represented in this entity include the event type and the event time. The event type defines the type of the event (e.g. business event, educational event, music event, sporting event, or social event) and the event time provides information
about the start time, the start date, and the event duration. Furthermore, for the recurring events, it also provides information to show how often the event will be repeated, and when the repeating event will be ended.

Device Entity: Device entity describes the characteristics of a device. As it is depicted in Figure 4, this model classifies devices into two types, computable and non-computable profiles. The non-computable profile defines the general characteristics of a device. For example, for a vehicle it can consist of the vehicle type, number of doors, and the number of seats. On the other hand, computable profile describes the specification and capabilities of each device. We classify the computable profile into three sub-classes:

- Software profile determines the available software on the device and their context related capabilities;
- Hardware profile consists of sensor profile and general hardware profile.

We represent the relationships between these entities as $E_1 R t E_2$ where $E_1, E_2 \in \{Person, Organization, Device, Location, Event\}$, $R \in \{Know, Own, MemberOf, LocatedIn, PartOf, InvolvedIn\}$ and $t$ is an optional attribute that denotes the type of the relationship.

2.2.2. CQL

To fulfil all the discussed requirements for querying and sharing context between entities in an IoT environment, we propose a novel query language called CQL. As we mentioned, one of the requirements of CQL is to support both pull-based and push-based queries (Requirement 3). The pull-based queries are used for synchronous queries to obtain contextual information using a pull-based mechanism. However, since there is a common need in context-aware applications to discover situations’ changes and adjust to them automatically, there can be a number of queries about the changes in context. These context changes should be detected and pushed to applications as notifications. Therefore, to address this need, we design push construct which support push-based queries.

$$CQL = PULL | (PUSH WHEN) WHERE;$$
Figure 5 visualizes the above production rule and highlights the core elements of CQL, namely Pull, Push, When, and Where. In the rest of this section, the details of each of these elements will be discussed.

The pull clause determines the query response structure. Like traditional queries in relational databases, each context query can return a set of values as query result. In CQL, we define an element called CONTEXT-ATTRIBUTE to represent context related attributes. This element consists of two parts: CONTEXT-ENTITY and IDENTIFIER. The IDENTIFIER determines the type of context we are interested in. For example, it can be location, temperature, or any other type. The CONTEXT-ENTITY defines the entity which the context attributes need to be queried from. The value for this attribute can be any of the entities known to the IoT ecosystem. We provide a mechanism to define such entities using a define clause which is explained later in this sub-section (Requirement 1).

To support different levels of context, a context query language needs to also represent high level contextual information (Requirement 5). In this regard, we introduce an element FUNCTION-CALL. The FUNCTION-CALL has three components: FUNCTION-NAME, AS, and OPERANDS. The AS is an optional element and designed to assign a name to the generated value. On the other hand, FUNCTION-NAME is a mandatory module and determines the context function that needs to be applied to a set of context attributes or context entities.

As we mentioned, it is a common need for context-aware applications to discover changes of situations and react to them in an automatic manner. Therefore, we introduce push clause to address this requirement. By having this feature, different entities or applications can subscribe to be notified when a specific condition or situation is detected. Push clause consists of two main parts, SELECT and CONTEXT-REQUESTER, where the CONTEXT-REQUESTER refer to the ID of a class of entities or application (e.g. all traffic lights in a location) that register to get notified and SELECT identifies the information that needs to be pushed to the subscriber.

The Third element of CQL is WHEN. We introduce this element to support periodic (e.g. check the temperature of a room every 10 minutes) and continuous (e.g. tracking location of a person) context queries. This element is also responsible to define when a context query needs to be triggered (in push-based queries). This is important in a number of context-aware IoT applications that perform continuous monitoring or need to be notified when a specific situation is detected.

The WHEN clause consists of three parts, CONDITION, INTERVAL and OCCURRENCE. INTERVAL shows the sampling interval for a context query. In order to represent continuous queries, the value of this attribute should be set to 0. On the other hand, the UNTIL attribute indicates the timespan of the context retrieval. The other element of when clause is CONDITION. This element characterizes the conditions that need to be detected in order to trigger a context query. CONDITION is designed to support compound predicates that consists of several constraints connected by logical operators (AND/OR).
We introduce a new element to provide a guideline on how to define constraints inside each condition. This element, which is named CONSTRAINT, consists of OP and OPERANDS. OP stands for the operator and indicates which logical operator should be applied to the operands. The list of constraint operators is shown in Table 1 Constraint Operators.

<table>
<thead>
<tr>
<th>OP</th>
<th>OPERANDS</th>
<th>Meanings</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>Equal</td>
<td>LTE</td>
</tr>
<tr>
<td>GT</td>
<td>Greater than</td>
<td>CHANGED</td>
</tr>
<tr>
<td>GTE</td>
<td>Greater or equal</td>
<td>CONT</td>
</tr>
<tr>
<td>LT</td>
<td>Less than</td>
<td>NEG</td>
</tr>
</tbody>
</table>

In CQL, each operand can be a contextFunction, a contextAttribute, CONTEXT-ENTITY, or a simple value represented in string or number.

The last core element of our proposed CQL is the DEFINE clause. This element is considered as the heart of CQL and consists of two main parts: CONTEXT-ENTITY and CONTEXT-FUNCTION. CONTEXT-ENTITY identifies the entities that are involved in a context query since CQL is capable of querying contextual information from multiple context entities (Requirement 1). On the other hand, the aim of CONTEXT-FUNCTION is to add the capability of supporting custom context functions to the CQL (Requirement 5).

The CONTEXT-ENTITY indicates the entities that are involved in a context query and consists of three parts, an IDENTIFIER that assigns a name to the entity, which will be used when it is needed to refer to the entity (e.g. to pull an attribute of a specific context entity), ENTITY-TYPE that defines the type of entity (e.g. Person, Device, or Event), and CONDITION which provides a guideline on how to filter out unwanted context entities from a large number of available entities. As we discussed earlier in this section, each condition is an AND or OR clause over a set of constraints. These constraints define characteristics of the entity of interest. The constraints can be applied either to low-level context (CONTEXT-ATTRIBUTE) or high-level context (CONTEXT-FUNCTION).

In CQL, the context functions can be categorized into three subclasses: aggregation functions, relational functions, and situational functions.

Aggregation functions refer to those context functions that apply aggregation operations to the provided context attributes to return a single value. CQL provides a number of pre-defined aggregation functions, namely: MAX, MIN, SUM, AVG, and COUNT. However, there might be a need for more sophisticated aggregation functions when dealing with a more complex query. For instance, a query might need to calculate the distance between two locations. In this regard, there should be a distance function that accepts two locations (i.e. latitude, longitude) and calculates the distance between them. As aggregation functions are usually application dependent, pre-defining a comprehensive list of them is not possible. Instead, we design CQL in a way that aggregation functions can be defined dynamically as a part of query. In CQL, we decide to provide aggregation functions through a restful API. Two main reasons that lead us to provide this kind of context function through an API call rather than defining it inside the queries are to increase the reusability and ease of development as it is hard to represent complex aggregation functions as a part of query. Therefore, AGGREGATION-FUNCTION has a mandatory attribute that denotes the restful method URI. It has one to many IDENTIFIERS where they identify the parameters of the restful method. Incorporating restful APIs into aggregation functions enables users to develop their custom-defined functions as restful web services and use it in the CQL.
The RELATIONAL-FUNCTION is the second subclass of context functions which is designed to determine whether there is a relationship between two or more context entities. isInvolvedIn(p1,o1,e1) is an example for a relational function where p1 is a person, o1 is an organization, and e1 is an event. This function determines whether both p1 and a1 are involved in the event e1 or not. To represent this type of context function, we consider three components, ID, RELATION-TYPE, and ENTITY-TYPE; where ID assigns a unique IDENTIFIER to each function, RELATION-TYPE identifies the type of the relationship, and ENTITY-TYPE indicates the type of entities that the existence of the specified relationship needs to be checked on them.

The last category of context functions includes SITUATIONAL-FUNCTIONS. It is a common need in many context-aware IoT applications to query about the situation of a context entity. Situations are high level contextual information that are inferred from multiple low level context [7]. For example, roomSituation(location:room) will determine whether there is a meeting, presentation, or party in a specific room. Same as aggregation functions, situational functions are application dependent and it is not practical to predefine them. Therefore, we extend CQL to support dynamic definition of situational functions as a part of the query. We will discuss the user defined context functions in more detail when we introduce the define element. Therefore, we introduce situationalFunction for representing situations in CQL. To achieve this goal, we adopt the model introduced in [8] which defines a situation in pervasive environments as a collection of regions in multidimensional spaces. Based on [8], to represent the situation of a context entity, three elements should be defined:

- Possible situations for a context entity;
- Context attributes that are involved in the reasoning;
- A collection of accepted contextual values (i.e. regions) for each situation.

To clarify this concept, consider we want to represent the possible situations of a smart room, which provides a place for meetings, presentations, and public gatherings and is equipped with sensing devices. Each of these situations can be characterized using three context attributes, namely, the room light level, the number of people in the room and the room noise level. Then the acceptable regions of values for each context attribute should be defined e.g. the lower and upper bounds of light and noise levels. To represent this information in a query, we introduce SITUATIONAL-FUNCTION. SITUATIONAL-FUNCTION consists of an IDENTIFIER which define the function ID, an ENTITY-TYPE which indicate type of the corresponding entity, and one to many SITUATION(s). Each SITUATION has an IDENTIFIER as well, which represents the title of situation (e.g. meeting). It also has one to many CONTEXT-ATTRIBUTE. To represent the acceptable range of values for each attribute, we define a new element called RANGE. In order to express the grammar of CQL, we used Extended Backus–Naur Form [9](EBNF).

### 2.3. CoaaS and CDQL Framework

This section proposes a framework that utilized the proposed Context Definition and Query Language to advertise, discover and invoke context services with minimal user involvement. However, to provide a better understanding, we start this sub-section by providing a brief description of the Context as a Service (CoaaS) middleware.

CoaaS is responsible for providing a comprehensive method to allow a smart entity, namely, context service consumers, to request & receive a context service provided by another entity that is a context service provider. Figure 6 depicts CoaaS framework focusing on context query engine part, consists of seven modules, namely Security, Analytics, Reasoning, Storage, System Monitoring Manager, and Context Validator. A brief description of each of these components is presented in the following list.
1) **Security**: All incoming messages (e.g. subscriptions, context request, context update) go through the security module. One of the responsibilities of the security module is authentication (by checking the validity of the token). On the other hand, each context service might be accessible only for a specific set of CCs. Therefore, the security module is responsible to check if the context consumer has access to the requested context service or not. Furthermore, it also responsible for monitoring all the incoming messages to identify any suspicious patterns. For example, to stop distributed denial-of-service (DDoS) attack.

2) **Context Storage and Caching**: In CoaaS, different type of contexts (Cached contexts, CSD, Subscription repository, entity repository) are needed to be stored and queried. Therefore, it is essential to have a proper storage mechanism to enhance the performance of CoaaS. On top of that, Context Storage is also responsible for caching the received contextual information for further performance improvement. Each context provider should register its Context Service Description (CSD) in the CoaaS.

3) **Context Reasoning** is responsible to infer context from raw sensor data or from existing primitive low-level context. The middleware may apply feature extraction, description logic, rule-based reasoning or probabilistic inference on behalf of the application layer.

4) **Analytics** is used to produced meta-data based on the historical context to facilities the procedure of context service selection. Furthermore, this component is the enabling factor for context validation.

5) **Context validator** is designed to deal with context imperfectness. Contextual information might be incorrect (reflecting the wrong state of the world), inconsistent (containing contradictory information), or incomplete (missing some aspects of the context) [10]. Therefore, all the incoming contextual information (coming from external CPs) should be validated before getting used.

6) **System Monitoring Manager** monitors running CoaaS components, collects statistics about queries, context access, subscriptions and other log data in order to analyse, perform machine learning and identify possible improvements in overall efficiency of CoaaS.

7) **CDQL Manager** is considered as the heart of CoaaS. This module receives the context queries form CCs and provides an appropriate answer for them by utilizing the other six components. Since designing CDQL Manager is one of the main focuses of this research, we will explain this module in more depth in the rest of this sub-section.
Now that we defined the main functionalities of each component of CoaaS, we will take a closer at our proposed architecture for CDQL and its workflow for answering context queries. CDQL Manager consists of five major components. These components are Communication Manager, Context Query Engine, Context Service Discovery and Selection, Query Coordinator, and Context Service Invoker. To increase the readability of CDQL architecture framework we used color-coding. As it is shown in Figure 7, we used a unique colour to represent each of the seven modules of CoaaS. These colours are used to show the relationship between CoaaS modules and CDQL components. For example, Context Query Engine’s yellow colour implies that this component might uses Reasoning module.

1) **Communication Manager** is responsible to handle all the in-coming and out-coming messages, namely context requests, context updates, and context responses. It will receive the incoming messages from CPs and CCs and pass them to the corresponding module. To guarantee the privacy and security of the CoaaS, this component is linked to the security module.

2) **Context Query Engine** (CQE) has two main responsibilities: parsing the incoming queries, and producing the final query result. This component will receive the incoming queries and break them into several sub-queries. Furthermore, it is also responsible for aggregating the context responses (using the reasoning module) to produce the final query answer.

3) **Context Service Discovery and Selection** is responsible for find the most appropriate CP for an incoming request. This module consists of two sub-components:
a. **Context Service Discovery** is responsible for finding those context services that match the requirements of the context request. It will pass the candidates services to the Service Selector.

b. **Service Selector** is responsible for selecting the best available context service that can satisfy requirements of the context request form the candidates list. Due to the proliferation of smart devices and popularity of other context's sources, the same contextual information might be offered by several context sources. However, context services can differ in their quality levels (e.g. accuracy) and representations (e.g. position represented in coordinates and as an address) of the offered information, and costs (e.g. battery consumption) for providing the information.

4) **Query coordinator** is responsible for handling push-based (event based) and continues queries. Regarding the continues query, the query coordinator will trigger the registered query on the specified intervals. And in the case of event-based queries, the query coordinator will listen to the incoming contextual information form entities (it can be high-level or low level context) and trigger a query if the incoming information match the conditions of the registered query.

5) **Context Service Invoker**: The context service invoker is responsible for invoking the context services from the corresponding context providers to retrieve the required contextual information and pass the retrieved information to the CQE.

As we stated, context queries can be classified into two sub-classes, pull-based and push-based queries. The pull-based queries are used for synchronous queries to obtain contextual information using a pull-based mechanism. However, since there is a common need in context-aware applications to discover situations’ changes and adjust to them automatically, there can be several queries about the changes in context. These context changes should be detected and pushed to applications as notifications. Push-based queries are responsible to address this need. In the rest of this section, we will explain the workflow of CDQL for both pull-based and push-based queries.

<table>
<thead>
<tr>
<th>Pull-based query workflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Context consumer issues a query.</td>
</tr>
<tr>
<td>2. Query Engine parses the query, and breaks it down into sub-queries</td>
</tr>
<tr>
<td>3. Context Service Discovery and selection:</td>
</tr>
<tr>
<td>• Receives the sub-queries</td>
</tr>
<tr>
<td>• Finds the context services (providers) that can answer each sub-query by looking up in the CSD repository.</td>
</tr>
<tr>
<td>• If more than one service is found for each sub-query, selects the best service based on QoC, CoC, ... (Analytics services can be used here)</td>
</tr>
<tr>
<td>4. If CDQL works as a redirector, send the selected services back to the context consumer (End.); Else, send the selected services to the Context Service Invoker.</td>
</tr>
<tr>
<td>5. Context Service Invoker sends the context queries (requests) to the context provider(s)</td>
</tr>
<tr>
<td>• Note that before passing the context requests to the Context Service Invoker, the context consumer is authenticated.</td>
</tr>
<tr>
<td>• Note that context provider might be the caching server</td>
</tr>
<tr>
<td>6. Context provider(s) send the requested contexts back to the context Service Invoker</td>
</tr>
<tr>
<td>• Note contexts can be cached by caching server</td>
</tr>
<tr>
<td>7. Send the contexts to the Context Query Engine</td>
</tr>
<tr>
<td>8. Context query engine calculates the result and sends it to the context consumer</td>
</tr>
<tr>
<td>• Note that reasoning module (or aggregation) might be used to infer high level context</td>
</tr>
</tbody>
</table>
Push based queries workflow

The workflow of push-based queries is twofold, one: Subscribing to the CoaaS platform, and two triggering query. The explanation of both parts is provided as follows:

Part 1: Subscription
1. Context consumer issues a query.
2. Query Engine parses the query, and breaks it down into the sub-queries
3. Context Service Discovery and selection:
   - Receives the sub-queries
   - Finds the context services (providers) that can answer each sub-query by looking up in the CSD repository.
   - If more than one service is found for each sub-query, selects the best service based on QoS, CoS, ... (Analytics services can be used here)
4. Send the selected services to the query scheduler and register them in the subscription repository.
5. If applicable, register the situations of interest in the context providers/consumer to be notified when the situation is detected.

Part 2: trigger
1. Detected situations are sent to the query scheduler
2. Query scheduler looks up in the subscription repository
   - Reasoning module might be used
3. Corresponding query will be sent to the query engine
4. Go to 2 in pull-based queries

2.4. CQL EBNF

This section represents the CQL in EBNF form.

\[
\text{CQL} = \text{PULL} \mid (\text{PUSH WHEN}) \text{ WHERE};
\]

\[
\text{PULL} = \text{'Pull'} \text{ SELECT};
\]

\[
\text{SELECT} = \text{CONTEXT-ATTRIBUTE} \mid \text{FUNCTION-CALL} \{',', (\text{CONTEXT-ATTRIBUTE} \mid \text{FUNCTION-CALL})\}
\]

\[
\text{CONTEXT-ATTRIBUTE} =
\text{CONTEXT-ENTITY'.' IDENTIFIER;}
\]

\[
\text{CONTEXT-ENTITY} =
\text{IDENTIFIER; refer to an Entity which will be defined in the where clause}
\]

\[
\text{FUNCTION-CALL} =
\text{FUNCTION-NAME '(' ('OPERAND (', OPERAND)' )' 'as' IDENTIFIER; OPERAND = CONTEXT-ENTITY | CONTEXT-ATTRIBUTE | FUNCTION-CALL; CONTEXT-VALUE = string | number;}
\]
FUNCTION-NAME =
IDENTIFIER; refer to a function which will be defined in the where clause
IDENTIFIER =
letter, {letter | digit | "_"} ;
PUSH = ‘Push’ SELECT ‘into’ CONTEXT-REQUESTER;
CONTEXT-REQUESTER = IDENTIFIER; refer to the ID of the subscriber entity/application
WHEN = ‘When’ CONDITION [‘EVERY’ INTERVAL] [‘Until’ OCCURRENCE];
INTERVAL = number; in MS
OCCURRENCE = date | number ‘of occurrence’ | lifetime;
CONDITION = CONSTRAINT |
CONDITION AND CONDITION |
CONDITION OR CONDITION |
NOT CONDITION |
'(' CONDITION')';
CONSTRAINT = OPERAND ['NOT'] OP [OPERAND]
OP = EQ | GT | LT | GTE | …;
DEFINE = ‘define Entities as’ CONTEXT-ENTITY {, CONTEXT-ENTITY} [ ‘and Functions as’ CONTEXT-FUNCTION {, CONTEXT-FUNCTION}]
CONTEXT-ENTITY = ‘Entity’ IDENTIFIER ‘is from’ ENTITY-TYPE [‘where’ CONDITION];
ENTITY-TYPE = Person | Organization | Location | Event | Device;
CONTEXT-FUNCTION = AGGREGATION-FUNCTION| RELATIONAL-FUNCTION | SITUATIONAL-FUNCTION;
AGGREGATION-FUNCTION = ‘aFunction’ IDENTIFIER ‘is from’ URI ‘{‘ IDENTIFIER {‘,’ IDENTIFIER } ‘}’;
RELATIONAL-FUNCTION = ‘rFunction’ IDENTIFIER ‘is’ RELATION-TYPE ‘{‘ ENTITY_TYPE , ENTITY_TYPE {‘,’ ENTITY_TYPE } ‘}’;
RELATION-TYPE = Know | OWN | memberOf | locatedIn | partOf | invovedIn;
SITUATIONAL-FUNCTION = ‘sFunction’ IDENTIFIER ‘is on’ ENTITY-TYPE ‘situations’ SITUATION {‘,’ SITUATION };
SITUATION = ‘situation’ IDENTIFIER ‘{‘
CONTEXT-ATTRIBUTE ‘:’ RANGE
{‘,’ CONTEXT-ATTRIBUTE ‘:’ RANGE})’;
RANGE = (‘{‘ number ‘…’ number ‘}’ ) |
‘{‘ CONTEXT-VALUE {, CONTEXT-VALUE } ‘}’;
2.5. Development and execution environment for CDQL

In the previous sections, the CDQL manager was presented. In this section, the prototype implementation of the proposed CDQL manager will be described. The implementation follows the loosely-coupled, functionally driven component development principle. The developed prototype consists of several loosely-coupled components where each of them is responsible for a specific functionality.

The remainder of this chapter is organized as follows. In section 2.5.1, we discuss the tools and development environments involved to building the prototype. Section 2.5.2 describes the overall architecture of the implemented prototype of CDQL manager and explain its main components. Sections 2.5.3 is devoted to the Node-red integration of the prototype.

2.5.1. Development Environments and Tools

The prototype of the proposed CDQL manager implemented in Enterprise Java application using Java using Java EE technologies including Enterprise JavaBeans (EJBs) and Java Persistence API (JPA). The rest of this section presents the description of the development environments and tools that are used for developing and implementing the proposed CoaaS architecture.

**Development Platform & Tools**
- Application Platform: J2EE 7
- Programming language: Java with JDK 18
- IDE: NetBeans IDE 8.2
- Server: Glassfish Server 4.1
- Technologies: EJBs, Jersey RESTful Web Services framework

**Evaluation Devices**
The application was deployed and tested in a MacBook Pro (Retina, 15-inch, Mid 2015) laptop with the following technical specifications:
- **Processor**: 2.2 GHz Intel Core i7
- **Memory**: 16 GB 1600 MHz DDR3
- **OS**: OS X El Captain Version 10.11.6

**Third party libraries**
During the development of the CoaaS, a set of third party libraries have been employed to enhance the functionalities of the application. These libraries are as follows:
- **Antlr 4.6**
- **Json-ld 0.10.0**

2.5.2. Architecture of the Proof-of-Concept Implementation

As we presented in the previous sections, the CDQL Manager consists of five major components. These components are Communication Manager, Context Query Engine (CQE), Context Service Discovery and Selection (CSDS), Context Service Registration (CSR) Coordinator, and Context Service Invoker. The following package diagram (Figure 8) shows the structure of the implemented prototype.
In the rest of this section, we will briefly explain some of the main components of the implemented prototype.

**Context query engine**

The context query engine has two main responsibilities: parsing the incoming queries, and producing the final query result. This component will receive the incoming queries and break them into several sub-queries. Furthermore, it is also responsible for aggregating the context responses (using the reasoning module) to produce the final query answer. CQE consists of two parts, context query parser and context query planner. In the rest of this sub-section we will briefly explain these components.

**Context query parser**

To parse the incoming queries, a query parser is developed by using Antlr 4.6. ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. This framework accepts a formal grammar (written in a EBNF like format) as input and generates a parser for that language that can automatically build parse trees, which are data structures representing how a grammar matches the input. ANTLR also automatically generates tree walkers that you can use to visit the nodes of those trees to execute application-specific code. The CDQL grammar written is provided in the following code-snippet (Figure 9).
As it is stated before, CDQL supports complex context queries concerning various context entities where the information about each context entity might be provided by a different context provider. In another word, each context query can request contextual information related to one to many context entities where each of them will be later converted to a separate context request. Further, the context entities used in a query are most possibly dependent, which means information retrieving from one context entity might be used in definition of another context entity (like nested selects in normal SQL).

CDQL queries can be represented a unidirectional graph where each node represents one context entity and each edge between two nodes represents the relationship among those context entities. In order to calculate the execution plan, it is essential to find a path that visit all the nodes in the graph starting from a node with 0 inbound degree. To clarify, consider the following context query written in CDQL to find alternative parents who can pick-up Hannah from school when his father John is unable to arrive at Hannah’s school on time.
This query consists of five entities: Hannah, HannahSchool, HannahSchoolStudents, carWithSeat, and Parent. Figure 10 shows the directed graph generated based on this query.

```
PUSH 'pickup-request' to parents
When John.ArrivalTime GT HannahSchool.EndTime until lifetime Define Entities as
Entity Hannah is Person where parenthood(Hannah,John) EQ true,
Entity HannahSchool is Organization where School-Student (Hannah, HannahSchool) EQ true,
Entity HannahSchoolStudents is person where School-Student (HannahSchoolStudents, HannahSchool) EQ true,
Entity carWithSeat is Car where carWithSeat.emptySeats>0,
Entity Parent is person where Ownership(carWithSeat, parent) EQ true AND Parenthood(Parent,HannahSchoolStudents) EQ true AND Distance (parent.location, HannahSchool.location) lt 500
AND Functions as r
Function Parenthood is Know:parent (Person, Person) ,
rFunction Ownership is Own (Person, Device) ,
rFunction School-Student is MemberOf:student (Person, Organization);
```

As it is depicted in the graph (Figure 10), the inbound degree of entities carWithSeat and Hannah are 0. Therefore, these two entities can be retrieved simultaneously in the first step. In the next step, when the required information regarding Hannah is fetched, the context request related to Hannah school can be issued. In the same manner, in the next step, the request for HannahSchoolStudents can be executed. Lastly, when the required contextual information related to carWithSeat and HannahSchoolStudents fetched, a context request will be generated to find the eligible parents to pick-up Hannah from school on behalf of John. Therefore, the execution plan for this query can be written as below:

1. Hannah, and carWithSeat
2. HannahSchool
3. HannahSchoolStudents
4. Parent
Context Service Description model

In this section, we present our proposed Context Service Description Language (CSDL). CSDL is an abstract service model meeting the requirements of context-aware systems, like the one that is required in IoT. CSDL is a markup language for services that supports context descriptions and capable of defining services in terms of their semantic signature, and context-aware behavioral specification.

The context service description comprises three main parts (Figure 11): Context Service Profile, which is for advertising and discovering service capabilities; Context Service Model, which gives a detailed description of service signature; and Service Grounding, which provides details on how to interoperate with a service, via messages. The brief description of each of these three components is provided in the rest of this subsection.

- **Context Service Model** gives a detailed description of service signature and also identify the semantic vocabularies which is supported in the given queries.
- **Service Grounding** provides details on how to interoperate with a service. This component identifies which type of communication needs to be used in order to call the service (e.g. HTTP get, XMPP, TCP/IP). Further, based on the type of the communication, it will provide other required information to make the service invocation possible (e.g. URI in the case of HTTP get).
- **Context Service Profile** is used to make the service advertising and discovering possible. This component indicates the type of the entity that a context service can provide information about. Further, by using the power of CDQL language, it defines the context-aware behavior of the service.

In the current implementation, we used JSON-LD to represent context service description. The following text-box shows an example of CSD that written to register a context service that provides information about available car parks in a specific location (Figure 12).

Figure 11. CSDL Structure
Figure 12. CSD for a Carpark service
Context Service Discovery and Selection

In this section, we will describe the different levels of the context service matching process: first detailing the different distance functions used for computing the similarity between two context attributes with the same scope, secondly describing preliminary service matching, and finally explaining the specification-based mechanism.

Distance functions & Similarity functions

In order to compute the distance between two context attributes \((k_1, k_2)\) with same type \((k_1\.type = k_2\.type = \text{Temperature})\), we defined four different distance functions explained in the rest of this sub-section.

Boolean Distance: this function will compare two context values and return a Boolean value (either 0 or 1). The similarity is computed as:

\[
\text{Distance}(k_1, k_2) = \begin{cases} 
1 & \text{if } v(k_1) = v(k_2) \\
0 & \text{if } v(k_1) \neq v(k_2)
\end{cases}
\]  

(1)

where \(v(k)\) represents the value of instant \(k\) on this kind of context.

Continues Distance function: If the type of the context is numeric (e.g. speed), the similarity is computed as:

\[
\text{Distance}(k_1, k_2) = \begin{cases} 
\text{Min}(\text{end}_{k_1}, \text{end}_{k_2}) - \text{Max}(\text{start}_{k_1}, \text{start}_{k_2}) & \text{if } k_1 \cap k_2 \neq \emptyset \\
\text{end}_{k_1} - \text{start}_{k_1} & \text{if } k_1 \cap k_2 = \emptyset
\end{cases}
\]  

(2)

where \(\text{end}\) represents the value of higher bound of \(k\), and \(\text{start}\) represents the value of lower bound of \(k\).

Semantic Distance function: If the context refers to a semantic concept (e.g. semantic location), we introduce the semantic distance function for calculating the distance between the two concepts by introducing a depth variable. Depth variable denotes the number of edges on the path from the given node to the root node \([11][11][11][11][11]\). \(\text{LC}(k_1, k_2)\) denotes the lowest common concept node of both \(k_1\) and \(k_2\). The similarity is calculated as follows:

\[
\text{Distance}(k_1, k_2) = \frac{\text{depth}(\text{LC}(k_1, k_2))}{\text{depth}(k_1)}
\]  

(5)

Vector Distance function: If the context is a vector (e.g. required sensors), the distance is computed as follows:

\[
\text{Distance}(k_1, k_2) = \frac{n(k_1 \cap k_2)}{n(k_1)}
\]  

(6)

where \(n(k)\) is the number of elements in the set \(k\).

Preliminary Service Matching

In the initial matching phase, context services and context requests are only coarsely checked. More precisely to pass this phase, the following conditions must hold:

- Entity matching: The requested entity or entity type is a) the identical entity respectively entity type as offered or b) a generalization of the offered entity respectively entity type.
• Scope matching: The requested scope is a) the same scope as the offered scope, or b) is a generalization of the offered scope or c) matches with a nested scope of the offer (e.g. the representation of the offer is a composite representation and the requested scope is one of the dimensions of this representation).

• Access mode matching: The Context consumer should have access to the requested subscription mode of context services provided by the context service provider.

**Constraint checking Algorithm**

The final phase of the complete matching process is to check whether the constraints of context service and context request fit or not. A constraint $CAP_{csj}$ of a context service $cs_j$ and a constraint $CAP_{cr_i}$ of a context request $cr_i$ match if and only if the conjunction of both constraints $CAP_{cr_i} \land CAP_{csj}$ is satisfiable.

As we explained in previous sections, each context entity is defined by using a composite predicate, where composite predicate is a set of Context Attribute Predicates combined with logic operators. Therefore, $CAP_{cr_i} \land CAP_{csj}$ is equivalent to equation (3).

$$\left((CAP_{OP}^{ca}cv_1 \land ... ) \lor ... \lor (CAP_{OP}^{cd}cv_n \land ...) \right) \land \left((CAP_{OP}^{c1}cv_1 \land ... ) \lor ... \lor (CAP_{OP}^{cd}cv_n \land ...) \right) \quad (3)$$

Further, by applying De Morgan’s laws, we can transform equation (3) to DNF.

$$\left((CAP_{OP}^{ca}cv_1 \land CAP_{OP}^{c1}cv_1 \land ... ) \lor ... \lor (CAP_{OP}^{cd}cv_n \land CAP_{OP}^{cd}cv_n \land ...) \right) \quad (4)$$

We can state that $CAP_{cr_i} \land CAP_{csj}$ is satisfiable if and only if at least one of the set of products in equation (4) is satisfiable. To compute the level of satisfiability, we developed an algorithm that accepts a set of products and return two double values between 0 and 1. The first value denotes the similarity of context query and context service and the second value shows the confidence of the calculated similarity value. The constraint checking algorithm is represented in the Figure 13.
2.5.3. Node-red Integration

For visualization purposes, we integrate the current implemented prototype in Node-red. Node-red is a well-known flow based programming tools for the Internet of Things. It allows wiring together hardware devices, APIs and online services in new and interesting ways. It provides a browser-based editor that makes it easy to wire together flows using the wide range of nodes in the palette that can be deployed to its runtime in a single-click.

We developed four new custom nodes and added them to node-red. These nodes are CoaaS, Context Service, Context query, and Car entity which is a specific type of context query. By using these nodes, it is possible to generate a flow to register context service, and execute context quires. Figure 14 shows

\[
\begin{align*}
1. & \quad VistedConcepts = \emptyset \\
2. & \quad Branch = Branch 1 \\
3. & \quad similarity = 0 \\
4. & \quad unknownWeight=0 \\
5. & \quad knownWeight=0 \\
6. & \quad For each Node in Branch \\
7. & \quad \quad If Node.Concept in VistedConcepts \\
8. & \quad \quad \quad go to 3 \\
9. & \quad \quad End if \\
10. & \quad requestExpression=null \\
11. & \quad ServiceExpression=null \\
12. & \quad while node != null \\
13. & \quad \quad if (node.expression belongs to Context request) \\
14. & \quad \quad \quad requestExpression = requestExpression \cap node.expression \\
15. & \quad \quad else if (node.expression belongs to Service Description) \\
16. & \quad \quad \quad ServiceExpression = ServiceExpression \cap node.expression \\
17. & \quad \quad end if \\
18. & \quad node = node.parent \\
19. & \quad end while \\
20. & \quad VistedConcepts = VistedConcepts \cup \{Node.Concept\} \\
21. & \quad if (requestExpression not null and ServiceExpression is null) \\
22. & \quad \quad unknownWeight = unknownWeight + requestExpression.weight; \\
23. & \quad \quad go to 6; \\
24. & \quad \quad end if; \\
25. & \quad knownWeight = knownWeight + requestExpression.weight; \\
26. & \quad similarity = \frac{unknownWeight} {knownWeight + unknownWeight} \times distance(requestExpression, serviceExpression) \times requestExpression.weight \\
27. & \quad End for \\
28. & \quad confidence = \frac{knownWeight} {unknownWeight + knownWeight} \\
29. & \quad similarity = \frac{similarity} {knownWeight} \\
30. & \quad Output : similarity,confidence
\end{align*}
\]

Figure 13. constraint checking algorithm
a sample workflow developed to demonstrate the car-park use-cases. Figure 15 shows the related CDQL class diagram.

Figure 14. Node-red based example
Figure 15. CDQL class diagram
3. CSMS - Context Storage Management System

Generally, the intercommunication of devices and data processing in IoT are facilitated by IoT platforms, which are defined by Gartner as “a software suite or a Platform as a Service (PaaS) cloud offering that monitors, and may manage and control, various types of endpoints, often via applications end users build on the platform.” [12].

Usually, an instance of an IoT platform serves inside the infrastructure of one organization. While it is already bringing some value to the organization by reducing the complexity of managing a vast number of devices, the data is still locked in a silo of a current organization. It looks more like an “Intranet of Things,” “VPN of things,” but not the true IoT. Stamper defines three phases of the IoT evolution, which are namely M2M phase, IoT silo phase and IoT Systems phase [13]. While the research and development progress in the field of IoT platforms has reached the maturity state, we are still located in the 2nd phase or, in other words, IoT silo phase.

The power of IoT systems phase (phase 3) is integration and interoperability features, when a developer can design application based on data located in various silos without a need for long discussions with many data producers, signing SLAs, defining rules and protocols. Then, an application itself can find needed contextually relevant information dynamically by joining the ecosystem with known standards and guaranteed rules. Applications with such capabilities are usually referred to as context-aware. A key in achieving context-awareness at an ecosystem level is in organizing a special-purpose middleware platform, which will be facilitating the context processing in IoT environment. In other words, this platform will provide Context-as-a-Service (CoaaS) [14] [15]. Moore et al. define CoaaS as “a framework designed to provide the components required to build context-aware systems” [16]. We are using a term ‘CoaaS platform’ to refer to a middleware IoT platform that provides Context as a Service.

The most important function of the CoaaS platform is to provide ways for entities of the ecosystem to discover each other and request/exchange context in an interoperable and secure way. At the same time, we cannot say that the CoaaS platform acts only as a redirector. For many tasks having direct access to a set of data sources is not enough for the end-user application. First of all, stakeholders may agree to provide their data to a platform for making aggregations and distribute results, but not to distribute real data records of individual measurements. The CoaaS platform provides a needed trust level, which a typical application cannot provide. Secondly, retrieving needed contextual information can be very resource consuming for an individual organization both at the development and operational phases. The CoaaS platform does it instead of a client and reuses resources for many applications, bringing sufficient economic advantages. Thirdly, a CoaaS platform must provide interoperability features, transforming incoming data to a standardised format with the help of wrappers. Distinguishing features of a CoaaS platform are (i) the scale, (ii) multiple stakeholders, (iii) a broad range of use cases with completely different requirements, (iv) complex security, billing, and governance mechanisms.

All the discussion above leads to the conclusion that a CoaaS platform needs to provide means to collect, store and analyse massive context-related data in order to reason and deduce context.

The fundamental challenge that will be faced by CoaaS is to process and manage enormous amounts of context stemming from IoT data. We refer to this IoT data as big data due to the 3V (volume, variety, and velocity) characteristics of such data. Some of this data will have to be stored for such purposes as acquiring historical context, mining knowledge patterns, security, privacy, service discovery and other purposes. The need for storing and processing context stemming from IoT big data in near real-time dictates the need for organising a specialised storage subsystem. We call this subsystem a Context Storage Management System (CSMS). Researching, architecting, implementing and evaluating the CSMS is the main objective and expected contribution of this research.

The discussed research area is of high significance nowadays. For instance, Gartner identified IoT platforms, event stream processing, and IoT ecosystems among the top IoT research directions for 2017 and 2018 [17]. Among the types of ecosystems that are likely to emerge Gartner points out the Smart
Home, healthcare, and Smart City ecosystems. It proves our research is done in the right time and is just at the edge of innovation.

This research is related and contributes to the bloTope project [18], which aims to build an IoT open Innovation Ecosystem for Connected Smart Objects (CSO) with the primary application domain of Smart Cities. The bloTope project is a part of European Union’s Horizon 2020 Programme and is funded by the European Union. Data61, CSIRO is a partner in bloTope project and hence the link of this research with the bloTope project.

Among the key objectives of bloTope project, we would like to highlight the following, as they are directly connected with the scope of this research: (i) enabling interoperability between smart objects and vertical IoT silos by developing standards for open API’s, (ii) enabling creation of novel intelligent context-aware services, (iii) establish a framework, which will facilitate access to IoT data with respect to security, privacy, and trust.

For illustrating the capabilities of the proposed system, we have chosen the smart parking as one of the possible motivation scenarios. This use case can be significantly enhanced by additional contextual services which increase near real-time awareness and availability of critical data for decision support. Schematically, the process is presented in Figure 16.

It is well known that the process of parking in a modern city can take a significant amount of time and effort. Knowing in advance the exact place where to can park a car will significantly improve the driver's experience and reduce the time loss. Nowadays, garages and on-street parking are deploying sensors that can detect the availability of parking lots. However, modern cars do not have access to this data. If a car or manufacturer’s backend system could join the ecosystem of a Smart City and retrieve the contextual information about available parking lots, it would be possible to integrate this functionality into the vehicle’s navigation system. For a car manufacturer, it leads to a considerable simplification of the process in comparison with finding all possible data sources, adjusting the backend to their APIs and data formats and managing endless integration problems on site.

The car manufacturer now does not need to establish contacts and sign contracts with all managers of garages and parking lots in all the cities of the world, which is an impossible task. Instead of this, based
on the car’s location, the navigation system chooses the appropriate Smart City platform and gets all the needed context in real time.

For this scenario, the CoaaS platform considers a vehicle as context consumer and parking sensors as context providers. All the context providers have to be registered in the system for enabling the possibility of service selection or composition. Context consumers may be not registered and get the basic level of service in that case. If registered, a consumer's profile can be stored in the system, which would help in improving the quality of service (QoS). In our case, the profile can contain the size of a car, diver’s experience, habits, and preferences. The choice of a parking place can depend on its cost, distance from the desired location, weather, driver’s calendar, number of people on board and many other parameters, like the presence of the RFID-tagged umbrella in the car. In another case, a driver might be searching for a parking place with integrated charging point for electric vehicles. After providing a list of possible providers, the platform follows the driver’s choice, books a parking lot and monitors the process to ensure the quality of service.

3.1. Context storage requirements

In this section, we identify and discuss requirements for a context storage middleware. These requirements could be somewhat different from context representation that is used in edge computing because of significant differences in computing power, value and veracity of data streams and durability expectations.

Endpoint applications need to acquire context from various sources. The only way for these application to get context about the outside world is to communicate with some middleware, as communication with enormous numbers of sensors is not feasible due to many restrictions. These restrictions include such problems as network bandwidth, energy efficiency, access control and complexity of the task. The middleware receives requests for context from clients and tries to fulfill these demands. For this, the middleware platform should either store all the information inside, or query some other systems for retrieving the needed data. The first approach is disc space consuming, but it can improve performance. For example, modern search engines use indexed information for providing search results. The second method is time-consuming, as querying other systems and especially mobile sensors and devices can be a time-consuming process due to networks delays and slow response time or inaccessibility of mobile data sources. The IoT middleware may combine both approaches that will result in a better balance of disc space consumption, performance and data relevance.

We have identified the following requirements of a context storage middleware:

**Disk-based** – although in-memory systems are getting more attention nowadays, the amount and variety of data make processing not possible without keeping data persistently on disk.

**Scalability** - it is hard to predict the amount of stored information, but in the case of most IoT application domains, it would not be possible to provide the storage service by one server node. This means that proposed solution must be horizontally scalable.

**High Availability** – the storage should not have a single point of failure (SPoF).

**Data structure flexibility** – storage must be able to store structured data without applying restrictions on its structure.

**Entities connectivity** – in some cases storage must facilitate the means for storing highly interconnected data (e.g. relations between people, organisations, transport, infrastructure, etc.) and effectively running queries over such data.

**Veracity** – different sources can supply information that can be conflicting or uncertain and there should be a way to store all variants of incoming data with annotations about the identity and trust level of the originator and rank of the suggestion. The context of a querying side must be treated respectively during responding to the query.
**Large amounts of sensory data** – sensors and other Internet-enabled devices are generating vast amount of time series events of similar but not the same structure.

**Ontology support** – a number of research projects [2] model data using ontological principles as it is a good way for modelling the domain interconnections and facilitating reasoning over data. However, this approach does not seem to be suitable for storing large amounts of raw data and low-level context.

**Fast information retrieval and rich indexing capabilities** – performance is the key requirement for context delivery in smart cities applications. This highlights the need for efficient indexing of stored context.

**Fast data ingestion** – streams of sensor readings must be written on disk without long queues and expensive rebuilding of indexes.

**Geospatial data** – many of IoT applications are highly dependable of geospatial context, so the middleware storage must be able to provide efficient mechanisms for working with this type of context.

**Transaction management flexibility.** Traditionally, one of the main principles of database management systems is ACID – Atomicity, Consistency, Isolation and Durability. According to the CAP theorem, we cannot have consistency, availability and partitioning tolerance in one system at the same time. As mentioned, the context coming from different sources can already be uncertain and conflicting. That means, the middleware solution in some cases can relax the transactional support and consistency in favour of high availability and partitioning, as the requirements for horizontal scalability and availability have higher priority. At the same time, some parts of middleware system can have strong requirements for consistency, and these requirements must be satisfied.

After analysing the requirements for the middleware storage system, it becomes clear that fulfilling all the requirements with one existing solution is not feasible. The variety of data processing approaches leads us to the idea of hybrid storage architecture or, in other words, polyglot persistence. [19, 20]. In polyglot persistence, systems no longer try to accomplish all tasks using one data storage, but rather use different technologies to store data where each technology provides certain capabilities. In [19] a use case of PolyglotHIS, a health information system using three different databases is presented. The system uses a relational database (PostgreSQL) for storing structured transactional data, document-oriented data store (MongoDB) for storing schemaless documents and graph datastore (Neo4J) for storing data containing relationships. PolyglotHIS implements various software agents to achieve interoperation between involved data stores In [20] polyglot persistence approach is used to Enhance Performance of the Energy Data Management System (EDMS). EDMS uses MySQL, MongoDB and OpenTSDB [21] that runs over HBase.

In the next section, we briefly describe the several popular approaches for context modelling, representation and storage. In respect to the storage technologies we present several open-source software projects in each area.

### 3.2. Existing context representation and storage technologies

**Key-Value** is a popular NoSQL representation and storage technique that represents any information with a key association and retrieves it by the given key effortlessly and quickly. The key-value modelling is the fastest, easiest and noticeably scalable way of retrieving information from storage. However, standards, schema, verification and relations between entities are not offered. The most important point from our perspective is the absence of means for searching inside values, making it possible to request data only by key.

**Document-oriented** or Markup scheme tagged encoding is another NoSQL technique for representing context. It is still very flexible and scalable but allows to organise data in structures, usually using JSON as a serialization format. Documents are organised in collections and the most important – there are
ways to organise different types of indexes over collections, making fast queries possible. Data denormalization is a strong and at the same time weak point of this approach. It is fast to retrieve and write, but the data can easily become inconsistent. Furthermore, document-oriented approaches consume more disk space in comparison with the relational approaches due to applying data denormalization as a main data modelling technique. Organizing relations between documents is possible, but document storage engines usually do not support joins, as it assumes that higher-level software components should do this work. The most widely used document-oriented stores are MongoDB [22] and CouchDB. JSON-LD fits naturally with MongoDB document model. Another example of the document-oriented data store is ElasticSearch [23]. It is a multi-tenant search engine based on Lucene. The difference between ElasticSearch and other document-oriented data stores is its ability to automatically create mappings and index documents of structure that was not defined in advance. ElasticSearch indexes data using inverted lists and wide-columns based on the type of incoming data. ElasticSearch uses a specialised JSON-based query language called Query DSL. Another distinguishing feature of Query DSL is the presence of scoring function that enables data search based on unprecise queries with computing the relevance of returned data.

A relational database is another way of context storage. Relational database management systems (RDBMS) technology is one of the most well-established technologies and has been used as the main approach to data management for more than 40 years. Allowing an excellent level of stability, functional richness, knowledge base and other benefits, relational model has a deliberate disadvantage for modelling context – it is the rigid schema that makes it hard to store any information that is not structured in the way that is defined by the relational schema. Another problem is the expensiveness of joins between tables. Most popular open-source relational databases are PostgreSQL and MySQL.

Ontology-Based Modelling is a way of organizing context into ontologies using semantic technologies like RDF or OWL. A large number of development tools, reasoners, standards and storage engines [24] are available. Ontologies give capabilities for defining entities and expressing relations between them. However, when dealing with Big Data, retrieval of context can be resource consuming, and issues with scalability may arise. Besides, ontologies are not recommended for representing streams of sensor data. Examples of RDF storage engines are Jena2, Sesame, AllegroGraph, Virtuoso, etc. [24] Most popular serialization formats are Turtle, N-Triples, N-Quads, N3, RDF/XML and JSON-LD.

Graph-based modelling is a natural way for representing entities and interconnections between them. They are ideal for representing unstructured information and information that has ambiguity. Graphs are typeless and, schemaless, and there are no constraints on relations. This structure is great for representing social networks and is recommended for read-mostly requirements. Graph databases have a lot in common with RDF storages but use different languages for querying data. Some graph databases can be used as RDF storages with special plugins applied. According to [25] the popularity of graph databases has increased by 500% within 2014-2015 years period. Most popular Graph Databases are Neo4J, Titan, and OrientDB.

Object Based Modelling. Numerous projects focus on context-awareness common object-oriented programming languages technique of modelling context as objects [26, 27].

One of the theoretical foundations of CoaaS is the Context Spaces Theory (CST) which enables context representation and reasoning about situation awareness [8]. It uses geometric metaphors for representing context attributes and building multidimensional spaces. Special context situations algebra is used for situation detection and prediction.

The visualization of a situation subspace and context-situation pyramid [28] in CST is presented in Figure 17 CST proposes steps to a generic framework for context-aware applications and provides a model and concepts for context description and operations over context. This theory is implemented in two frameworks ECORA [27] and ECSTRA [26] and has been extended in Fuzzy Situation Inference (FSI) [27] for situation modelling and reasoning under uncertainty and other advanced reasoning capabilities. These frameworks use the aforementioned object-based modelling approach and do not focus on issues such as scalability or persistence. Developing methods for mapping the CST approaches to scalable
and efficient storage can help to implement these methods in large-scale IoT middleware usage scenarios.

These projects propose a solid theoretic base and develop advanced features for context processing without focusing on the persistence problem that makes them hard to use in a large-scale environment. Though numerous attempts were taken to develop object storage, the industry standard is still mapping objects to a relational database schema. This task is usually performed manually or with a special object/relational framework facilitating the automatic process of mapping entities and hiding the persistence level under ORM abstractions [29]. The main problem with this approach is called object-relational impedance mismatch [30], which represents a set of difficulties while transferring data from object model with polymorphism, inheritance, and encapsulation to the de-normalized table-based database approach.

Our research of context representation approaches is summarised by providing quantitate analysis in Table 2. We use the following designations: Disk based (D); Relations (R); Veracity (C); Geospatial data indexing (GSI); Storage of Sensory Data (SD); Schemaless/Structural data freedom (SL); Horizontal Scalability (HS); Fast Writes (FW); Strong/native support (++); Supported (+); Limited support (+/-); Not supported (-).

According to the analysis of context representation and storage techniques, we identify the document-oriented approach as the most suitable for our purposes.

Table 2. Summary of context representation approaches and their intersections with CoaaS platform storage requirements

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>R</th>
<th>V</th>
<th>GSI</th>
<th>SD</th>
<th>SL</th>
<th>HS</th>
<th>FW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relational</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+/-</td>
<td>-</td>
<td>-</td>
<td>+/-</td>
</tr>
<tr>
<td>Ontology</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Key-Value</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+/-</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Document</td>
<td>+</td>
<td>+/-</td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td></td>
</tr>
<tr>
<td>Wide-Column</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>++</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Graph</td>
<td>+</td>
<td>++</td>
<td>+</td>
<td>-</td>
<td>++</td>
<td>+/-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Object</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>++</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
</tbody>
</table>

As we already mentioned in section 2, the CoaaS platform, besides its unique features, will combine the work done in the field of context-awareness with the work done in the area of IoT platforms. For that reason, we have studied the data management components of IoT platforms. The findings are presented in the next section.
3.3. Data management in existing IoT platforms

CoaaS platform is an IoT platform, which facilitates interaction between heterogeneous systems and based on the principles of context-awareness. For that reason, we researched how the data management is implemented in the field of IoT platforms, as having this information will help in understanding the architectural patterns, which can be useful for the CoaaS storage subsystem.

3.3.1. Overview of data management in existing IoT platforms

In this section, we analyse how different IoT platforms deal with data management (storage) issues. We have grouped the studied platforms into two categories: commercial IoT platforms and academy/research projects. This division is caused by significantly different approaches applied by these two groups. The difference in targets and approaches makes it hard to compare projects from various groups, but we believe that best practices from one group can be applied in another for improving the functionality and performance.

Avoiding technologies that were not seriously tested in real-world applications, focusing on security, performance and cost efficiency are essential features of all commercial solutions. These solutions are often cloud-based and use a Software as a Service (SaaS) or PaaS model. Their main target is to allow customers straightforward and rapid development of applications that will connect companies’ “things” together. These solutions provide interoperability regarding various sensors and protocols, but the problem of horizontal data exchange between enterprises, state organizations, and individuals is not addressed. Research projects, on the other hand, provide open-source software that can be deployed on-premises and maintained by organizations themselves. These projects often focus on semantic and horizontal interoperability and while providing cutting edge functionality in some aspects can lack functionally or performance in others.

In this section, we are focusing on data management approaches, for instance, how data is ingested, stored, indexed, cached and retrieved by the IoT platform.

3.3.2. Commercial IoT platforms

Predix [31] is a PaaS IoT platform developed by General Electric and aiming to provide services for data collection and processing in the area of Industrial Internet of Things. Predix has a catalogue of provided services that includes a set of tools for data management. Predix is not trying to produce a one-fits-all solution and proposes to use one or several services best suited for the current task. These services are (i) Asset Data, (ii) Time Series, (iii) SQL database, (iv) Blobstore, and (v) Key-Value store. RabbitMQ – a message queue based on AMQP protocol, organizes communication of components. Asset data is a set of models used to describe machines and instances, which are created basing on these models. Time Series service provides means for efficient ingestion, distribution, and storage of sensory data, including indexing for making fast queries. Predix uses a graph database for its asset service to store data as RDF triples. A special Graph Expression Language (GEL) is used for data retrieval [32]. SQL database service is built on top of well-known open-source PostgreSQL database. Blobstore service provides means for storing and retrieving any amounts of binary data and ensures high availability and horizontal scalability. Key-Value store service is based on open-source Redis project and serves as an advanced cache store. Predix uses a hybrid (or polyglot persistence) storage solution, but all the responsibilities on choosing the right options are left to the application developers.

Data services is a promising feature of the platform that is in the beta stage. There are only two services available: Places data services and Seismic data services. This is a remarkable step to horizontal IoT solutions. The platform provides easily accessible data from external data sources or sensors to application developers making it possible to adapt industrial automation solutions to detected earthquakes or other accidents.
**Tibbo AggreGate** [33] is an example of a commercial non-cloud IoT platform. All data is logically separated into two groups: (i) configuration and (ii) events. This approach helps in providing flexibility of data storage in case of adding new business objects. Configuration data can be stored in almost any enterprise-grade relational database that supports JDBC connectivity, key-value database or in a file-based storage. In case of a relational database, the AggreGate platform includes an embedded database or a preconfigured version of MySQL. AggreGate provides means for database clustering for achieving high availability. Key-value integrated storage is recommended for scenarios, which need clustering together with a high update rate. File-based storage can be used in environments with limited resources. Event data can be stored in a relational database, NoSQL database or in-memory storage. RDBMS puts some limits on insertion performance. NoSQL database provides horizontal scalability, high insertion rates, and failover facilities. Memory storage is proposed for use in some of the embedded installations. Approximate estimated insertion rates for a relational database are about 500-2000 events per second and 10-20 thousand events per second for a NoSQL database on a regular server node [34]. AggreGate also provides functionality for building a failover cluster and achieving high availability.

**ThingWorx** [35] is a cloud-based platform enabling developers to build solutions for IoT. It provides three main ways for data storage: (i) Data Tables, (ii) Streams and (iii) Value streams. ThingWorx also uses concepts of an InfoTable and a DataShape.

The InfoTable is a JSON document in which all the objects share the same properties. InfoTables are fast in-memory objects and are recommended for storing temporary data. The DataShape specifies what property names are required in an object and what types they have. In other word, a DataShape represents a schema for defining a “thing.”

The concept of a DataTable in ThingWorx is similar to a table in relational databases, but columns are defined by a DataShape. A DataTable supports the creation of indexes upon its properties. It is recommended to build an index for each common request for achieving high performance and to use DataTables when it is expected to have not more than 100000 rows in it. Storage of time series data is facilitated by streams. A stream consists of a timestamp and additional properties defined by a DataShape. For dealing with things-driven models, it is recommended to use Value Streams, which have some differences with ordinary Streams. Value Streams provide persistence for associated property and return only property values on request. On the contrary, a stream returns a whole row when querying a single column.

**Amazon AWS IoT** is a cloud-based platform that makes use of all the impressive technological stack provided by Amazon. Communication between devices and cloud is organized by a Device Gateway which supports the publish/subscribe approach. The Rule Engine provides means for configuring rules for filtering and transforming incoming events. This configuration includes routing of data to various supported databases, messaging queues, AWS Lambda, and other services. Registration and monitoring of connected devices can be made via Device Registry. The configuration of processing rules for the device is performed in a JSON document consisting of an SQL statement and an action list.

Amazon’s IoT solution uses storage technologies provided by Amazon Storage Services. Amazon Storage Services focus on providing scalability, availability and elasticity for mostly well-known storage technologies and promote a so-called NoDBA approach, which reduces operational costs for customers. The variety of provided storage services includes Amazon DynamoDB, Amazon RDS, ElastiCache, and ElasticSearch. Amazon DynamoDB is a cloud managed NoSQL key-value store, but a version for on-premises installations is also available. Amazon Relational Database Service (Amazon RDS) can use any of the six most popular relational databases. Amazon’s in-memory data store cloud service is represented by so-called “ElastiCache”. This service can significantly improve system performance by reducing the number of slow disk reads. ElastiCache is based on two popular open-source in-memory engines: Redis as an in-memory data store and Memcached as a system for object caching. For such use cases as device log analysis and real-time monitoring of applications Amazon recommends the ElasticSearch service that is based on a popular cognominal search engine. Amazon IoT platform uses a messaging system based on Kafka-based named “AWS Kinesis” for event broadcasting. Capturing and
loading streaming data is performed by Kinesis Firehose and analytical processing of streaming data is done by Kinesis Analytics [36, 37]. Amazon AWS IoT introduces the “thing shadow” or “device shadow” concept. A special “Thing Shadows” service is responsible for managing fast and easy access to a JSON document with a current state of the device that was last reported to the platform.

**IBM Watson IoT platform** relies on the IBM Cloudant database. It is a cloud fully managed document-oriented database sharing many common features with Apache CouchDB. IBM recognizes the need for flexible storage solutions, but by now their solution is mostly document-oriented. Describing plans for future, IBM’s specialists state that variety of tasks causes different requirements to latency, scalability, cost and performance, raising the necessity for various storage solutions [38]. By now, data from devices can be stored in two formats. If the API receives a valid JSON, it is stored in the same way. In another case, the data is saved as a base64 encoded string inside the payload field of a JSON document. Recently IBM introduced a feature named “Last Value Cache.” As the most common request to an IoT device is about its current state, it makes sense to provide a way for answering such requests in the fastest possible way. These cached values are stored for 12 months and can be retrieved using the standard API.

Many other PaaS platforms including Carriots, Xively, Zatar, Realtime.io and Flowthings.io provide only APIs for uploading or downloading data and do not provide information about their backend and storage architecture. **Table 3** offers a comparison of various data capabilities in IoT platforms.

### 3.3.3. Academic/research projects

The **FIWARE** community is aiming to create an open ecosystem that will enable development of Smart Applications. This ecosystem is based on royalty-free standards and covers a broad range of tasks. Software for the category of tasks is grouped into a module called “generic enabler” (GE) [39].

FIWARE provides several generic enablers for dealing with various types of storage. The central module of FIWARE ecosystem is the Orion Context Broker. This component uses a connector called “Cyggnus” that is responsible for persisting or retrieving data from a specific storage. The current release of Cygnus can communicate with HDFS, MySQL, PostgreSQL, CKAN, MongoDB, Comet, Kafka, DynamoDB and CartoDB.

Time series data in FIWARE ecosystem is managed by a component called Comet or Short Term History (STH). This component deals with the storage, retrieval, and removal of raw time series data as well as aggregated context information. This component relies on MongoDB as the datastore.

Semantic Application Support (SAS) GE provides the possibility for developing applications based on Semantic-web technological stack. In [40] Ramparany et. al. suggest that OWL and other Semantic technologies can help in solving such problems as (i) Semantic data interoperability, (ii) data integration and abstraction, (iii) data discovery, and (iv) reasoning. FIWARE developers admit that despite large investments and development of markup and query languages the progress with penetration of Semantic web technologies into the market is still too slow. They identify several reasons, which include both technical and commercial problems. SAS GE tries to solve technical and engineering problems, namely (i) scalability, (ii) performance, (iii) distribution (iv) security, (v) lack of methodologies and best practices, and (iv) lack of development instruments. The GE consists of a GUI client and server-side components that are responsible for storing and managing ontologies. Server-side components provide scalable and secure ways to publish and retrieve metadata as well as instruments for managing the infrastructure and data.

The data layer of SAS GE consists of a relational database that stores information about ontology documents, and a Knowledge Base that supports OWL-2RL. By now, there is no knowledge base-independent solution, and the knowledge base is implemented as a combination of Sesame and OWLIM [41].
OpenIoT (Figure 18) is an open source IoT platform, which includes a set of novel functionalities [42], namely:

- Incorporation of IoT data and applications inside cloud computing infrastructures;
- Providing secure access to semantically interoperable applications;
- Enabling and supporting discovery of sensors and data at run-time;
- Supporting mobile sensors and corresponding QoS parameters.

In OpenIoT (Figure 18) the registration, data acquisition, and deployment of sensors are managed by X-GSN. X-GSN is an extension of the GSN [43], which is responsible for semantically annotating both sensor data and metadata. The virtual sensor is the main fundamental concept in X-GSN, which is capable of representing any abstract entity (e.g., physical devices) that collects any parameters. To make a virtual sensor accessible from the rest of the OpenIoT platform, each virtual sensor needs to register within the Linked Sensor Middleware (LSM). LSM is another core component in OpenIoT that is responsible for handling the sensor data delivery chain. In this regard, LSM transforms and annotates (based on the supported ontologies) the data coming from virtual sensors (through X-GSN) into a Linked Data representation i.e., RDF, and stores it in the database. The OpenIoT platform relies on OpenLink Virtuoso (it is also known as Virtuoso Universal Server) as the main database. OpenLink Virtuoso is a hybrid database engine that combines the functionality of a traditional RDBMS, ORDBMS, virtual database, RDF, XML, free-text, web application server and file server functionality in a single system [44]. According to information on the website, Virtuoso can handle the insert rate of 36K triples per second on a single 4-core machine.
Table 3. Underlying storage technologies of IoT platforms

<table>
<thead>
<tr>
<th>Solution/Feature</th>
<th>IBM IoT</th>
<th>Predix</th>
<th>ThingWorx</th>
<th>Tibbo AggreGate</th>
<th>Amazon IoT</th>
<th>FI-Ware</th>
<th>OpenIoT</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/L</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>H/L</td>
<td>L</td>
</tr>
<tr>
<td>TS</td>
<td>IBM Cloudant</td>
<td>Column storage</td>
<td>Streams (Cassandra)</td>
<td>NoSQL</td>
<td>DynamoDB</td>
<td>Comet</td>
<td>-</td>
</tr>
<tr>
<td>R</td>
<td>-</td>
<td>PG</td>
<td>PG (any JDBC)</td>
<td>Embedded, MySQL or any other enterprise-grade DB</td>
<td>RDS (6 databases)</td>
<td>MySQL-PG</td>
<td>Virtuoso</td>
</tr>
<tr>
<td>D</td>
<td>Cloudant (Couch DB)</td>
<td>JSON</td>
<td>Integrated NoSQL</td>
<td>ElasticSearch</td>
<td>MongoDB</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>KV</td>
<td>-</td>
<td>Redis</td>
<td>-</td>
<td>Integrated NoSQL</td>
<td>DynamoDB</td>
<td>Comet</td>
<td>-</td>
</tr>
<tr>
<td>IM</td>
<td>-</td>
<td>Redis</td>
<td>InfoTables</td>
<td>Integrated</td>
<td>Redis/ Memcached</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>MQ</td>
<td>MQTT</td>
<td>RabbitMQ</td>
<td>MQTT</td>
<td>-</td>
<td>Kinesis</td>
<td>RabbitMQ</td>
<td>-</td>
</tr>
<tr>
<td>HS</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>SW</td>
<td>-</td>
<td>RDF Graph</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

C = Cloud, L = Local/on-premises deployment, R = Relational, D = document-oriented, KV = Key Value, IM = in-memory, BD = big data, HS = Horizontal Scalability, SW = Semantic Web stack, MQ = Messaging Queue, PG = PostgreSQL, “-” = Not described in documentation or not implemented.

3.3.4. Benchmarking and Evaluation

Comparing the performance of different IoT systems is a complex task. The complexity grows exponentially with the number of features and possible use-cases. However, performance benchmarks are needed for both consumers and developers. Product consumers can rely on the benchmarking results for making a better choice and developers can analyse weaknesses of their product, improve and demonstrate the results. In the world of transactional databases, this effort was started and supported since 1988 by a non-profit organization called Transaction Performance Council (TPC) [45]. Actual benchmarks are TPC-C, TPC-H, TPC-E, TPC-DS, TPC-DI and TPCx-HS which cover such areas as OLTP, ad-hoc DSS, complex OLTP, complex DSS, data integration and Big Data. In the NoSQL movement, which has significant differences in approaches with classical transactional world, the most popular benchmarking approach is the Yahoo Cloud Serving Benchmark (YCSB) [46] which is supported by some open-source tools [47]. Semantic web community also introduces a number of benchmarking strategies for RDF Stores [48].

Discussion of benchmarking strategies for the IoT platforms has already started [49], [50] and some attempts are already made [51, 52]. A significant research work in the field of industrial IoT cloud platforms is done in General Electric. Williams et al. described benchmarking of an industrial solution based on in-memory data grid in [53]. HP develops the IoTAbench [54] with an initial focus on use-cases like smart metering. The problem is in the variety of vendors’ understanding of the IoT platforms principles, tasks, main features and system complexity in general. A common way for IoT platforms is to integrate several storage and caching technologies for different types of data. In the dynamic world of IoT, automatic data management strategies are becoming more and more valuable. These data management
strategies, supported by the ingestion and retrieval pipelines will affect benchmarking results at the same level as the underlying storage technologies.

### 3.3.5. Discussion

After analysing the data storage capabilities of several commercial and research IoT platforms, we have identified that such platforms mostly tend not to limit developers in their choice of a data storage format. Some IoT platforms introduce their own storage technologies; others offer well-known open source or commercial solutions. Mostly these platforms offer the following storage types: (i) in-memory, (ii) document-oriented, (iii) column-oriented, (iv) relational, and (v) RDF. Organization of blob storage is performed using OpenStack Swift. Scalability and high performance of message queuing are achieved by using technologies like Apache Kafka, RabbitMQ, or ZeroMQ. For Big Data processing IoT platforms usually rely on Apache Hadoop or Apache Spark. Research prototypes often use RDF or OWL, but this trend is still mostly avoided by commercial companies due to the issues with scalability and low industry penetration of Semantic Web technologies.

It is also worth noticing that some of the discussed platforms are developing and introducing new features at a very high pace so that we can expect serious changes in the market in the nearest future. Table 2 displays key features of several popular IoT platforms from an IoT data storage angle. It is easy to notice that several platforms rely on multiple technologies for data ingestion and storage instead of using one approach for all types of data.

Another field of ongoing research is benchmarking of IoT platforms. Some initiatives in the filed already exist, but the stage of agreement is not yet reached. Mostly, proposals cover limited industry-based use cases.

Performance evaluation of the CoaaS storage subsystem will need to integrate benchmarking approaches from IoT and database communities together with specifically designed parts covering the distinguishing features of CoaaS.

### 3.4. Proposed Context Storage Management System

In this section, we propose the CSMS which is a core component of the CoaaS platform for bloTope project [15, 55].

First of all, we need to briefly describe the CoaaS platform, its general architecture and objectives. Then we focus on a core component of the platform - the Context Storage Management System (CSMS). We discuss how the storage subsystem facilitates the whole process of processing the context query, which parts it consists of and how these parts communicate with and support each other. The CoaaS framework is being developed with an aim for being operated in large-scale IoT environments, so it must be adapted for dealing with big data. It means that the scalability of the CSMS becomes one of the main requirements as well as challenges [56].

The conceptual architecture of CoaaS middleware is presented in Figure 19 followed by a brief discussion of CoaaS. The main aim of the CoaaS platform is facilitating context exchange between a large number of consumers and providers. In general, context providers can be any device or system that can provide contextual information or answer requests. Context consumers are any devices or systems that can make requests to the system. The range of providers and consumers starts with a tiny sensor that belongs to an individual citizen and goes to an enterprise-class software system operated by a company or governmental organization. Another instance of CoaaS middleware can also be a consumer or provider. The will of participating stakeholders to use context or share/sell raw data for mutual benefit is the force that brings together consumers and providers of context. When a consumer needs context, it sends a context query to CoaaS. The query is defined using a CDQL language [15].
The CoaaS platform parses the context query, finds the most suitable context providers, retrieves data, processes it according to the query and sends the results back to the consumer. A query can be executed once, but a consumer can also register a query in the CoaaS system to provide permanent situation monitoring. A context provider can also push information to CoaaS periodically or on an event basis. For example, Figure 20 presents a CDQL push-based query to notify cars in the school areas to reduce their speed during the school time.

```
PUSH "reduce speed" to car
WHEN Distance (car.location, school.location) lt 500
AND time between 2:00pm and 3:00pm
DEFINE Entities as
  Entity school is organization
    WHERE school.organization-type eq "school"
  Entity car is Object
    WHERE car.type eq "vehicle" AND
    car.Situation="driving" AND car.speed gt 40
```

Figure 20 - Push-based CDQL query

Context middleware brings advantages that are impossible to achieve in a peer-to-peer system. First of all, it serves as a place where stakeholders of the system can find each other. Secondly, it can provide aggregated context to consumers without disclosing personal information of data contributors. Thirdly, it hides all the complexity of retrieving context in large-scale environments from the customer.
After initial research, it became clear that the CoaaS platform can not only retrieve information from providers, fuse it and send back to customers, serving just as a redirector or a reasoner. First of all, network latencies, need to query numerous sources of information and potential unavailability of these sources can lead to unexpected delays in serving the query. We expect the delay to be in sub-second range. Secondly, the process of deriving context is based on knowing patterns and history as well as predicting future context to enable proactive adaptation. The only way to acquire these patterns is storing the incoming data and continuously analyzing it. Reasons above bring us to the understanding of the need to store data and context as a core part of the CoaaS middleware platform which is currently under development. The next section describes the architecture of CoaaS storage management system in more detail.

### 3.5. 5.1 Conceptual architecture of Context Storage Management System

Principal elements of the context storage management system are: (i) the Storage Query Execution Manager (SQEM), (ii) Context Service Description Module (CSDM), (iii) Context Repository (CR), (iv) Performance Data Repository (PDR), and (v) a set of recommenders that enable efficient query processing and tuning the system performance. These components rely on the in-memory database for context caching, a document store, an analytical subsystem based on MapReduce and HDFS for collecting the CSMS usage statistics, and (potentially) a framework for working with semantic data (knowledge), which is individually integrated as a plugin for certain domains. These components communicate with each other using a messaging queue (MQ). Schematically the CSMS architecture is presented in Figure 21. We draw attention to the fact that the storage subsystem is a special case of a context provider. The final answer to a query may consist of data parts that were retrieved from internal context storage and parts that were received from context providers during the query execution process. The detailed description of the CSMS components is provided in the next section.

![Figure 21 - Conceptual architecture of CoaaS storage management system (CSMS)](image-url)
3.6. Components of CSMS Architecture

3.6.1. Storage Query Execution Manager (SQEM)

The SQEM is the entry point for context queries. The main CoaaS query execution engine passes the initial context query coming from consumer together with hints that were added by CoaaS main query execution engine during the parsing phase. First of all, the query execution engine needs to obtain a list of context providers, which can serve as data sources. For that reason, the context service description repository with selector service is used. Secondly, while serving a request to the context storage, SQEM has to decide which data should be retrieved and how it should be accessed. We aim to provide near real-time context query processing for large-scale deployments. For that, an efficient planning of resource usage is needed.

SQEM builds a plan that describes how to query all the storage components of the system. Based on the incoming query and the cost and security models it makes decisions (i) whether the caching layer should be queried or not, (ii) which and how many data sources should be involved, and (iii) should the context service selection procedure and all the following steps be initiated or it is possible just to reuse an existing plan. SQEM also checks the authorization permissions for the current consumer and finds the optimal balance between the cost of the request and the quality of context.

Next, SQEM constructs queries to the underlying datastores, retrieves data, sends it to appropriate reasoners if needed, and returns the result when it is ready to the CoaaS query engine.

3.6.2. Context Service Description Module (CSDM)

CSDM is the only source of information for the CoaaS query engine to understand which context providers can supply the needed contextual information. The result of work of the Context Service Selector is a list of possible data sources together with their properties. CSDM consists of three logical parts, which are presented in Figure 22. Context Services is the part responsible for describing services in general, not the real data sources. For instance, we can create a description of a service, which provides information about the availability of a parking place. We can think of the service description like a class in terms of object-oriented programming. Changes in the list of services happen relatively rare. Context services collection stores such parameters as “Service Name”, “Keywords”, “Description” and “Service Schema”. The Service schema contains a description, according to which a new document with information about a service can be validated. We propose to group service-description parameters in three categories: (i) compulsory, (ii) optional, and (iii) open. Compulsory parameters must be defined when a service provider registers itself in the system, otherwise even basic application scenarios would not be able to make use of the service. For example, in the case of a car park scenario, GPS coordinates of the car park are essential for the application. CSDM will reject the registration of a car park service provider without data about location. Compulsory parameters are standardized by some dictionary, so the semantics is clear to applications.

Optional parameters are also standardized, but a provider can omit these parameters in the case when they are not applicable, or there is no technical possibility to provide them. The open category of parameters contain any parameters, which a provider wants to include.
Another part of the CSDM is a collection of Context Providers. It is responsible for storing individual entities of context providers that inherit features of the context service and add their own custom properties. In the example with parking, a context provider is linked to a context service “ParkingSensor” and the information about location, availability, and pricing policy is added. Here, we need to explicitly mention that context providers can be mobile. Mobility, and, possibly, other dynamic features of data sources, make the ‘Context Providers’ collection in CSDM open for frequent updates. This is needed for the proper choice of mobile sensors during the service selection process. Main parameters of the Context Providers collection are the “Data Access Method” (how to access the provider, for example it can be a REST interface for a web service, or Google Cloud Messaging API for a mobile device), URI (where to send the request), location, mobility, the cost of the request, etc. Except for parameters that are provided by the context provider during the registering or update phase, there should be some parameters that are computed by the platform itself, e.g. expected latency and quality of supplied context. These parameters are calculated based on statistics that is stored in Performance Data Repository and used during the phase of service selection.

The third part of CSDM is the utility information collection. It provides links to historical data of the chosen provider and a link to a cached value of the last retrieved sensor measurement, in the case when a context provider is a sensor.

Incoming requests to CSDM include: (i) context provider selection queries, include (ii) registration of context services, (iii) registration of context providers that is initiated by providers themselves or by crawlers and (iv) update of providers’ state initiated by the Subscription Module (SM) and the Performance Data Repository (PDR).

The result of a query is a sorted (according to the cost model and consumer profile) list of providers that is returned to the main query engine.

3.6.3. Subscription Module (SM)

Subscription module, which is schematically presented in Figure 23, has a twofold purpose. Firstly, it facilitates the whole process of notifying the subscribers about a detected CSDL-defined situation. Each consumer can register an unlimited number of situations in the system, and all the incoming events will be ‘percolated’ through these queries. If needed, CoaaS will initiate periodical requests to data sources for keeping the situation state up to date. Another type of subscription is timer-triggered push-based context query. In this situation a consumer requests the CoaaS platform to perform a query and send results back based on a specified period.
Secondly, it provides means for keeping the data in CSDM updated according to incoming data streams. In this case, CSDM becomes a subscriber of a special type, registers itself as an internal subscriber in SM and subscribes for state updates of the context providers.

Main collections of the SM are (i) Subscribers, (ii) Situation Definitions and (iii) Actions. Collection ‘Subscribers’ contains properties of customers, that registered a subscription (e.g. endpoint where to send data in case of an external consumer, CSDM entity in case of internal state update registration). Situation definition collection is designed to contain a large number of situation definitions, which are represented in Context Definition Language (CDL). The Collection ‘Actions’ contains definitions of actions (scripts) according to which the SR module will act in the case when a situation is detected.

Figure 23 - Subscription module

Stream processing framework for SR

Context requests, and especially subscriptions to situations, are formulated in CDL that relies on Context Spaces Theory (CST) [8] based representation of context and description of situations. Existing CST-based frameworks provide tools for situation modeling and reasoning under uncertainty and other advanced reasoning capabilities.

These frameworks use the object-based modeling approach and do not focus on issues such as scalability or persistence. A massive number of incoming events must be efficiently processed for near real-time situation understanding. For this purpose, we need to apply a scalable and high throughput framework for working with data streams. Using CST situation reasoning over the stream framework facilitates the parallel and fast functioning of subscription module.

Potentially, other frameworks for context representation and situation reasoning can also be integrated for enhancing the capabilities of the SM.

3.6.4. Performance Data Repository (PDR)

For the purpose of managing the storage query process and adjusting the performance of the system, we introduce the Performance Data Repository. This component is designed to collect logs, history and statistical information about system workload, including incoming queries, plans that were built while processing these queries and metrics obtained during the query execution time. Collected data is used by specialized recommenders. These software components analyse collected data in background batches and produce decisions; based on these decisions the system adjusts data placement and indexing strategies according to the workload. These components are: (i) Cost-based adaptive indexing optimizer, (ii) In-memory caching and collocation and (iii) Proactive raw data retrieval (PRDR).
**Cost-based adaptive indexing manager (CBAIM).** CoaaS storage is a highly dynamic system with loads that vary and change over time. Moreover, queries are constructed by consumers, causing a problem to develop an optimal indexing strategy for data stores at design time. For that reason, we propose to include CBAIM in the system. This module relies on logs of data access, queries, plans, responses and traces. After periodical analysis of these records, CBAIM constructs commands for different parts of storage system for building new indexes or deleting old ones. Some work on adaptive indexing strategies can be found in [57].

Another job of CBAIM is to understand the needed granularity of stored data and suggest procedures for reducing it by applying suitable algorithms.

**In-memory caching and collocation (IMCC) module** is responsible for offering two types of recommendations: (i) what piece of information should be kept inside the in-memory caching layer, and (ii) how to collocate data in such a way, that the number of data transfers between cluster nodes is minimized. The first task involves making a prediction about the probability of reusing the same context based on its parameters (especially context lifetime), behavior patterns of consumers and other relevant historical data. The second task involves background analysis of data access patterns, understanding the data affinity principles and constructing recommendations how to reallocate data between nodes based on this analysis.

**Proactive Raw Data Retrieval (PRDR)** module is responsible for initiating the data retrieval from remote context producer before an incoming query requested it. This strategy can significantly reduce the time of serving the query. At the same time, retrieving data that will not be requested is not cost efficient. For that reason, the PRDR module relies on predictive algorithms to keep the optimal level of performance and cost.

Potentially, the number of recommenders can increase to cover other dimensions of efficiency. We present a more detailed discussion of our motivation to research the efficiency and adaptability of the CSMS in the next section. We also discuss the CSMS efficiency objectives along with possible approaches for increasing the level of efficiency.

### 3.6.5. Efficiency and adaptability of data management in CSMS

As we have shown in previous sections, the CoaaS platform and its storage subsystem need a new architecture and cannot completely repeat existing works. Efficiency is very important for CoaaS, both in terms of fast query processing and reducing the cost of running the system.

For that reason we have performed a review of existing methods in data management systems that can be potentially valuable for the CoaaS storage subsystem efficiency tuning modules.

Even in a well-established world of relational databases, finding an optimal way for storing, indexing and retrieving data is still not an easy task for experienced developers and administrators. That is the reason why most advanced DBMS gather statistics about usage of data and resources (for example, PostgreSQL uses pg_stat_statements module for this task). This statistics is uses by integrated tools that help developers to optimize indexing strategies, queries and get the best performance out of the available resources.

When we move to more complicated hybrid systems with a much wider variety of use cases, managing the optimal structure manually becomes an impossible task. Moreover, that is particularly the case for NoSQL datastores where redundancy is used to reduce the time and cost of joins for the price of disk space and consistency.

The optimization process can pursue several, often mutually conflicting, objectives: fast data ingestion, fast query response, low consumption of main memory and disk space, low consumption of processing power, network bandwidth and other resources. We can roughly divide these objectives into two categories: (i) Quality of Service (QoS) and resource optimization. Main criteria for the decision process are
the available amount of computation, storage, and network resources, ability to scale these resources elastically and the cost considerations and QoS requirements.

There is a broad range of research efforts in the area of Quality of Service (QoS) and resource optimization covering such topics as:

- Saving and reusing results of completed computations;
- In-memory caching;
- Reducing big data;
- Proactive retrieval of raw data;
- Pre-computation of results;
- Adaptive indexing;
- Elastic cluster sizing;
- Dynamic choice of storage format and data placement [58] [59];
- Data collocation.

These areas rely on sophisticated statistical, probabilistic and machine learning techniques. We believe many of the developed approaches can be useful for the CoaaS middleware. A discussion of the mentioned aims and research efforts is provided in the next section.

**Context Storage Management System efficiency and adaptability objectives**

We have figured out four main directions that need to be considered for the efficient functioning of the CoaaS storage system. These aspects namely are (i) fast data ingestion, (ii) fast query response, (iii) low consumption of main memory, disk space, and computing power and (iv) low consumption of network bandwidth.

**Fast data ingestion**

As it was already discussed in section 2, the volume of data being ingested by a CoaaS platform can potentially reach petabytes. A bigger problem is not even the volume itself, but that this data arrives in the form of short messages coming at a high frequency. Each of this messages must be processed through the ingestion chain starting with data source authorization, then percolating the message through registered subscriptions and situation detection modules, persisting the data if needed and adding it to an index. Another problem is the organization of a queue for reducing the probability of lost messages because of processing components being overloaded at some points of time. As we move to the area of tens of thousands incoming messages per second, fast processing becomes a real technical problem which should be carefully thought of at the design stage. We have performed a study of IoT platforms ingestion in [60] and shown how one underperforming component can significantly slow down the whole system.

The techniques used to achieve higher ingestion rates include:

(i) Parallel processing of the incoming data streams,

(ii) Organizing the message queue for a sustainable interconnection of components during peak loads,

(iii) Batch/micro-batch ingestion and indexing.

**Fast query response**

The main function of every database system is serving queries. The speed of query executing is critical to most of the applications. Reports show that with every 100 ms extra delay Amazon loses 1% of sales and every 0.5 seconds extra delay drops Google’s traffic by 20% [61, 62].
That requirement will not be different for most of the IoT applications, excluding big analytical batch processing jobs. We can suggest that fast query processing must be considered as the most serious non-functional requirement for a platform.

There exists a variety of techniques for increasing the speed of query execution which include:

(i) in-memory data allocation,
(ii) NoSQL approaches (e.g. redundancy instead of joins)
(iii) result caching and reusing
(iv) advanced indexing,
(v) predictive pre-computation of results,
(vi) predictive retrieval of raw data.

**Low main memory, disk space, and computation power consumption**

First of all, we can state that the price for data storage has significantly dropped during the last decade. This low price for persistent memory allowed the NoSQL movement, in which redundancy is one of the most significant patterns, to succeed. At the same time, storage of big data can be expensive, and it is always preferable to cut such expenses if it is possible. Apart from the storage, processing of such volumes of data is also a challenging procedure. For example, we can imagine a full table scan on a petabyte-size table.

In many cases, it is possible not to keep all the data points. Many use cases allow using effective data compression or aggregation, even with some information getting lost during the point. A survey on big data reduction methods was performed by [63]. The main research directions in this area are the network theory, big data compression, redundancy elimination, data pre-processing and dimension reduction. Each of these areas has several branches. In the case of the data compression area, authors mention spatiotemporal, gZip, AST, compressed sensing, parallel compression, and sketching. For example, for Smart-City use cases connected with vehicular movement the Sketching [64] method can be used to compress the data.

A significant research was done in the area of elastic cluster sizing and resource allocation for reducing the cost of cloud services while satisfying the SLA [65], [66]. Dutreilh et al. propose to use reinforcement learning for efficient automated resource allocation in cloud systems [67].

Dynamic resource reallocation between HDD and SSD is presented in [68], [69], [70], [71], and [72].

**Low network bandwidth consumption**

A real deployment of a CoaaS platform in a data centre is based on a set of server nodes, which are connected by a high-performance network. External entities can have any type of connection to the network, including slow low power technologies like Low Power Wide Area Network (LPWAN) Even with the fastest network technologies available, latencies introduced by transferring the data through the network are still significant even for the internal data centre communication. In the case of communication with outside world, the cost of network and latencies increase significantly. Approaches for lowering the load on network infrastructure and decreasing the network latencies include (i) bringing the computations to the data [73], (ii) data collocation [74], (iii) caching and reusing data from remote providers in the middleware.

Next section provides a brief analysis of methods used for achieving the above-mentioned objectives.

### 5.3.2 Possible approaches to meet the efficiency objectives

In this section, we highlight some of the mentioned research areas, which we believe will be useful for the CoaaS storage subsystem.

The Context Storage Management System will have to contain a cost model and rely on it for enabling the possibility to act efficiently. While having a single purpose, the model will logically consist of two
Theoretical Framework for Context and Situation Awareness in IoT

main parts: (i) the cost model of external services and (ii) the cost model of internal resources. Here, we assume that the SLA is a part of a consumer profile or a part of the context request, and, correspondingly, is located higher than the cost model.

The first part of the cost model has to facilitate the description of the context provider properties, facilitating the work of algorithms responsible for the choice of the most suitable context provider or a set of context providers. These properties include the cost of the query itself, in case a context provider charges the client on a request basis, the legitimate possibility to reuse cached data for following queries, the probability of receiving new queries that will be able to reuse the cached context. As the result of a query execution can be derived from data retrieved from multiple sources, the problem can be viewed as a problem of service composition. Related work on service composition can be found in [75], [76].

Li et al. propose to use a probabilistic approach for modeling and analysing the cost of service composition together with its reliability [77]. For solving the problem of service composition in mobile environments, where the data sources can change their location, Wang proposed to use mobility prediction as one of the main patterns [78]. A survey on formal methods for service composition is presented in [79].

The second part of the cost model will be responsible for the description of the available hardware and software resources.

In general, the unified cost model will be used as a reference for all the algorithms responsible for tuning the performance of CSMS.

**Proactive data retrieval**

One of the main functions of the context storage subsystem is to serve as a context provider, which caches data coming from external data sources and provides the cached data during the query processing stage. This approach can significantly reduce the time of query processing, as the latencies that occur while sending requests to remote entities can be inadmissible. Simultaneously, proactive retrieval of the data that will not be used or that will become outdated before being used will only increase the cost for the CoaaS platform and energy consumption for data sources. Implementing the query-based data proactive data retrieval adaptation is the promising approach. For instance, PRESTO, a feedback-driven data management system for sensor networks, is described in [80]. The main aim of PRESTO is to reduce the number of queries to sensors by using a shared by sensors and central node prediction model. With this approach, the sensor has to push data to the system only in the case when the measurement is different from the prediction, and the central node can serve queries without querying remote sensors. Simultaneously, the system automatically adjusts the models based on query dynamics, especially to such parameters as error and precision tolerance.

In [81] the design of COUGAR system is explained. This system aims to serve queries over sensor data, but the difference with the traditional approach is in the possibility to issue queries on stored data and the data that needs to be retrieved.

Ives et al. describe the problem of adapting to source properties in integration systems [82]. The proposed Tukwila system uses previously gathered information about external data sources to optimize the query at runtime.

In the area of predictive queries based on objects location, a substantial work was performed by Hendawi et al. In [83] a system called Panda is described, which is capable of scaling up and support a large number of moving devices, as well as a large number of queries. Predictions of location trajectories are of high importance in most of IoT domains.

We believe, more research should be done in this area, as it is one of the keys to the right balance between the query performance and cost of retrieving data from remote sources in CoaaS platforms.

**Advanced indexing**

Indexing involves building an accessory data structure and maintaining data in this structure to facilitate fast access to the place where the full document (or row in case of a relational database) is
located. Maintaining the right indexing strategy is a hard task that is typically performed by the database administrator [84]. Lack of indexes causes full scanning of available data, which consumes a lot of time and resources. At the same time, maintaining indexes also has a cost, as every new added, deleted or modified piece of data needs to be added to (or removed from) all the corresponding index structures. An over-indexed datastore will be performing unacceptable in many cases, especially the ingestion process will suffer.

With semi-structured contextual information with not predefined and constantly changing queries and workloads maintaining the right balance manually becomes impractical. There is a variety of works discussing and proposing techniques that will allow building indexes based on background analysis of workload. First of all, there is a need to understand the characteristics of the workload. The research in automated workload characterization area that was conducted as part of the Cloud-TM project is presented in [85]. Authors stress that effective mechanisms for self-tuning are essential for a cloud system that has to deal with the changing workload. They present a component called Workload Analyser that is responsible for gathering statistical data from cluster nodes, producing workload profiles and generating characteristics of current and predicted needs of the deployed applications.

In [57] authors divide approaches to indexing into four classes that are graphically presented in Figure 24. These approaches are: (i) no indexes with full table scanning, (ii) the traditional offline approach, (iii) online tuning and (iv) adaptive indexing, when the index is built as a side effect of the query execution.

![Figure 24 - Four approaches to indexing with regards to query processing](image)

Moreover, authors propose a methodology for benchmarking the effect of adaptive indexing on the performance of the system.

Other approaches to the problem are soft indexes [86] and online tuning [87], which involve a monitoring phase and an index building phase. These approaches suffer underperforming during the initial learning phase, but still, have the potential for being used in the CoaaS storage subsystem.

**Caching**

Caching the results of query execution in memory (or even on disk) can sufficiently increase the system performance by reusing these results instead of performing the whole query once again. Brin and Page in their paper [88] indicate caching of results as one of the most efficient approaches for increasing the performance of a web search engine. As the CoaaS platform has significant similarities with search engines, the result caching approach must be considered as the part of work to be done during the research and development of a CoaaS storage subsystem.

Wang and Jia conducted a survey of caching schemes for the Internet applications [89]. They raise valuable questions about the organisation of a caching system, list a number of requirements, such as robustness, scalability, adaptivity, stability and load balancing. Authors also state that a key aspect of the effectiveness of a caching system, which is the hit rate, is the algorithm that manages placement and replacement of documents. Authors divide caching algorithms in three main groups: traditional replacement strategies, key-based replacement strategies, and cost based replacement strategies. Traditional strategies include such algorithms as Least Recently Used (LRU), Least Frequently used (LFU), and Pitkow/Recker. Key-based replacement strategies include LRU-MIN, LRU-Threshold, Hyper-G, and Lowest Latency First (LLF).
Cost-based replacement strategies are more sophisticated and include such algorithms as GD-Size, Hybrid, Least Normalized Cost Replacement (LCN-R), Bolot/Hoshka, Size-Adjusted LRU (SLRU), server-assisted scheme, etc.

The drawback of the result-caching approach is the obsolescence of results as the incoming data that was ingested after the resulting set was generated is not taken into account. Keeping the cache up to date requires resources and effort. The result caching approach has a limited usage scope. Still, it can be applied for many applications where the number of served clients and query response time are much more valuable than getting an entirely real-time result.

### Pre-computation of results

Gathering statistics of queries and understanding patterns in them is a key to another efficient technique for achieving high performance – the pre-computation of results. For example, consider a user, who drives every morning from point A to point B at 7:00 and queries the fastest route. We can predict that pre-computing this route at 6:59 will be efficient, as the user will get a result in an order of milliseconds, resulting in higher level of user satisfaction. The patterns can be much more sophisticated.

In [90] the basic principles of SensorDB are discussed. Authors state that while having thousands of data points coming from sensors, the individual results are usually not as important as patterns and correlations. For this reason, SensorDB pre-calculates certain features of the data stream using a set of pre-defined aggregation windows. This technique helps to increase scalability, query response time and overall performance of the system.

The same approach is presented in [91]. Authors of SmartFarmNet platform use the micro summarisation approach to calculating statistical features of a data stream for aggregation windows. SmartFarmNet also makes a decision about the placement of results based on data access patterns. For example, frequently accessed data is stored in the in-memory database. Related work in the area on location predictions can also be found in [92].

Lempel and Moran [93] describe the technique called probability driven cache (PDC) for predictive caching and prefetching of query results. The work was done for a search engine, but still can be valuable for the CoaaS platform. Their approach called “prefetching” predicts the next query that is likely to be issued by a user during the search session. After predicting the query, the result is pre-computed and put into the cache. In case the prediction is found to be true, the user gets the result without further delays. The difference with the CoaaS platform is in the location of the raw data. In [93] the data is already located inside the storage system of a search engine. In the case of CoaaS, the raw data can be partly stored internally, but also a need to retrieve it from remote sources can arise. The decision has to rely both on the prediction and the CoaaS storage cost model that was described in the corresponding section.

The drawback of the pre-computation approach is its unreliability in the case of serious changes in query patterns. Another drawback is a need for storing or caching result data, which adds redundancy to the system. Despite the drawbacks, the approach is very promising for the CoaaS storage subsystem.

### In-memory data allocation and processing

Discussion of in-memory databases and In-memory Data Grids (IMDGs) is very active nowadays. It is more technological, rather than a methodological way to reach efficiency and high performance of a system. IMDGs receive much attention in the IoT community [53]. In a survey of big data management and processing technologies [94] Zhang et al. state we are facing a revolution in system design. Access to memory is hundreds of times faster in comparison with spinning disk or SSD technologies. This makes IMDGs leaders in performance as compared with all other competitors. At the same time, a cost
of a system that works with big data can become unreasonably high, due to a need of deploying hundreds or even thousands of server nodes to achieve the needed volume of memory. Another issue is robustness of such a system.

One of the problems that will be needed to be solved for complex context management scenarios is the collocation of data on cluster nodes based on affinity principles. Di Sanzo et. al. [95] propose machine learning techniques for tuning the performance of IMDGs.

The research trend in this very fresh area needs to be monitored, and IMDGs (e.g. Apache Ignite, Hazelcast) should be evaluated as a possible technological platform for CoaaS storage modules that need the highest possible performance despite the cost.

### 3.7. Storage technologies

CoaaS architecture depends on several types of data: structured, semi-structured and unstructured. Moreover, the architecture relies on multiple knowledge frameworks, needs a caching subsystem for achieving high performance, and the scalability requirement also should be considered. Above mentioned requirements make it hard to build the whole system based on one technology. For this reason, we propose to use the polyglot persistence approach [96]. With this approach systems no longer try to accomplish all tasks using one data storage, but rather use different technologies to store data where each technology provides certain capabilities.

**Document store (DS)**

DS is the core part of the CSMS. It stores all the incoming data and derived context that can be reused for serving future context requests. This load puts especially strict requirement towards reliability and scalability of DS, as this part must always be available. As it is considered that the size of DS will be enormous, proper indexing is required to keep the data retrieval inside the SLA range.

**Logs and history (LH)**

Logs and history data are essential for system operability. By logs, we mean all the information collected during system operation, which include context requests, plans developed by Storage Query Execution Planner, results that were sent back, and all the metadata that accompanies serving a query. Not even mentioning monitoring and debugging purposes, which are outside the scope of current research, logs are vital for all the recommenders that were described in the previous section. Many components of the system produce an enormous amounts of logs and mining them brings serious complexity. We propose storing logs on HDFS to enable the use of MapReduce approach. Planners and recommenders must be able to construct analytical queries and adjust their behavior based on the results.

**Knowledge**

The process of deriving context in real environments deeply depends on knowledge, which represents data about entities, their relations, profiles and many other parameters. The knowledge base forms a semi-static base layer for many decision-making algorithms. This knowledge can come from two main sources: (i) it can originate from an existing dataset, or (ii) it can be derived from processing big data, event streams and logs using data mining, machine learning, and other available techniques. For example, in Smart City scenarios, datasets that represent the city road map and everything related to spatial data analysis are essential data sources, as they provide a set of constraints, on top of which real-time situations can be defined.

The knowledge block of CoaaS storage can contain a very broad stack of datasets and toolkits for working with them. The CoaaS platform will have to provide APIs for enabling the pluggable introduction of domain-specific knowledge bases into the system.
In-memory cache

Operating in near real-time is one of the requirements for the CoaaS middleware. For this reason, we propose to integrate an in-memory caching layer for storing context requests and answers. This layer can significantly reduce the query processing time in the case when a similar query was already processed recently. Most significant tasks in cache layer management are correct choice of the time to live (TTL) parameter and cache invalidation in the case of a situation change.

In summary to this section, building a CoaaS middleware platform involves significant research and development efforts in such areas as interoperability standards, specialized context definition and query language, context query processing technologies, etc. In our research, we are focusing on the Context Storage Management System as a core component of CoaaS. We have shown the need for including the storage subsystem into the platform and defined main elements of this subsystem. It is already clear that the context storage management is a sophisticated system and much effort will be needed to produce a stable, performance-oriented and fully operational prototype.

3.8. Experimental and testing environment for CSMS

Figure 25 illustrates the Context Storage Management System (CSMS) implementation environment.

In general, Context Storage Management System consists of two main layers: (i) Storage layer and (ii) Processing layer. Other important issues that have to be considered are the development environment, vocabularies and data representation standards, and the organization of collaboration. The implementation environment of the CSMS is presented in Figure 1.

Both, the storage and the processing layers are developed with an aim for operating in the multi-node scalable environment to facilitate efficient processing of big data.

We have deployed virtual machines (VMs) in CSIRO Data 61 Bowen cloud for hosting storage and processing layers. Virtual machines are physically located in Melbourne, Australia.

![Figure 25. CSMS implementation environment](image-url)
Characteristics of the machines used in our experiments are provided below:

1. Processor:
   - Architecture: x86_64
   - Model name: Intel(R) Xeon(R) CPU E5-4640 0 @ 2.40GHz
   - Number of cores: 8
   - CPU MHz: 2400.000
   - BogoMIPS: 4800.00
   - Hypervisor vendor: VMware
   - Virtualization type: full
   - L1d cache: 32K
   - L1i cache: 32K
   - L2 cache: 256K
   - L3 cache: 20480K
2. RAM: 64 Gb
3. DISK: 2 Tb + additional network area storage
4. OS: Debian Linux 7.0 Wheezy

Storage layer
As it was discussed in the CSMS overview section, the architecture follows the polyglot persistence approach. CSMS will have to deal with structured, semi-structured, and unstructured data, and operational requirements are different across use-cases.

Respectively, CSMS uses several different data stores for being able to fulfil the functional and non-functional requirements.

We use PostgreSQL 9.6 for storing relational data. PostgreSQL 9.6 provides the best functionality in the class of open-source relational databases. It can serve in high-availability mode. However, organizing the high-performance processing of big data, where ACID requirements are not so strict in a relational database is a challenging process that can be avoided by using NoSQL approaches.

The pilot version of CSMS will use MongoDB 3.4 as the main place for storing vast amounts of data and context. However, we also have plans to evaluate Elasticsearch 5.4.0 as an alternative for storing, indexing and retrieving of JSON and JSON-LD data.

High-performance CSMS solution also requires an in-memory caching layer for an ultra-fast serving of queries which are issued on a regular basis or can be predicted. We use Redis 3.2.9 as an in-memory data store for our caching layer.

Functioning of all the storage and processing components, including the SQEM, will produce enormous amounts of logs, traces and other semi-structured and unstructured data that can bring value after proper analysis. We use HDFS as a storage layer for such kind of data.

Processing layer
Principal components of the processing layer that are being developed are the SQEM, Situation Monitoring module, and Recommenders. We use Java EE and Glassfish 4.1.2 as the main development technology and application serving for SQEM. Recommenders will need to analyze huge amount of log data
stored in HDFS. For this, we run recommenders over the Spark framework. The Situation Monitoring module requires efficient parallel processing of incoming data streams. For that reason, we adopt the Spark streaming framework.

**Development**

Development and productivity tools are crucial for modern software development cycle. By now, we use the following stack:

- NetBeans for Java development
- Robomongo for MongoDB
- PgAdmin3 for PostgreSQL
- Sense for Elasticsearch
- Redis Desktop Manager for Redis
- Hortonworks tools for Spark

We have chosen the most advance of available open-source or free software. However, the choice of the development environment is a matter of taste and in many cases does not influence interoperability.

**Vocabularies**

Vocabularies and standardization are of paramount importance in the project. Any use-case will have to rely on discussed and developed data structures. We have chosen to store data and Context in CoaaS in a single format. We assume, in case if new context consumers or providers which do not support our standard will join the system, we can develop converters, which will stand on the border of the system (on our or their side) and perform translation of formats.

We agreed on using the Schema.org vocabularies as a base. However, these vocabularies are not enough for modelling all possible use case. Consequently, we compose our vocabularies on top of schema.org. For example, the smart parking and electric charging use-cases are using Mobivox. ITMO University (St. Petersburg), in turn, is developing a vocabulary for the Smart Waste Management use-case. We assume that the number of vocabularies will grow with the number of use-cases. Adapting these vocabularies to internal data structures of the CSMS will have to be performed as these vocabularies are established.

**Collaboration**

We have chosen GitHub as a collaboration platform for the development of CSDM.
4. Conclusion

This deliverable “D4.5 Context-Aware Actions and Self-Adaptation Framework v1” has addressed the following objectives:

- Architecture of Context-as-a-Service (CoaaS) bloTope core component.
- Context query engine as means for entities to request context and respond to context queries.
- Context storage management system for handling potentially massive amounts of context related to large number of monitored objects and situations.

WP4 team will continue developing beyond the state-of-the-art context provisioning platform as a core component of the bloTope ecosystem.
5. References

1. Internet of things spending forecast to reach nearly $1.3 Trillion in 2019 led by widespread initiatives and outlays across Asia/pacific, (2015).
25. Paul Andlinger: Graph DBMS increased their popularity by 500% within the last 2 years, http://db-engines.com/en/blog_post/43.
49. Malim, G.: Looking for a Benchmarking Framework for IoT platforms,


