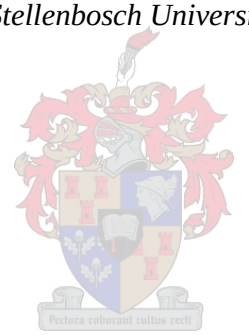
# A Design Framework for Aggregation in a System of Digital Twins

by
Carlo Human

*Dissertation presented for the degree of Doctor of Philosophy*
*in the Faculty of Engineering at*
*Stellenbosch University*




Supervisor: Prof Anton Herman Basson
Co-supervisor: Dr Karel Kruger

April 2022

# Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: April 2022

# Abstract

## A Design Framework for Aggregation in a System of Digital Twins

C. Human

*Department of Mechanical and Mechatronic Engineering*
*Stellenbosch University*
Dissertation: Ph.D. (Mechatronic Engineering)
April 2022

The digital twin (DT) concept has become a popular means of capturing and utilising data related to physical systems and has been applied in many domains. The data provided within DTs allow for the integration of services and models to improve understanding and decision-making related to the physical system. Through aggregation, multiple DTs can be combined to represent larger, more complex system, while maintaining the separation of concerns.

The design framework presented in this dissertation aims to enable systematic, effective decisions when designing a system of DTs to represent a complex physical system. In particular, this framework adopts hierarchical aggregation as one of its primary enablers and it considers the use of a services network, such as a service-oriented architecture, as well. The design framework is intended to be broadly applicable, by remaining vendor-neutral, and it enables traceability of design choices.

The approach starts with an analysis of physical system complexity to identify key needs related to managing complexity. A suitable requirements classification is then introduced to help translate the needs into requirements that the system of DTs should satisfy. Hierarchical aggregation is also introduced as a primary architectural approach to manage complexity. Hierarchical aggregation allows for the separation of concerns, computational load distribution, incremental development and modular software design. The design framework is arranged in six steps: 1) needs and constraints analysis, 2) physical system decomposition, 3) services allocation, 4) performance and quality considerations, 5) implementation considerations and 6) verification and validation.

The dissertation then introduces a general reference architecture that combines a system of DTs (which follows hierarchical aggregation principles) with a services network to allow for reliable and adaptable service provisioning. The design framework is then discussed in the context of the general reference architecture.

The design steps of the design framework are then moulded into six design patterns, which simplify the design process by focussing of key quality attributes. The quality attributes considered for the respective design patterns are performance efficiency, reliability, maintainability, compatibility, portability and security.

The use of the design framework and design patterns are then demonstrated and validated through three case studies, two high-level case studies and one detailed case study. The high-level case studies consider a water distribution system and a smart city, respectively. The detailed case study considers a heliostat field.

The dissertation concludes that the design framework, as well as the design patterns, enable a systematic approach to designing a system of DTs. The design framework can also be applied to numerous and varying domains, such as the case studies considered.

# Uittreksel

## ‘n Ontwerpsraamwerk vir Samevoeging in ‘n Sisteem van Digitale Tweelinge

C. Human

*Departement van Meganiese en Megatroniese Ingenieurswese*
*Universiteit Stellenbosch*
Proefskrif: Ph.D. (Megatroniese Ingenieurswese)
April 2022

Die digitale tweeling-konsep het 'n gewilde manier geword om data wat met fisiese stelsels verband hou, vas te lê en te gebruik en die konsep word in talle gevalle toegepas. Die data wat binne ‘n digitale tweeling verskaf word, bied die geleentheid vir die integrasie van dienste en modelle om begrip en besluitneming, met betrekking tot die fisiese stelsel, te verbeter. Deur verskeie digitale tweelinge saam te voeg kan groter, meer komplekse stelsel voorgestel word, terwyl die skeiding van belange gehandhaaf word.

Die ontwerpsraamwerk wat in hierdie proefskrif voorgestel word, beoog om sistematiese, effektiewe besluite moontlik te maak wanneer 'n stelsel van digitale tweelinge ontwerp moet word om 'n komplekse fisiese sisteem voor te stel. Hierdie raamwerk maak gebruik van hiërargiese samevoeging en dit oorweeg die gebruik van 'n dienstenetwerk, soos 'n diensgeoriënteerde argitektuur. Die ontwerpsraamwerk is bedoel om breed toepaslik te wees, deur verskaffer-neutraal te bly, en dit maak naspeurbaarheid van ontwerpkeuses moontlik.

Die benadering begin met 'n ontleding van fisiese stelselkompleksiteit om sleutelbehoeftes te identifiseer wat verband hou met die bestuur van kompleksiteit. 'n Geskikte vereistesklassifikasie word dan ingevoer om die sleutelbehoeftes te omskep in vereistes waaraan die stelsel van digitale tweelinge behoort te voldoen. Hiërargiese samevoeging word ook ingevoer as 'n primêre argitektoniese benadering om kompleksiteit te bestuur. Hiërargiese samevoeging maak voorsiening vir die skeiding van belange, berekeningsladingsverspreiding, inkrementele ontwikkeling en modulêre sagteware-ontwerp. Die ontwerpsraamwerk is georden in ses stappe: 1) behoeftes- en beperkingsontleding, 2) fisiese stelselverdeling, 3) dienstetoewysing, 4) prestasie- en kwaliteitsoorwegings, 5) implementeringsoorwegings en 6) verifikasie en validering.

Die proefskrif stel dan 'n algemene verwysingsargitektuur voor wat 'n stelsel van digitale tweelinge (wat hiërargiese samevoegingsbeginsels volg) kombineer met 'n

dienstenetwerk om voorsiening te maak vir betroubare en aanpasbare diensverskaffing. Die ontwerpsraamwerk word dan in die konteks van die algemene verwysingsargitektuur bespreek.

Die ontwerpstappe van die ontwerpsraamwerk word dan in ses ontwerppatrone gevorm, wat die ontwerpsproses vereenvoudig deur te fokus op sleutelkwaliteiteienskappe. Die kwaliteitseienskappe wat vir die onderskeie ontwerppatrone oorweeg word, is prestasiedoeltreffendheid, betroubaarheid, instandhoubaarheid, verenigbaarheid, oordraagbaarheid en sekuriteit.

Die gebruik van die ontwerpsraamwerk en ontwerppatrone word dan deur drie gevallestudies, twee hoëvlak gevallestudies en een in diepte gevallestudie, gedemonstreer en gevalideer. Die hoëvlak gevallestudies oorweeg onderskeidelik 'n waterverspreidingstelsel en 'n slim stad. Die gedetailleerde gevallestudie beskou 'n heliostaatveld.

Die proefskrif kom tot die gevolgtrekking dat die ontwerpsraamwerk, sowel as die ontwerppatrone, 'n sistematiese benadering tot die ontwerp van 'n stelsel van digitale tweelinge moontlik maak. Die ontwerpsraamwerk kan ook op talle en variërende gevalle toegepas word, soos die gevallestudies wat oorweeg is.

# Acknowledgements

I would like to thank my supervisors, Prof AH Basson and Dr K Kruger, for their tremendous guidance, kindness, and patience. Without them, I would not have made it this far. I would also like to thank the MAD Research Group for all the brainstorms and coffee breaks.

Thank you to Dr W Smit with STERG for providing funding for this PhD as part of the larger PreMa project.

Thank you to my wife, Kelly, and my parents, Carl and Sabine, for all the love and support. It has been much appreciated and I hope I have made you proud.

Finally, may this thesis bring glory to Jesus, the Father and the Holy Spirit.

# Table of contents

# List of figures

# List of tables

# List of abbreviations

AAS – Asset Administration Shell

ACID - Atomic, Consistent, Isolated, Durable

AES – Advanced Encryption Standard

AMQP – Advanced Message Queuing Protocol

API – Application Programming Interface

AWS – Amazon Web Services

BSI – British Standards Institute

CA – Certificate Authority

CCU – Cluster Controller Unit

CDBB – Centre for Digital Built Britain

Cloud IAM – Cloud Identity and Access Management

CoAP - Constrained Application Protocol

CICD – Continuous Integration Continuous Deployment

CPS – Cyber Physical Systems

CPU – Central Processing Unit

CRS – Central Receiver System

CS – Configuration Server

CSP – Concentrated Solar Power

CUI – Central User Interface

DNI – Direct Normal Irradiance

DS – Directory Service

DT – Digital Twin

DTA – Digital Twin Aggregate

DTE – Digital Twin Environment

DTI – Digital Twin Instance

DTM – Digital Twin Monitor

DTP – Digital Twin Prototype

ELT – Extract, Load, Transform

ESB – Enterprise Service Bus

ETL - Extract, Transform, Load

FCU – Field Controller Unit

FR – Functional Requirements

G - Gateway

GCP – Google Cloud Platform

GPRS – General Packet Radio Service

GPS – Global Positioning System

GPU – Graphical Processing Unit

GUI – Graphical User Interface

HDF – Hierarchical Data Format

HTTP - Hyper Text Transport Protocol

HTTPS - Hyper Text Transport Protocol Secure

ICT – Information Communication Technology

IDL – Interface Definition Language

IETF – Internet Engineering Task Force

I/O – Input/Output

IoT – Internet of Things

ISO – International Organisation for Standards

IT – Information Technology

IPv4 – Internet Protocol version 4

JSON – JavaScript Object Notation

JWT – JSON Web Token

LCU – Local Controller Unit

Mbps – Megabits per second

MCL – MAYA communication Layer

MQTT – Message Queuing Telemetry Transport

MSF – MAYA Simulation Framework

MSI – MAYA Support Infrastructure

MVCC – Multi-Version Concurrency Control

NDT – National Digital Twin

NFR – Non-Functional Requirement

NoSQL – Not only SQL

OPC UA – Open Platforms Communications Unified Architecture

OS – Orchestration Service

PLC – Programable Logic Controller

PLM – Product Lifecycle Management

PV – Photovoltaic

QoL – Quality of Life

QoS – Quality of Service

RAM – Random-Access Memory

RAMI4.0 – Reference Architecture Model Industry 4.0

REST – Representational State Transfer

RF – Radio Frequency

RPC - Remote Procedure Call

RTU – Remote Terminal Unit

SCADA – Supervisory Control and Data Acquisition

SDK – Software Development Kit

SLADT – Six Layer Architecture for Digital Twins

SLADTA – Six Layer Architecture for Digital Twins with Aggregation

SOA – Service-Oriented Architecture

SQL – Structured Query Language

SS – Security Service

STERG – Solar Thermal Energy Research Group

TLS – Transport Layer Security

VM – Virtual Machine

VPN – Virtual Private Network

WAN – Wide Area Network

WDS – Water Distribution System

WSN – Wireless Sensor Network

XML – Extensible Markup Language

# 1 Introduction

## 1.1 Background

This dissertation is in the context of digital twins (DTs), a concept that was first introduced by Michael Grieves in 2002 as the conceptual ideal for product lifecycle management (Grieves & Vickers, 2017). The original premise was to create a virtual representation of a physical product, where the virtual representation could mirror the physical product throughout its lifecycle. Since then, the DT concept has evolved and has become a significant enabler of Industry 4.0 initiatives (Durão, Haag, Anderl, *et al.*, 2018). The industry 4.0 initiatives include connectedness and intelligence to allow for decentralised and adaptable production environments. Furthermore, the DT concept has been adopted in domains outside of production to achieve the same principles of connectedness and intelligence as part of larger digitisation movement.

The DT concept has become a popular means of capturing and utilising data related to physical systems and has been applied in many domains, such as manufacturing (Bao, Guo, Li, *et al.*, 2018; Redelinghuys, Basson & Kruger, 2020), smart city design (CDBB, 2018; Pan, Shi & Jiang, 2020), water treatment facilities (Therrien, Nicolaï & Vanrolleghem, 2020), the maritime domain (Bekker, 2018), wind turbines (Pargmann, Euhausen & Faber, 2018), aerospace (Glaessgen & Stargel, 2012), healthcare (Lutze, 2019), etc. The data provided within DTs allow for the integration of services and models to improve understanding and decision-making related to the physical system (Kuhn, Schnicke & Oliveira Antonino, 2020; Longo, Nicoletti & Padovano, 2019).

The dissertation title refers to a system of DTs. This concept is particularly relevant considering complex systems. A complex system is a system that consists of a large network of components which give rise to complex collective behaviour, sophisticated information processing and adaptation via learning or evolution (Mitchell, 2009). System complexity can be described as a measure of how difficult it is to understand the behaviour of a system and how difficult it is to predict the consequences of changing the system (SEBoK Editorial Board, 2021). Considering that digital twins are recognised as a concept that could help understand and manage the data related to complex system, there is a need to implement digital twins for complex systems and systems-of-systems.

However, to feasibly apply the DT concept to complex systems, the aggregation of DTs is necessary, which leads to a system of DTs. Minerva, Lee & Crespi (2020) mention that the simulation and prediction of behaviour of complex systems, based on an aggregated set of DTs, is a unique benefit of the DT concept.

Aggregation of DTs is a strategy to enhance separation of concerns, reconfigurability, scalability, and it is therefore attractive for reflecting complex physical systems (Ciavotta, Bettoni & Izzo, 2018; Lutze, 2019; Redelinghuys, 2019). It is possible to view a system from different perspectives, by aggregating different DTs and data features, while also allowing for different levels of detail for each perspective (Borangiu, Oltean, Raileanu, *et al.*, 2019; Ciavotta *et al.*, 2018; Villalonga, Negri, Fumagalli, *et al.*, 2020). This ability to accommodate different perspectives makes it easier to accommodate multiple stakeholders and it makes the data more comprehensible (Lutters & Damgrave, 2019; Lutze, 2019).

Therefore, this dissertation considers a framework to aid in the design of a DT aggregation hierarchy. The framework considers complex systems and provides general principles to help design a DT aggregation architecture to manage the complexity of the systems.

## 1.2 Objective and contribution

The objective of this dissertation is to develop a design framework to guide the detailed design of a DT aggregation architecture, or a system of DTs, to reflect complex systems.

The framework makes use of hierarchical aggregation design principles. In particular, the framework provides methods and principles that guide a user through the design process. The framework also aims to make the user aware of trade-offs that may form part of the design approach and provides aggregation and implementation alternatives to manage the trade-offs.

The framework is intended to be independent of the domain of application of the DTs, and thus it prescribes general design principles and best practices according to prioritised design requirements. Using the framework, the user should be able to design a detailed architecture for a DT aggregation hierarchy that reflects a complex system within a given domain. The resulting detailed architecture should provide a list of applicable DTs, the functionality of those DTs and how they relate to each other. Furthermore, the framework also supports implementation decisions related to key aspects to the architecture.

It is not feasible within the scope of a PhD dissertation to comprehensively evaluate the decision framework. However, it is applied to different case studies in different domains as a preliminary evaluation and to demonstrate the efficacy of the framework.

This dissertation contributes to the body of knowledge on DT design through the novel design framework. Most research on DTs involve the development of an architecture to satisfy the needs of a particular domain or use case. Some

architectures are defined to be more generally applicable. However, being an architecture, they explain the composition of the DT, but not how to approach the design of the components and their interactions. Little research has been done on methods and principles for the design of DTs. Furthermore, the design frameworks that do exist are still quite limited because they consider the design of a single DT or they only consider particular conceptual aspects of the DTs design.

The design framework presented here is intended to contribute to the digital twin body of knowledge in the following ways:

- The framework considers the design of a system of DTs and how those DTs can interact to represent a complex system.
- The framework enables a systematic approach to decision making during the design of the system of DTs, starting with the fundamental needs and associated requirements.
- The framework enables traceability from user defined needs and derived needs to architectural and implementation choices.
- The framework makes provision for key sources of complexity, such as multistakeholder environments and distributed computing environments.
- The framework considers the integration of DTs with a service-oriented architecture.
- The framework is unique in its provisioning of design patterns according to key quality requirements.

## 1.3 Motivation

Although much has been published about DTs, the use of aggregation with DTs, is fairly recent and poorly defined. The earliest papers in Scopus with "digital twin" and "aggregation" in their title, keywords or abstract, were published in 2018 (Ciavotta, Bettoni & Izzo, 2018; Lutters, 2018). Therefore, there is a need for research with respect to the aggregation of DTs since it is an enabler for the effective digitisation of complex systems.

For example, the Centre for Digital Built Britain (CDBB) is researching the possibilities of creating a digital twin of Britain, named the National Digital Twin (CDBB, 2018; Lamb, 2019). This is a highly diverse, complex and large system with numerous levels of subsystems. CDBB (2018) states: “The vision for the national digital twin (NDT) is not that it will be a huge singular digital twin of the entire built environment. Rather, it is envisaged to consist of ‘federations’ of digital twins joined together via securely shared data.”

The complexities related to implementing DTs and systems of DTs are not yet fully understood. A survey on DTs by Minerva, Lee & Crespi (2020) raised concerns regarding the scalability and interoperability of DTs and the viability of the concept when multiple stakeholders are present. Villalonga *et al.* (2021) also mention some shortcomings of research with respect to distributed DTs, including 1) a lack of well-defined frameworks to combine DTs, 2) the limitation of methods for aggregating DTs and 3) the poor use of the gathered data. These concerns give rise to research questions such as:

- What complexities arise when implementing a system of DTs, particularly with regard to making the system of DTs scalable, interoperable and suitable for multi-stakeholder environments?
- What challenges are addressed by using aggregation and what additional challenges arise when aggregating DTs?
- What are some of the key aspects to consider when aggregating DTs?
- How can DTs and aggregations of DTs help to servitise data?

The design framework is intended to help identify and navigate some of the complexities of implementing DTs, including aspects related to scalability, interoperability and multi-stakeholder environments. Furthermore, the framework allows for the distributed implementation of DTs and provides methods and principles to help aggregate the DTs. Finally, the framework also addresses the issue of data endpoints, where the framework incorporates literature regarding the servitisation of data within and through DTs.

The novel contributions to the growing body of knowledge about digital twins, indicated in the previous section, are further motivation for the dissertation. These contributions are intended to improve the feasibility of developing a digital twin system for complex systems and thereby make them more applicable to industry.

## 1.4 Methodology and dissertation overview

The primary deliverable of this dissertation is a design framework to help reason about the design of a DT aggregation hierarchy. An overview of the design framework is provided in Chapter 7.

To develop the design framework, software and systems engineering steps and principles as presented by SEBoK Editorial Board (2021) and Bourque & Fairley (2014) were considered. These general systems design guides include steps such as performing stakeholders' needs analysis, requirements engineering, identifying design constraints, identifying design trade-offs, etc. Furthermore, these sources suggest principles such as iteratively designing and testing the design and ensuring

traceability for good change management. Based on these design steps and principles, the methodology detailed below discusses how the design framework was developed.

A review of the relevant literature is presented in Chapters 2, 3 and 4. Chapter 2 presents literature related to DTs and particularly literature related to the aggregation and design of DTs, as well as the integration of DTs and service-oriented architectures (SOAs). Chapter 3 is devoted to the identification and classification of general needs related to the complexities of DT system design and implementation. The complexity needs were identified from digital twin literature, as well as literature concerning similar software systems and technologies. In particular, the related domains that were also investigated are: wireless sensor network (WSNs), Internet of Things (IoT), edge, fog and cloud computing, big data pipelines and service-oriented and microservices architectures.

The complexity needs were then analysed in the context of digital twins to determine if they were applicable and if so, how they can be translated to a general set of requirements to help manage the complexity. However, to accomplish this, a requirements framework for digital twins was necessary. Therefore, Chapter 4 discusses the set of requirements considered for this dissertation and how they relate to the complexity needs identified in Chapter 3.

With the above inputs, the framework was developed iteratively. The iterative process started by considering fundamental concepts of complexity management in systems engineering and software engineering. The final set of fundamental principles are presented in Chapter 5. The fundamental principles were then related to architectural and implementations solutions within a given context to determine how the principles can be embodied. Chapter 6 presents a reference architecture that is intended to embody the aforementioned principles and it provides context when applying the design framework.

Chapter 7 introduces the design framework. The chapter starts with the objectives of the design framework and an overview is provided before each step of the design framework is discussed. The design steps are discussed with a running example to clarify the concepts.

Chapters 8, 9 and 10 each consider one of the design framework's steps in more detail. Chapters 8 considers the service allocation step, Chapter 9 considers performance and quality aspects of the architecture and Chapter 10 expands on Chapter 9 by providing guidelines with regards to key reoccurring implementation choices. Each chapter also considers how the design choices can change depending on the context.

The evolving framework was regularly tested against various digital twin aggregation contexts that were being considered in the research group. These trial

applications helped to clarify the important distinctions (such as architectural and implementation aspects) and the decision sequences that offers a systematic approach with limited iteration overall, but intensive iteration inside the few major steps. The results of the above development are presented in the dissertation and not the process of the framework's development.

In Chapter 11, the design framework is moulded to six design patterns. The design patterns are intended to simplify the entire design process by highlighting the key needs, architectural choices and implementation choices related to the requirement that the design pattern is named after.

In Chapter 12, the design framework is applied to two high-level case studies: a water distribution system and a smart city. The purpose of the high-level case studies is to demonstrate the systematic approach of the design framework in two different case studies. The case studies each present unique challenges and considerations and by applying the design framework to each of these cases, the general applicability of the design framework is also demonstrated.

A water distribution network is a critical piece of infrastructure in any city and thus reliability is important. Water distribution networks also have a relatively long operational lifetime and thus associated DTs will need to be maintained for the duration of the water distribution network's lifecycle. The water distribution system also presents some noteworthy characteristics, such as the continuous nature of the piping network and the large geographical distribution.

The smart city case study presents a are large systems with numerous heterogeneous subsystems and a substantial amount of heterogeneous data, making interoperability a major concern. Smart cities also have many different stakeholders, where secure data sharing is a high priority. Furthermore, smart cities have mobile entities, such as public transport vehicles, which present some unique challenges.

In Chapter 13, the design framework is applied in a more detailed case study. The purpose of this case study is to further demonstrate the systematic approach and generality of the design framework. This case study further validates the design framework's approach by discussing how the architecture was implemented and tested. The implemented architecture's scalability and reconfigurability were tested to validate the ability of the architecture in key areas of concern.

Chapter 14 presents an evaluation of the design framework, where overall aspects of the design framework are discussed, as well as the individual design steps. The dissertation concludes in Chapter 15, where the key findings are summarised.

# 2 Literature review

This section provides a short overview of literature applicable to DTs. Section 2.1 provides a definition and qualifying criteria for DTs. Section 2.2 reviews literature related to the aggregation of DTs, whereas Section 2.3 reviews literature related to the combination of DTs with SOAs. Finally, Section 2.4 considers existing DT design frameworks. Furthermore, Chapters 3 and 4 also form part of the literature review, where each of these chapters considers a particular subject related to the design framework. Chapter 5 also makes extensive use of existing literature but with the intention of formulating new concepts.

## 2.1 Qualifying criteria for digital twins

DTs have been considered in a multitude of application domains with a multitude of perspectives (as discussed in Section 1.1). The multitude of perspectives have resulted in many definitions of digital twins. Drawing from a number of sources (Grieves & Vickers, 2016; Kritzinger, Traar, Henjes, *et al.*, 2018; Minerva *et al.*, 2020; Moyne, Qamsane, Balta, *et al.*, 2020; Taylor, Human, Kruger, *et al.*, 2020; VanDerHorn & Mahadevan, 2021), a DT is defined here as the virtual representation of a real-world entity (the physical twin), including that:

- The representation is maintained in soft real-time (or near real-time) through data flows from the physical twin (sometimes referred to as synchronisation between the physical and digital twins) and/or from associated models. Soft real-time requires the virtual representation to be updated within a given time-period. However, occasionally missing the deadline is not detrimental to the decision-making performance (as opposed to hard real-time where failure to meet a deadline can cause critical failure). Alternatively, near real-time only requires the virtual representation to be updated as soon as feasibly possible. The required update frequency will most likely be dictated by the decision-making frequency or the potential rate of physical system change.
- Past representations are maintained (historical data storage).
- The representation is constrained to features of interest, which may differ for the soft-real time and the historical representations. Furthermore, the required fidelity of the model(s) within the digital twin is dependent on the case. High-fidelity models are not always feasible (since not all states, inputs and outputs can be measured) or even required (depending on the service requirement). Aspects related to DT fidelity are further discussed in Section 9.2.1.
- The DT enables data-led decision making within and/or beyond the digital twin. The decisions affect the physical twin and/or other systems. Furthermore, data-led decision making requires that decisions be made based

on data of the physical reality (captured and generated in the virtual representation), but it does not exclude the use of human observations, expert experience and intuition, etc.

- Subject-matter-expertise (also known as domain expertise) of the physical twin is embedded within the above aspects of the digital twin.
- The above aspects are tailored to one or more life cycle stages of the physical twin (which can include design and planning, construction or manufacturing, ongoing operations, support during the operational phase and/or disposal).

Further, a digital twin may include:

- Application-specific functionality, such as simulations or predictions of future representations.
- Bi-directional communication between the DT and the physical twin to affect the behaviour of the physical twin in response to decisions made autonomously by the digital twin and/or decisions made outside the DT. Some sources, such as Kritzinger *et al.* (2018), insist that a DT must have automatic bi-directional communications. However, other sources, such as VanDerHorn & Mahadevan (2021) and Minerva *et al.* (2020), do not require bi-directional communication, arguing that requiring bi-directional communication is too restricting, particularly when considering domains other than manufacturing.
- Facilities for integration with digital systems and services outside the DT, such as providing data-related and/or modelling-related services as part of a larger service-oriented architecture.

Therefore, regardless of the architecture used to develop a DT, it must be able to continually (based on Ciavotta, Maso, Rovere, *et al.*, 2020; Harper, Malakuti & Ganz, 2019; Minerva *et al.*, 2020; Redelinghuys, Basson, *et al.*, 2020; Redelinghuys, Kruger & Basson, 2020; Therrien *et al.*, 2020; VanDerHorn & Mahadevan, 2021):

- Collect (possibly heterogeneous) data from its physical twin, where this physical twin may contain multiple data sources such as sensors, human observations, etc. This is also referred to as data acquisition.
- Process the data into a suitable, consistent format (typically, an agreed upon, standard format). This is also referred to as data transformation or data pre-processing.
- Store data that is ingested by or created within the DT, as well as metadata and other static data that provides context. Note that not all ingested data must necessarily be stored.
- Use models of the physical twin to generate information about the physical twin's reality. The models use as inputs the data collected from the physical twin or the collected data can be used for model validation.

- Utilise the data in a way that meets the users' requirements. This may involve services built into the DT (that may also be accessible via APIs), but these services are constrained to the data captured within that DT. Utilisation is a broad term that encompasses any functionality related to decision-making, e.g. control, modelling, data visualisation, data analytics, etc. This is also referred to as data consumption.
- Communicate the data, insights and decisions contained within the DT to the physical twin (when applicable), as well as to other entities, such as operators or other DTs.

## 2.2 Aggregation of digital twins

Since its initial conceptualisation, the concept of aggregating DTs has evolved to take various meanings.

The concept of aggregating DTs was introduced by Grieves & Vickers (2016) who defined a DT prototype (DTP), a DT instance (DTI), a DT aggregate (DTA) and DT environment (DTE). A DTP describes a prototypical physical object to the extent that the physical object could be produced based on the information contained in the DTP. A DTI is an instance of a DTP that is connected to a specific physical object and it continues to gather information about the physical object during its lifecycle. The DTA aggregates all the DTIs of a certain type to allow for a larger and more complete dataset regarding the operation of a type of physical object. The DTE is an integrated, multi-domain physics application space that makes use of the DTs for multiple purposes, such as simulating future system behaviour.

In Borangiu *et al.* (2019) aggregation is used to collect, process and reduce data from multiple DTs to inform a control application. Karanjkar, Joglekar, Mohanty, *et al.* (2019) performs aggregation on historical data (captured within DTs) to manage the large amount of historical data. Similarly, Pan, Shi & Jiang, (2020) use hierarchical data format (HDF) data compression to aggregate large amounts of heterogeneous data to deliver a more complete and unified data representation.

In Lutze, (2019) *personal digital twins* are used to keep individual medical records of patients, while *group digital twins* and *system digital twins* are aggregations of personal digital twins, based on certain criteria, that are used to train machine learning models. While the personal DTs contain a collection of all relevant medical data about a patient, the group and system DTs only collect data relevant to the desired model and thus also omits data such as the patient's name to preserve patient privacy and anonymity.

Architectures for DT aggregations have been proposed. Villalonga, Negri, Biscardo, *et al.* (2021) propose a hierarchical aggregation approach, where *local digital twins*

are concerned with asset health monitoring and diagnostics while *global digital twins* are concerned with decision-making. Ciavotta *et al.* (2018) proposes a framework where data from multiple digital twins are used to define layers of information (where each layer addresses a concern) and these layers can then be aggregated into different collections depending on the user's interest and desired level of detail. The six-layer architecture for digital twins with aggregation, SLADTA (Redelinghuys, 2019; Redelinghuys, Kruger, *et al.*, 2020), proposes a hierarchical assembly of DTs. The following section considers SLADTA in more detail because it is the reference architecture used for the internal design of the DTs in the system of DTs.

## 2.3 The six-layer architecture for digital twins with aggregation

The Six-Layer Architecture for Digital Twins (SLADT) is a reference architecture for digital twin development and it has been applied to a manufacturing cell for close to real-time monitoring and fault detection (Redelinghuys, Basson, *et al.*, 2020). SLADT with Aggregation (SLADTA) is an extension of the SLADT framework that allows multiple digital twins to aggregate data for a system level perspective (Redelinghuys, 2019; Redelinghuys, Kruger, *et al.*, 2020). SLADTA is presented in Figure 1 where the individual DTs are designed according to the SLADT layers and where aggregation of DTs is performed through layer 4.

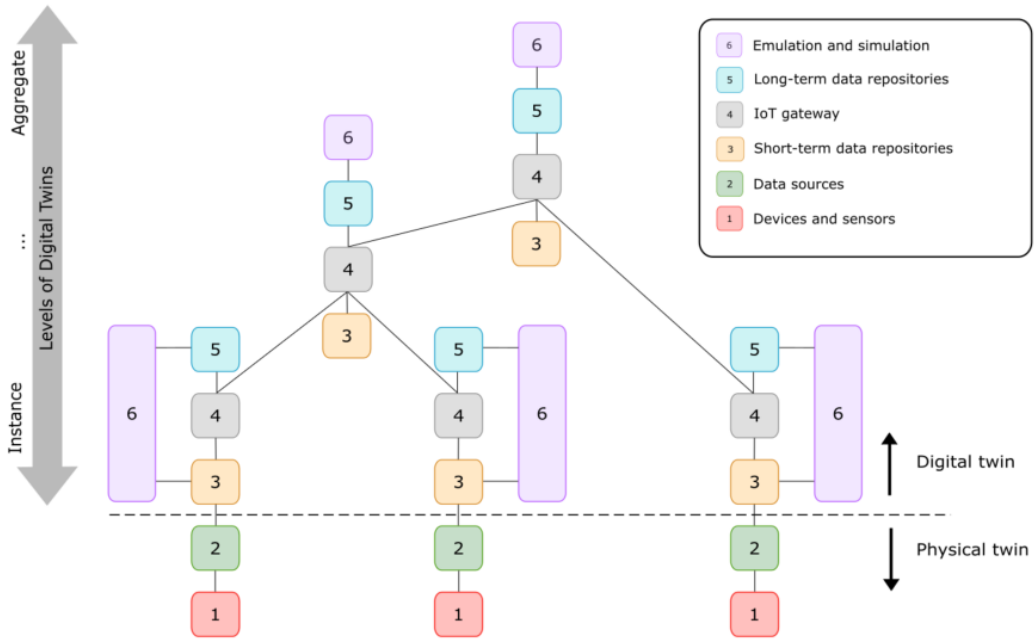



**Figure 1: The six-layer architecture for digital twins with aggregation. (Adapted from Redelinghuys, Kruger, *et al.*, 2020)**

The six layers of the SLADT are 1) *devices and sensors*, 2) *data sources*, 3) *short-term data repositories*, 4) *IoT gateway*, 5) *long-term data repositories* and 6) *emulation and simulation*. Layer 1 (devices and sensors) refers to physical devices, such as actuators and sensors, that can send and receive signals. Layer 2 (data sources) refers to devices, such as controllers, that can virtualize data by processing the signals received from Layer 1 and that are able to induce physical changes based on virtual information. Layer 2 is considered separate from layer 1, because it can perform functions in physical space as well as virtual space. Together, these two layers form the physical twin.

Layer 3 (short-term data repositories) consists of data repositories that are typically hosted near the physical twin. Such repositories would include a database hosted on the Layer 2 device. For example, a Raspberry Pi microcomputer can be used to perform the functionality of Layer 2 and host a local database to fulfil the functionality of Layer 3. Although, smart sensors and controllers could send data directly to the cloud, often there is a desire or a necessity for local short-term storage, particularly in large systems. This could be for security, reliability, latency, etc. and thus layer 3 is included in this architecture.

Layer 4 (IoT gateway) is custom-developed software that links layers 3 and 5 by coordinating and managing data flow. This includes data reduction and basic data pre-processing, for example, calculating a time difference as opposed to sending a start and end timestamp. Layers 3 and 4 are typically responsible for aggregation and thus, when considering the SLADTA, these two layers become essential.

Layer 5 (long-term data repositories) is intended to be a collection of long-term storage databases, typically hosted in the cloud, that contain the data sent by Layer 4. Layer 6 (emulation and simulation) is an information endpoint where the information is utilized for an intended purpose. The most common purposes include system monitoring, data visualization, data analysis and simulation (Minerva *et al.*, 2020).

The SLADTA adopts the terms DTI and DTA from (Grieves & Vickers, 2016), but the term DTI refers to any DT that has a direct link to a physical twin, while a DTA refers to any DT that aggregates data from multiple DTIs and/or other DTAs. In Figure 1, three DTIs can be observed (where DTIs have Layers 1 and 2) and two DTAs can be observed (Layers 1 and 2 are absent). The DTAs can be considered as a *digital twin of twins* because they make use of other DTs to provide a system perspective.

In the case study presented by Redelinghuys (2019), the short-term local repositories were Open Platform Communications Unified Architecture (OPC UA) servers. In the case study, a DTI's IoT gateway sends data to its OPC UA server which passes the data to a DTA's OPC UA server (if they use separate servers) and then the OPC UA client in the DTA's IoT gateway can access the data. Therefore,

aggregation is coordinated by Layer 4 but performed through Layer 3. Aggregation through Layer 3, as described by Redelinghuys (2019), is enabled by OPC UA and if any other short-term storage solution is used, this type of aggregation would require DTs to have a shared database. However, it is not always feasible for DTs to share a database, particularly in geographically distributed systems. Therefore, this dissertation considers an adaptation of SLADTA, where aggregation is performed through Layer 4, as discussed in (Human, Kruger & Basson, 2021).

The SLADTA is a vendor-neutral, general framework for the development of digital twins. Through aggregation, the architecture enables system-level decision-making by gathering only the necessary data from lower-level DTs, thereby also limiting data exposure, for privacy and confidentiality reasons, and limiting data bottlenecks. In this dissertation, the SLADTA has been chosen for the internal design of DTs because it adheres to the criteria and functionality presented in Section 2.1 and it allows for the aggregation of DTs.

## 2.4 Combining digital twins and service-oriented architectures

Microservices and SOAs have a high level of adaptability, making them an attractive solution to exploit the data captured within DTs (Ciavotta *et al.*, 2020; Pernici, Plebani, Mecella, *et al.*, 2020). Therefore, a number of approaches have been proposed to combine DTs and SOAs.

The *MAYA platform* aims to join microservices, DTs and big data within a manufacturing environment to extend the capabilities of cyber-physical systems (CPSs) beyond control to include advanced simulation and big data capabilities (Ciavotta *et al.*, 2020). The MAYA platform consists of three high-level components, namely the *MAYA Communication Layer* (MCL), the *MAYA Support Infrastructure* (MSI) and the *MAYA Simulation Framework* (MSF) (Ciavotta, Alge, Menato, *et al.*, 2017; Ciavotta *et al.*, 2020).

Essentially, the MCL is responsible for interfacing with the CPSs and integrating them into the rest of the platform, the MSI is microservices and big data middleware responsible for managing the DTs and the MSF contains the DTs (encompassing simulations and models). Therefore, MAYA uses microservices (in the MSI) to manage data between the CPSs (exposed by the MCL) and DTs (in the MSF).

Kuhn *et al.* (2020) describes the use of the *asset administration shell* (AAS), as proposed by the Reference Architecture Model Industry 4.0 (RAMI4.0) (Adolphs, Bedenbender, Dirzus, *et al.*, 2015), to create DTs of products, processes or production machines within a manufacturing environment. The AAS captures data about the physical asset or process and exposes that data in an SOA where a

stakeholder can then access the data via a dashboard. The dashboard makes use of a service orchestrator and service registry to find and coordinate services to achieve the application requirements.

The *AgileChains* architecture is a service-oriented DT approach in the context of supply chain management (Pernici *et al.*, 2020). The AgileChains architecture has three primary components: the DTs, the *coordinator* and the *smart dataspace.* This approach proposes that each DT exposes a set of services that are related to the physical twin, such as monitoring, diagnostics and prognostics. The coordinator orchestrates the services provided by the DTs to achieve a given process plan. Finally, all the data is stored in the smart dataspace which is a collection of heterogenous data sources that implements data mapping to translate and exchange data between entities.

The *service-oriented digital twin* approach was proposed in the context of manufacturing, specifically as a means to empower manufacturing employees to make better decisions regarding the machines that they are responsible for (Longo *et al.*, 2019; Padovano, Longo, Nicoletti, *et al.*, 2018). This architecture consists of the service-oriented DTs that each expose a set of services through RESTful APIs. The manufacturing employees (and other stakeholders) can then access these services through a remote terminal unit (RTU), such as a smartphone or tablet. All the communication between the DTs and the RTUs are facilitated by an enterprise service bus (ESB).

The approaches discussed above can be divided into two groups. The first approach is to keep the services separate from the DTs (the DTs only encapsulate data, models and simulations) and then allow the services to interact with the DTs. The ASS and MAYA platform follow the first approach. The second approach is to include all the services within the DTs and then orchestrate the services between the DTs using, for example, an ESB. The AgileChains and service-oriented digital twins architectures follow the second approach.

## 2.5 Digital twin design frameworks

The design framework presented by VanDerHorn & Mahadevan (2021) consists of four primary steps and only considerers a single digital twin, although it can be a digital twin of a complex system. The four steps are 1) *specify the intended outcomes*, 2) *scope the solution* (by defining the physical system of interest and the levels of abstraction), 3) *create a virtual representation* and 4) *establish the required data interconnections*.

Specifying the intended outcomes (step 1) refers to establishing measurable and quantifiable deliverables that the DTs must provide to ensure user satisfaction. Scoping the solution (step 2) consists of two sub-steps namely: 2.1) determining

the physical reality of interest and 2.2) determining the levels of abstraction of the models. Determining the physical reality of interest (step 2.1) refers to determining the boundary between the physical reality being reflected and the environment it is operating in. Determining the levels of abstraction of the models (step 2.2) refers to the process of determining the level of detail of the data required for the models that form part of the DT.

Creating the virtual representation (step 3) involves the creation of the data model (step 3.1) and then the creation of a computational model (step 3.2). The creation of a data model (step 3.1) refers to determining what data features are applicable, how the data will be stored and how the data will be visualised. The creation of a computational model (step 3.2) refers to the development of computational models that simulates the behaviour of the physical reality of interest.

Finally, establishing the required data interconnections (step 4) involves the establishment of 4.1) how the data will be collected, 4.2) the frequency at which data is collected and 4.3) how data is exchanged between physical and virtual spaces.

An alternative design framework presented by Moyne, Qamsane, Balta, *et al.* (2020), follows object-oriented programming principles. The authors present the object-oriented DT framework which specifies four aspects with regards to their *baseline digital twin object-oriented framework*, including:

- A DT object class, which specifies that each DT class is a type of DT that delivers a specific capability to the DT client. Therefore, DTs belonging to the same class must have the same generally defined scope and commonly defined behaviour.
- A generalisation hierarchy and inheritance, which states that the output metric, common behaviour and the scope of the sub-class must fall within the super-class. The sub-class is typically a refinement of the superclass that better suits a specific application. For example, a super class may be “motor”, while a sub-class may be “conveyor motor” or “robot motor”.
- Aggregation hierarchy, which is a specification that allows for the combination of DT object instances. Aggregation membership is specified in terms of purpose and scope. For example, aggregate all entities that deliver a capability to production line “A”.
- Instantiation and implementation, which refers to the instantiation of a given DT implementation. The DT can occur anywhere in the hierarchy or it can be restricted to be above or below a certain point. This essentially refers to instantiation rules that govern which sub-class in a hierarchy of classes is applicable to which physical objects.

# 3 Needs related to managing system complexity

## 3.1 Overview

A complex system is a system that consists of a large network of components which give rise to complex collective behaviour, sophisticated information processing and adaptation via learning or evolution (Mitchell, 2009). System complexity can be described as a measure of how difficult it is to understand the behaviour of a system and how difficult it is to predict the consequences of changing the system (SEBoK Editorial Board, 2021).

Some of the most cited characteristics of complex systems are independence, interconnectedness, diversity and adaptability (Page, 2009). Independence refers to autonomous system elements that can make their own decisions, influenced by information from other system elements. Interconnectedness refers to system elements that are connected via physical connections, shared data or through visual (sensory) awareness. Diversity refers to the differences between system elements, for example technological differences or differences in function. Adaptability refers to system elements' ability to self-organise in response to their environment to support themselves or the entire system.

Similar to complex systems, systems-of-systems are an assemblage of other systems, where the following criteria must be satisfied: the subsystems must be 1) operationally independent, and 2) managerially independent (Maier, 1999). Sage & Cuppan (2001) expanded the list of criteria to include: a system-of-systems must 3) be geographically distributed, 4) have emergent behaviour and 5) evolve over time (its structure, function and purpose is continually evolving).

This chapter classifies commonly cited needs related to software design for complex systems, where software design refers to any software-based entities such as data, models, services, etc. The needs are functionally defined and are classified into multi-stakeholder, integration, evolutionary, reliability, data related, infrastructure and development needs. The taxonomy introduced here is to help make the needs more understandable and identifiable, but it must be noted that most of these needs do span over more than one of the categories. Each of the following subsections start with an introductory paragraph and table that summarises the needs and indicates where each need is addressed by the design framework. The table entries are explained after the table.

## 3.2 Multi-stakeholder complexity

A system of DTs is expected to have many stakeholders involved in multiple parts of the system's lifecycle and these different groups of stakeholders should be accommodated. Stakeholders can be divided into primary users, secondary users and indirect users (BSI, ISO & IEC, 2011). Primary users are users who directly interact with the system to achieve the goal of the system, such as machine operators and data scientist. Secondary users are users who contribute to the composition and functioning of the system, such as component suppliers and system administrators. Finally, indirect users are users that rely on the outputs from the system, but without interacting with the system directly. This section highlights commonly cited needs related to multi-stakeholder environments which are summarised in Table 1.

**Table 1: Multi-stakeholder needs and how they are addressed in the design framework.**

| Need number | Need | Design framework reference |
|---|---|---|
| N1 | Handle service requirement conflicts | Hierarchical aggregation (Chapter 5) helps address this need as well as the service allocation method discussed in Section 8.2. |
| N2 | Enable secure data sharing and storage | Section 10.1, as well as the security design pattern (Section 11.6), address issues related to secure data sharing and storage. |
| N3 | Provide for proprietary technologies | This is part of the motivation for the design framework being vendor neutral (Chapter 6) and the compatibility design pattern helps address interoperability issues (Section 11.4). |
| N4 | Identify and address requirements imposed on the system by external regulatory bodies. | The design framework includes provision for external requirements within its requirements breakdown (Section 4.2). However, the implications of such externally imposed requirements are case dependent. |

N1. Different stakeholders reasoning about different decisions regarding different system components and subsystems will be interested in different datasets, with different levels of detail (Durão *et al.*, 2018; Villalonga *et al.*, 2021). These different datasets and levels of details are referred to as *viewpoints* by van Geest, Tekinerdogan & Catal (2021). The differing viewpoints required by different stakeholders may give rise to conflicting system and service requirements (Galster & Bucherer, 2008). Therefore, the system of DTs must provide mechanisms to handle requirement conflicts and to minimise trade-offs.

N2. To maintain a competitive edge and to be rewarded for their investments into research, many companies are concerned about their intellectual property and data security (Moyne *et al.*, 2020). In response to concerns about data security, privacy and confidentiality, the Centre for Digital Built Britain (CDBB, 2018) emphasise the need for the system of DTs to be trustworthy. This essentially means that the system of DTs must enable secure sharing of relevant and accurate data between different subsystem and data owners. It also requires the secure storage and regulated access to data within the system.

N3. The system of DTs is intended to have multiple contributors to the physical and software subsystems. These contributors are likely to have differing preferences and requirements related to their subsystem. Therefore, there is a need to provide for differing and proprietary technologies, such as differing programming languages, communication protocols and data formats.

N4. Some domains, such as manufacturing and healthcare, have regulatory and quality constraints that may inhibit the adoption of new technologies and may require extended quality testing procedures (O'Donovan, Leahy, Bruton, *et al.*, 2015). Furthermore, institutions and countries also have specific data privacy and confidentiality needs (Harper *et al.*, 2019; O'Donovan *et al.*, 2015) Therefore, external quality and testing requirements, such as those imposed by regulating bodies, must be identified and addressed.

## 3.3 Integration complexity

Complex systems typically have many contributors, many of which may be third-party service providers. To ensure that the system functions as expected, the internal components from different development teams as well as the third-party components, must all integrate seamlessly. Therefore, this section highlights commonly cited integration and interoperability challenges. These challenges are summarised in Table 2.

**Table 2: Integration needs and how they are addressed in the design framework.**

| Need number | Need | Design framework reference |
|---|---|---|
| N5 | Provide guidelines and standards for software interaction | The design framework allows for better communication of the software between stakeholders (Chapter 6) as well as guidelines to improve interoperability through the compatibility design pattern (Section 11.4). |

| Need number | Need | Design framework reference |
|---|---|---|
| N6 | Allow for retrofitting and differing levels of technological maturity | The design framework considers issues related to long-term maintenance and reconfiguration and provides the maintainability (Section 11.3), compatibility (Section 11.4) and portability (Section 11.5) design patterns to help address such issues. Hierarchical aggregation is also well suited to handle reconfiguration problems (Chapter 5). |
| N7 | Integrate with new and existing information systems | The design framework facilitates the identification and management of data sources and relationships in various parts of Chapter 9. The compatibility design pattern (Section 11.4) also addresses such issues. |
| N8 | Allow for the integration of humans | The design framework does not directly address the integration of humans, but the framework is flexible enough to accommodate humans that are represented by digital administration shells. |

N5. Software artifacts must adhere to exacting specifications for proper functioning and interaction, but there is no universal law that governs the exacting specifications (Brooks, 1995). Therefore, guidelines and standards must be agreed upon that allow for the interaction of the software artifacts.

N6. Complexity management related to long-term maintenance, repair and end-of-life behaviour is increased by the succession of old devices with newer versions, as well as by software and firmware updates (Lutters, 2018). Therefore, the system must provide for differing levels of technological maturity. Many companies have made significant investments into their existing IT and automation infrastructures and often that infrastructure is still fully operational and effective (O'Donovan *et al.*, 2015). Therefore, the system of DTs must provide for retrofitting to accommodate legacy systems (Liu, Leng, Yan, *et al.*, 2020).

N7. Aspects such as retrofitting, long-term maintenance and the adoption of new technologies over time also give rise to the need to integrate with new and existing information systems and solutions. For example, it may be necessary to integrate with an existing local database and migrate it to a cloud platform. Manufacturing system also typically have SCADA system already installed and integrating with SCADA systems is a common need in the manufacturing domain.

N8. Some industries are labour intensive and the role of humans within the system of DTs may be uncertain. This may cause hesitance to the adoption of DTs and similar enabling technologies (Bertoli, Cervo, Rosati, *et al.*, 2021).

Furthermore, humans are more capable and/or more feasible options for solving certain problems than digital or robotic solutions. Humans are often integrated into digital environments using administrations shells that capture and manage data related to the humans and their interaction with other systems. Example of such administrations shells are the RAMI 4.0 AAS (Adolphs, Bedenbender, Dirzus, *et al.*, 2015; Kuhn *et al.*, 2020) and the BASE administrations shell (Sparrow, Kruger & Basson, 2021). Therefore, it is important to provide for the integration of humans that are represented by digital administration shells.

## 3.4 Evolutionary complexity

Users' interests and needs are likely to change over time. This is one of the main drivers of system evolution and is (at least partially) addressed by changing aspects of the system to adapt to the new needs. Physical components can be added, removed, exchanged or changed to adapt to changing demands or changing system ability. Software is also highly malleable (easily changed) and is the most frequently changed part of a software intensive system (SEBoK Editorial Board, 2021).Therefore, this section considers needs related to system evolution and these needs are summarised in Table 3.

**Table 3: Evolutionary needs and how they are addressed in the design framework.**

| Need Number | Need | Design framework reference |
|---|---|---|
| N9 | Allow for efficient system reconfiguration | Hierarchical aggregation helps accommodate system reconfiguration (Chapter 5). Furthermore, the service separation guidelines (Section 8) and the maintainability (Section 11.3) and portability (Section 11.5) design patterns further aid in reconfigurable design. |
| N10 | Verify and validate system changes and manage dependencies | Hierarchical aggregation helps to clearly define system relationships (Chapter 5). The design framework also promotes the traceability of design choices which helps change management (Section 6). Furthermore, the design patterns (Chapter 11) provide some evaluation metrics. |
| N11 | Provide support services | The support services are detailed in Section 10.2 and the design patterns in Chapter 11 also make recommendations for support services. |
| N12 | Provide decision validation and feedback where possible | This need is not directly covered in this dissertation since it is very case dependent. |

| Need Number | Need | Design framework reference |
|---|---|---|
| N13 | Manage machine learning model changes | This need is not directly covered in this dissertation since it is very case dependent. |
| N14 | Allow for easy system maintenance and extension | Hierarchical aggregation eases system maintenance and extension (Chapter 5). The maintainability (Section 11.3) and portability (Section 11.5) design patterns help design systems for easier maintenance and extension. |

N9. Complex systems and systems-of-systems adapt and evolve over time, meaning that they change in composition and orientation over time. (Sage & Cuppan, 2001; SEBoK Editorial Board, 2021). However, large structures can be difficult and expensive to design, maintain, and modify (Duffie, Chitturi & Mou, 1988; Ismail, Truong & Kastner, 2019). Software systems also often undergo refactoring, which can affect multiple software components and services (Engel, Langermeier, Bauer, *et al.*, 2018). Therefore, the system of DTs must allow for system reconfiguration (including the addition, removal, exchange or change of components), while minimising the impact of the changes on the surrounding subsystems.

N10. As the system evolves, the behaviour of the system may change and this change may be desirable or undesirable, expected or unexpected. In some domains, such as manufacturing and industrial automation, minimising system downtime is vital. For this reason, such domains are very cautious to introduce changes (such as software changes or new technologies) into the existing system because these changes may introduce new faults and cause failures (Ciavotta *et al.*, 2020; Ismail *et al.*, 2019; O'Donovan *et al.*, 2015). Therefore, there is a need to implement a means of assessing the effect of system changes (physical or software changes), i.e. there is a need for DT verification and validation in response to system changes (Moyne *et al.*, 2020). A particular subsection of this need is the need for dependency management in software design (Aderaldo, Mendonça, Pahl, *et al.*, 2017; Engel *et al.*, 2018).

N11. Complex software architectures usually require additional support functions and services to enable the components to work together and to work efficiently despite some components changing regularly (Balalaie, Heydarnoori, Jamshidi, *et al.*, 2018; Ciavotta *et al.*, 2020; Engel *et al.*, 2018; Karabey Aksakalli, Çelik, Can, *et al.*, 2021). Examples of such services include discovery services, coordination and orchestration services, debugging services, load balancing, service monitoring, etc. Therefore, the system of DTs must provide the necessary support functions for a particular use case.

N12. Understanding or even quantifying the effect of decisions made within and about the system (where the effects can be short-term or long-term) can be particularly difficult in complex systems. Therefore, feedback should be provided, where possible, on the effect of decisions that have been made regarding the system, i.e. provide decision validation and feedback where possible (for both short-term and long-term decisions) (Villalonga *et al.*, 2021).

N13. Lutze, (2019) mentions that the continuous learning ability of software (such as periodically updating machine learning models with new datasets) presents promising prospects for improving performance while simultaneously raising severe concerns related to the deterioration of performance. Therefore, it is important to manage machine learning model changes within the DT.

N14. Many companies lack the expertise and knowledge required to further develop and maintain the software and infrastructure that forms part of the system of DTs (Bertoli *et al.*, 2021; Ciavotta *et al.*, 2020; Kuhn *et al.*, 2020). Therefore, long-term system maintenance and extension should be simplified as far possible.

## 3.5 Reliability related complexity

The system evolves over time as discussed in Section 3.4 and new components must be integrated into the system. Therefore, it is important to ensure that the system remains available and performs its functions satisfactorily despite the changes. This section highlights commonly cited needs related to system reliability and availability. Table 4 provide a summary of these reliability needs.

**Table 4: Reliability needs and how they are addressed by the design framework.**

| Need number | Needs | Design framework reference |
|---|---|---|
| N15 | Provide a fault tolerant system | The reliability design pattern (Section 11.2) makes suggestions for fault tolerant systems. |
| N16 | Automate system support functions | The support services in Section 10.2 help automate tasks and the maintainability (Section 11.3) and portability (Section 11.5) design patterns help incorporate such support functions. |
| N17 | Manage sensor health | This need is not directly covered in this dissertation. |
| N18 | Ensure high availability | The reliability (Section 11.2) and performance efficiency (Section 11.1) design patterns help design highly available systems. |

| Need number | Needs | Design framework reference |
|---|---|---|
| N19 | Accommodate testing methods and metrics | This need is not directly covered in this dissertation but the design patterns do provide some potential testing metrics. |

N15. System components often rely on inputs from other components to inform their own functioning and behaviour. However, other components might not always be available, may be delayed in their response, might not provide the expected response, etc. (the dependence of a component on other components is referred to as the system's vulnerability to volatile inter-actor behaviours) (Engel *et al.*, 2018; Lindsay, Gill, Smirnova, *et al.*, 2021; Pernici *et al.*, 2020). Therefore, system components must be able to handle faulty information, connection breakdowns, delayed connections, lost messages, etc. to prevent undesirable component behaviour.

N16. Manually maintaining a large system of physical and software components and services is cumbersome and time consuming (Ciavotta *et al.*, 2020; Moyne *et al.*, 2020). Therefore, the automation of various aspects regarding the testing and validation, (re)deployment, fault handling and configuration of DTs is suggested by many researchers (Aderaldo *et al.*, 2017; Ciavotta *et al.*, 2020; Moyne *et al.*, 2020; Therrien *et al.*, 2020; VanDerHorn & Mahadevan, 2021).

N17. Sensors and devices need to be maintained and they eventually degrade. This influences the quality of the data being captured and can significantly influence all downstream uses of that data such as modelling and services. Therefore, sensor lifetime management, health checking and cross-validation may be required. (Therrien *et al.*, 2020)

N18. The system of DTs is intended to help users make better decisions related to their area of interest. This is only possible if the user can use the system when the decision(s) need to be made. Therefore, the system of DTs must be available to users when they need it.

N19. Software testing and validation for distributed software components is non-trivial because the software is deployed across different types of hardware, the software must integrate with many other components (including external components) and testing all possible scenarios and variations of software execution is not feasible (Petrova-Antonova, Manova & Ilieva, 2020). Therefore, good testing and validation procedures and methods should be accommodated.

## 3.6 Data related complexity

Complex systems have many data sources, data is exchanged between many components and processing within components can differ. The abundance of data sources, data processing and data endpoints can cause some difficulty related to the management of the data within the system. Therefore, this section highlights commonly cited issues related to managing data within a complex system. Table 5 summarises these data related needs.

**Table 5: Data related needs and how they are addressed by the design framework.**

| Need number | Need | Design framework reference |
|---|---|---|
| N20 | Identify and address data management issues | Section 9.2.1 lists some common data quality and management issues and the effect that those issues have on aggregation and the architecture. |
| N21 | Adequately structure and supplement data | Considerations for heterogeneous systems are made throughout the dissertation and are summarised in the compatibility design pattern (Section 11.4). |
| N22 | Provide for multiple data viewpoints | Hierarchical aggregation (Chapter 5) and the inclusion of a service architecture (Section 6) help to provide for multiple system viewpoints. |
| N23 | Facilitate heterogeneous data handling | The compatibility design pattern (Section 11.4) helps design interoperable systems despite heterogeneity. |
| N24 | Provide for high data capacity requirements | The performance efficiency design pattern (Section 11.1) helps design system with high throughput needs. |

N20. Data generated by some components are used as inputs to other components, meaning that data is widely shared among software components. This results in data related issues such as data integrity, consistency and persistence management across the distributed components (Bourque & Fairley, 2014; Zimmermann, 2017). Therefore, the relevant data management issues must be identified and structures and/or guidelines must be put in place to facilitate them.

N21. Therrien *et al.* (2020) mention data collections called *data swamps* and *data graveyards*, which refers to collections of data that have become unusable

because of their lack of structure and metadata. Therefore, data should be adequately structured and supplemented with metadata.

N22. Lutters (2018) mentions how the increasing amount of data for decisions support, as well as the difficulty of predicting the outcome of a decision, stifles engineers' ability to make decisions in a timely manner. This is because engineers cannot inspect and understand all the information fast enough. Lamb (2019) shares this concern and mentions that the national DT concept (DT of Britain as proposed in (CDBB, 2018)) is highly complex, diverse and very large and that it hinges on the ability to be: 1) *multi-scale*, being able to represent small components to the whole system (referred to as variety in spatial scale in (CDBB, 2018)), 2) *multi-component*, being able to model distinct but interconnected assets, and 3) *composite*, including federated models from different sources. Therefore, data obscurity should be prevented by allowing for an information overview, while also providing quick access to relevant viewpoints of different subsystems and components.

N23. Diversity in complex systems give rise to one of the most cited challenges in complex system – the need for heterogenous data handling (Engel *et al.*, 2018; Ismail *et al.*, 2019; O'Donovan *et al.*, 2015; Pan *et al.*, 2020; Pernici *et al.*, 2020). Diverse systems produce data that can differ in type, range, structure, sampling rate, etc and extensive amounts of time and money are spent getting data in the right format before it can be utilised (Lutters & Damgrave, 2019). This is closely related to the need, expressed by many organisations, to integrate data silos for a more holistic view of the system (Lamb, 2019). However, to integrate data silos, efficient communication mechanisms and network management are required to transmit the data effectively and this may include the need to support heterogeneous communication protocols (Ismail *et al.*, 2019; Lutze, 2019; O'Donovan *et al.*, 2015).

N24. The interconnectedness of complex systems results in large networks of diverse components where each component displays individual behaviour, while also contributing to a complex collective behaviour (Mitchell, 2009). To digitally represent such a system, a large amount of data must be captured and exchanged resulting in high levels of data traffic and complex processing (Huang, Liu, Xiong, et al., 2020). Therefore, the system of DTs must have the capacity to capture, process, transmit and store a large amount of data in a timely manner (Bertoli et al., 2021; Pan et al., 2020).

## 3.7 Infrastructure complexity

The infrastructure and hardware used to host the digital twins contribute to the complexity of the system. Any software intensive system will be dependent on the infrastructure used to support it and thus it is important to identify what complexity may be encountered as a result. Therefore, this section highlights

commonly cited issues related to the hosting infrastructure. Table 6 summarises these infrastructure related issues.

**Table 6: Infrastructure needs and how they are addressed by the design framework.**

| Need number | Needs | Design framework reference |
|---|---|---|
| N25 | Provide for resource constrained devices | The performance efficiency design pattern (Section 11.1) helps design resource efficient systems. |
| N26 | Identify network constraints | Section 9.1.3 provides some considerations when determining network ability. |
| N27 | Avoid physical resource contention | The performance efficiency (Section 11.1) and compatibility (Section 11.4) design patterns help avoid resource contention (particularly the hosting related recommendations) |

N25. Data is often gathered using power and resource constrained devices, such as IoT devices (Pourghebleh & Navimipour, 2017; Ullah, Azeem, Ashraf, *et al.*, 2021). Therefore, there is a need to provide for resource constrained devices and sensors.

N26. Distributed architectures and software systems that make use of cloud platforms are very reliant on the network infrastructure (Taibi, Lenarduzzi & Pahl, 2018). Therefore, important network characteristics, such as network latency, bandwidth and availability, must be identified and managed.

N27. Host machines and hardware often host more than one software component to optimise resource usage (Bourque & Fairley, 2014; Karabey Aksakalli et al., 2021). However, when some software components need to sustain high loads they monopolise the resources (this is referred to as physical resource contention) (Lindsay *et al.*, 2021). This typically happens when there are very dynamic workloads. Therefore, physical resource contention between software components on the same host machine must be managed and avoided.

## 3.8 Development complexity

The development of any system is subject to numerous constraints that limit the solution space that a design must fit into. These constraints significantly influence the quality of the system since they inform decisions regarding trade-offs and they place limitations on the extent to which a need can be met. Therefore, this section highlights commonly cited needs related to development that have not been

mentioned in any of the other sections. Table 7 summarises these additional development needs.

**Table 7: Development needs and how they are addressed by the design framework.**

| Need number | Need | Design framework reference |
|---|---|---|
| N28 | Manage the balance between cost, time and quality. | The design framework is intended to promote systematic and purposeful design and it facilitates a common understanding and purposeful communication amongst stakeholders. |
| N29 | Decompose the system and data effectively | The aggregation hierarchy (Chapter 5) and service separation guidelines (Section 8) help with system decomposition. |
| N30 | Improve primary users' system understanding | The design framework provides some service patterns for DTs in Section 8.1 that can help a primary user understand how a DT can help them. |
| N31 | Improve secondary users' mutual understanding of the system | Design framework is also a framework for project communication and it promote traceability from user needs to design choices (refer to Chapter 6). The separation of concerns facilitated by the aggregation hierarchy (Chapter 5) also makes the systems more comprehensible. |
| N32 | Track system changes | The support functions in Section 10.2 are intended to help track system changes. |

N28. First and foremost is the famous triple constraint of project management: cost (or resources) vs time (or schedule) vs scope (or quality). Many enterprises prefer cheap, plug-and-play solutions that are easy to use and that cause minimal disruption to existing operations (Bertoli *et al.*, 2021). Furthermore, high time pressure often results in a lack of quality (Chung & Do Prado Leite, 2009). Therefore, there is a need for modular solutions that are quick and easy to integrate and a framework that can help ensure quality despite time pressure.

N29. Software decomposition and data separation are not trivial and good decomposition of complex software systems is a commonly cited difficulty in software design (Engel *et al.*, 2018; Karabey Aksakalli *et al.*, 2021; Salah, Zemerly, Yeun, *et al.*, 2016). Poorly decomposed systems result in excessive

communication complexity, interface complexity and unit complexity. Therefore, there is a need for decomposition guidelines and best practises.

N30. Moyne *et al.* (2020) expresses the need to improve the consistency, understanding and assessment of DT benefits through agreed upon metrics. Furthermore, O'Donovan *et al.* (2015) state that without understanding of how a new technology can benefit an organisation, adoption of that technology is not likely. Therefore, support should be provided to improve the understanding and usability of the system and its components, as well as how the components contribute to the system as a whole.

N31. Software has no physical presence and must thus be represented using various methods, e.g. by using diagrams and code, to communicate the composition of the system and the interactions between the components (Brooks, 1995). Therefore, terminology, schematics, architectures, etc. must be agreed upon to promote mutual understanding between contributors.

N32. Keeping an overview of the software system is a challenge, particularly in large distributed systems, such as microservices systems, where services have been independently developed and refactored separately (Engel *et al.*, 2018; Salah *et al.*, 2016). Some related needs are the need for a code version control repository (Aderaldo *et al.*, 2017), the need for portable designs (Moyne *et al.*, 2020) and the need for good document management (Harper *et al.*, 2019). Therefore, there is a need to track system changes.

# 4 Requirements analysis

## 4.1 Background

"A requirement is a condition or capability that must be met or possessed by a system, system component, product or service to satisfy an agreement, standard, specification or other formally imposed documents" (ISO, IEC & IEEE, 2010). Requirements are typically separated into functional and non-functional requirements, where function requirements (FRs) describe what functions the system must perform and non-functional requirements (NFRs) describe how (or how well) the system should perform its functions (ISO *et al.*, 2010).

There are also multiple perspectives on what NFRs are. For example, NFRs can be considered to be requirements that describe the properties, characteristics or constraints of a software system. Alternatively, NFRs can be considered to be requirements that describe the quality attributes that the software product must have (Mairiza, Zowghi & Nurmuliani, 2010). Here NFRs are considered to include properties, characteristics and constraints of the system.

NFRs are often neglected in the development of software system because 1) of a lack in knowledge or experience in development, 2) NFRs are harder to model, verify, test and measure than FRs and 3) high time pressure related to providing a working system often results in a lack of attention to quality (Chung & Do Prado Leite, 2009; Mairiza *et al.*, 2010). In a review of NFRs in the context of agent-based systems, Clark, Walkinshaw and Hierons (2021) also report that the majority of the reviewed information artifacts did not facilitate specification or testing of NFRs.

NFRs should be incorporated early into the software development life cycle to ensure acceptable software performance and customer satisfaction, particularly in large and complex systems (Bajpai & Gorthi, 2012; Chung & Do Prado Leite, 2009; Galster & Bucherer, 2008; Lamb, 2019). Considering NFRs from the start of the development process (as opposed to assessing the NFRs after development/prototyping) is beneficial because it allows traceability from requirements to implementation decisions and it provides a better basis for architectural and project decisions and trade-offs (Poort & De With, 2004). Therefore, the design framework provides a systematic approach that incorporates NFRs and software quality requirements from the start of the design process.

## 4.2 Requirements classification

There are many different requirements classification models and methods. However, finding the best method of classification is not as important as that the

development teams should all understand the classification methods and terminology of the chosen classification model (Chung & Do Prado Leite, 2009). With this in mind, this section combines the taxonomies presented by (Galster & Bucherer, 2008; Poort & De With, 2004) and the ISO 25010 standard. Together, these taxonomies form a comprehensive classification of the design requirements.

Poort & De With, (2004) classify the requirements as follows:

- Primary functional requirements: Functions that directly contribute to the goal of the system or yield direct value to the user. They represent the principal functionality of the system. All primary requirements are functional requirements, but all functional requirements are not primary requirements.
- Supplementary requirements
    - Secondary functional requirements: Functions that help achieve the primary goals without contributing to the goals directly, e.g. management functionality such as logging.
    - Quality attribute requirements: Quantifiable requirements related to the system quality.
    - Implementation requirements: Constraints placed on the system that cannot be measured by assessing the system, e.g. cost constraints, development time constraints. They contribute to the development process, but are not a measure of system quality.

Similarly, Galster & Bucherer (2008) classify the non-functional requirements related to service-oriented development into three groups, namely *process requirements, service requirements* and *external requirements.* Process requirements relate to the development process of the services and include aspects such as cost and time and thus it corresponds to the implementation requirements of Poort & De With (2004). Service requirements are related to the quality of the service, such as reliability, usability, security, scalability, etc and thus they correspond to the quality attribute requirements in Poort & De With (2004). Finally, Galster & Bucherer, (2008) add external requirements, which are any NFRs that are not classified as process NFRs or service NFRs, such as legal constraints and market conditions.

BSI *et al.* (2011) define a hierarchical decomposition of software quality that further decomposes the service NFRs of Galster & Bucherer (2008). BSI *et al.* (2011) decompose quality into attributes which in turn can be further decomposed into sub-attributes, etc. The decomposition continues until a measurable quality-related property is reached. The Systems and software Quality Requirements and Evaluation (SQuaRE) model consists of three aspects: the quality in use model, the product quality model and the data quality model. The

product quality model, which consists of eight quality attributes summarised in Table 8, is the model of interest here.

**Table 8: System and software product quality model. (Adapted from BSI *et al.*, 2011)**

| Quality attribute | Definition | Sub-attributes |
|---|---|---|
| **Functional suitability** | The system's ability to provide functions that meet the stated and implied needs of the user. | Functional completeness, functional correctness, functional appropriateness. |
| **Performance efficiency** | The system's performance relative to the amount of resources used. | Time behaviour, resource utilisation, capacity. |
| **Compatibility** | The system's ability to exchange information with other systems and/or to perform its functions despite sharing a hardware or software environment. | Co-existence, interoperability. |
| **Usability** | The system's ability to achieve specified goals with effectiveness, efficiency and satisfaction while being used by specified users. | Appropriateness recognisability, learnability, operability, user error protection, user interface aesthetics, accessibility. |
| **Reliability** | The system's ability to perform specified functions under specified conditions for a specified period of time. | Maturity, availability, fault tolerance, recoverability. |
| **Security** | The system's ability to protect information and data so that other persons or systems can only have the degree of access appropriate to their level of authorisation. | Confidentiality, integrity, non-repudiation, accountability, authenticity. |
| **Maintainability** | The system's ability to allow for effective and efficient modification to improve, correct or adapt to changes in the environment and requirements. | Modularity, reusability, analysability, modifiability, testability. |
| **Portability** | The system's ability to be transferred between hardware and software environments, as well as between operational and usage environments. | Adaptability, installability, replaceability. |

A definition for each sub-characteristics is given on the ISO 25000 website (ISO & IEC, 2021).

The quality model also specifies three different groups of stakeholders, as mentioned in Section 3.2 (BSI *et al.*, 2011):

- Primary user: Person who interacts with the system to achieve the primary goal.
- Secondary user: Persons who provide support, such as content providers, system administrators and security managers.
- Indirect user: person who receives output, but does not interact with the system.

The quality in use for primary users is significantly influenced by: functional suitability, performance efficiency, usability, reliability and security. The performance efficiency, reliability and security will also concern other stakeholders that specialise in these fields. The compatibility, maintainability and portability are significant for the quality in use for secondary users. (BSI *et al.*, 2011)

Figure 2 presents the requirements classification that will be used further in the design framework. It is a combination of the taxonomies discussed and includes all the requirements classified either as FRs or NFRs. FRs are decomposed into primary and secondary FRs (as described by Poort & De With (2004)), where primary FRs directly contribute to the user needs (e.g. predictive analytics and data dashboards), while secondary FRs provide support functionality (e.g. system data logging for debugging).

The NFRs are here grouped into quality attributes, development constraints and external NFRs. Quality attributes correspond to service NFRs (Galster & Bucherer, 2008) and quality attribute requirements (Poort & De With, 2004) while development constraints correspond to process NFRs (Galster & Bucherer, 2008) and implementation requirements (Poort & De With, 2004). External NFRs are as defined by Galster & Bucherer (2008) and the external NFRs in Figure 2 are examples of external NFRs, not necessarily all the possibilities.

The development constraints and external NFRs describe requirements placed on the system that constrain the development of the system and thus also influence the quality. The following section expands the quality attributes, describing requirements placed on the system to ensure the system functions according to a sufficiently high standard of quality to satisfy the user expectations. The quality attributes are further decomposed according to the software product model as described in BSI *et al.* (2011) and as summarised in Table 8. The design patterns, presented in Chapter11, are also designed based on these quality attributes, where each design pattern focusses on a quality attribute. The development constraints decomposition is discussed in Section 4.4.

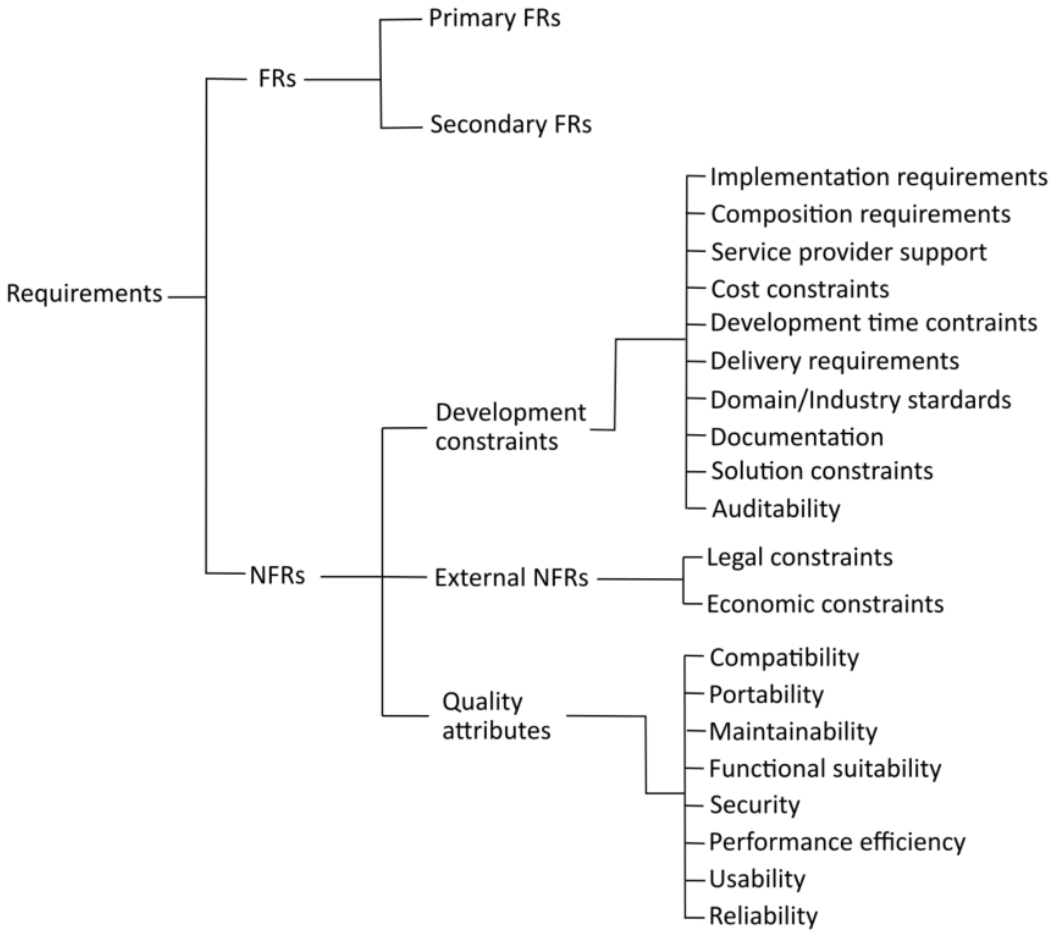



**Figure 2: Requirements classification.**

## 4.3 Alternative quality attributes

The taxonomy presented in Section 4.2 does not cover all the NFRs that have been used in research and in industry. Mairiza *et al.* (2010) present a list of 114 NFRs and further mention that 252 NFRs were identified in their literature study. Therefore, it is not feasible to consider all these NFRs, but this section relates some of the most popular NFRs to the taxonomy to avoid confusion.

Responsiveness is a system's ability to respond to user input in a timely manner and is commonly referred to when referencing reactive architecture design and The Reactive Manifesto (Bonér, Farley, Kuhn, *et al.*, 2014). Responsive systems focus on providing rapid and consistent response times. Responsiveness relates to time behaviour, a sub-attribute of performance efficiency, as well as to availability, a sub-attribute of reliability.

Resilience is a system's ability to remain responsive despite failures (Bonér *et al.*, 2014). Resilience relates to the ISO 25010 requirement of reliability.

Scalability is a system's ability to make use of more computing resources to increase its workload while maintaining stable performance. It is measured as the ratio of throughput gain to resource increase (Bonér *et al.*, 2014; Márquez, Villegas & Astudillo, 2018). Furthermore, scalability can be divided into, 1) vertical scalability, which is the ability to scale to demand by adding resources to a single instance of the program (Tovarnitchi, 2019), 2) horizontal scalability, which is the ability to scale to demand by duplicating the program and load balancing between the identical instances (Tovarnitchi, 2019), and 3) z-axis scalability, which is the ability to scale to demand by duplicating the program by with different subsets of data as presented by the AKF scale cube (Abbott, 2020).

For example, consider an online dictionary application that needs to meet increased demand. Providing the dictionary application with more RAM and more CPU time would be vertical scaling. Duplicating the dictionary application and then load balancing requests between the duplicate instances would be horizontal scaling. Duplicating the dictionary application functions but hosting one version with words from A to M and another version with words from N to Z would be z-axis scaling. Vertical scaling is typically achieved by using a better machine or by using dedicated hardware. Horizontal scaling and z-axis scaling is typically achieved by using multiple, lower-grade, machines.

Elasticity refers to the dynamic allocation of resources at runtime to increase or decrease throughput as demand varies. Elasticity can only be achieved if a system is scalable and thus elasticity is an extension of scalability (Bonér *et al.*, 2014). Therefore, both scalability and elasticity relate to the ISO 25010 quality attribute for performance efficiency, particularly the relation between the sub-attributes of capacity and resource utilisation.

Reconfigurability (in the context of manufacturing and cyber-physical systems) is the ability of a system to allow for the addition, removal, exchange or change of system components to respond to internal or external changes (Kruger & Basson, 2019). Good reconfigurability means that system components can be added, removed, etc. with minimal system down time and ramp-up time, thus improving availability. Reconfigurability more closely resembles the sub-attributes of maintainability and portability, such as modularity, reusability, replaceability and modifiability. Scalability is also one of the critical characteristics for reconfigurability (Koren & Shpitalni, 2010) which is linked to performance efficiency. Therefore, reconfigurability is a characteristic that promotes performance efficiency, portability and maintainability as a means to improve reliability.

Agility is a system attribute that allows for the quick, incremental addition of features to provide new functionality in response to changing user needs (Ciavotta *et al.*, 2017). This relates to the ISO 25010 sub-attribute of modifiability (a sub-attribute of maintainability) and adaptability (a sub-attribute of portability).

Extensibility is the system's ability to integrate new interfaces, data types, connectors and components (Ismail *et al.*, 2019). This relates to the ISO 25010 sub-attributes of adaptability (a sub-attribute of portability) and it is also dependent of the system's interoperability, which is a sub-attribute of compatibility.

## 4.4 Development constraints

The development constraints are constraints placed on the development of the system and thus limit the solution possibilities for the system. Galster and Bucherer (2008) further divide the development constraints into ten sub-categories which are given in Table 9. Each of these development constraints will influence the degree to which the quality attributes can be achieved and it is important to determine how the constraints may affect the design.

**Table 9: Development constraints. (Adapted from Galster & Bucherer, 2008)**

| Development constraints | Description | Examples |
|---|---|---|
| **Implementation** | Constraints on implementation methods or technologies. | This includes constraints such as having to use AWS as a cloud platform or having to use the SCRUM development method. |
| **Composition** | Constraints related to how the software should be composed. | All development must make use of internally produced APIs or only certain packages or framework within a development language are allowed. |
| **Service provider support** | The degree of support offered by external service providers. | Technical support staff, Q&A forums, documentation, tutorials and community support forums are examples of service provider support. |
| **Cost** | The allowed cost of development, equipment, deployment, etc. | A limit on deployment costs may prevent to use of preferred equipment. |
| **Development time** | The amount of time allowed to develop the software or subsections of the software. | Deadlines for milestone events and the expected time-to-market of features. The SCRUM development method breaks the development time into *sprints*. |

| Development constraints | Description | Examples |
|---|---|---|
| **Industry standards** | Standards imposed by a certain industry. | Conforming to the ISO 27001 standard for information security management. |
| **Documentation** | The documentation that should be provided along with the system. | Documentation should provide a list of all systems APIs and examples of how they are intended to be used. |
| **Solution constraints** | Constraints placed on the solution space without considering implementation. | The need to integrate with legacy systems is a common solution constraint. The scope of the solution being implemented also constrains the solution since it must receive certain, externally determined, inputs and provide certain outputs. |
| **Auditability** | The degree to which the system must provide support for audits. | Software audits include aspects such as a review and evaluation of design and development methods, testing and validation procedures, implementation procedures, etc. |

To better balance and manage the development constraints, an incremental implementation approach is often adopted (VanDerHorn & Mahadevan, 2021). This typically helps to reduce upfront cost, reduce time to market and to identify where enhancements are most urgently required. Incremental development strategies also make the development more manageable by dividing the development into attainable sub-tasks.

## 4.5 Quality attribute conflicts

All the quality attributes are important to consider within a complex system, but some of the attributes require conflicting solutions and thus trade-offs are necessary. The relative importance of the quality attributes, the influence of development constraints and external NFRs are case dependent. However, this section provides some of the most cited trade-offs that may need to be considered.

### 4.5.1 Reliability vs agility

Agility is the ability to change easily in response to a changing environment or changing user needs. Software agility is closely linked to agile software design methods which emphasise the incremental development and addition of new

features, functions and services as the operational lifecycle continues. This is referred to as continuous integration and continuous deployment (CICD).

Agility is useful to reduce the time-to-market of software and it is useful to adapt to constantly changing user needs. However, these changes increase the risk of introducing new faults and failures into the current system and thus they increase the risk of downtime (Ciavotta *et al.*, 2020; Kuhn *et al.*, 2020). Therefore, there is a conflict between agility and reliability since reliability is focussed on minimising downtime. Furthermore, one of the sub-attributes of reliability is maturity which is hard to accomplish in a system where features can continually change.

Hardware agility can also negatively impact reliability. In the context of production environments, Lutters (2018) mentions how components are replaced based on their perceived ability to add value that older technologies are not capable of. However, this adds to the complexity of the production environment and causes the system to be less predictable and thus less reliable.

The conflict is caused by the desire for quick system changes in response to external changes. Therefore, it is important to determine the degree to which external changes need to be accommodated. In production and automation environments, these changes need to be handled, but the system does not necessarily have to change because of them. Therefore, in this context reliability is often preferred. On the other hand, in microservices architectures, it is vital that the needs of the users are met and thus agility gains priority. In this context reliability is achieved through other means, such as replication and extensive automated testing, despite the constant changes.

### 4.5.2 Interoperability vs security

The conflict between interoperability and security arises from the need to keep data private and confidential, while also needing to share it among systems to fulfil certain functionalities. In particular, Lamb (2019) mentions the difficultly related to data sharing in an environment where companies want to protect their intellectual property and competitive edge. This problem is echoed by the Centre for Digital Built Britain (CDBB, 2018) and Ismail *et al.* (2019). Collaboration agreements are cited as the most common solution to data sharing in highly competitive environments. However, the negotiation related to the collaboration agreements can last between six months to a year (Griffiths, 2018).

In some domains, such as healthcare, where data is being captured about people, there is a strong need for data privacy. Therefore, when sharing data, there are strict policies that need to be adhered to and, in such cases, people also have the right to withhold consent (Lutze, 2019). The strict policies and need for consent mean that data sharing and, by implication, aspects of interoperability are

hindered or prevented. Such issues must be identified and developers must adhere to such policies.

Furthermore, using good, open standard security methods, such as the SSL/TLS protocol, AES encryption and the OAuth protocol, helps secure data while still maintaining an acceptable degree of interoperability.

### 4.5.3 Interoperability vs performance efficiency

The use of intermediaries, such as brokers, middleware or directories, is a common way to improve interoperability. However, such intermediaries can increase latency since they create extra communication steps and they often perform protocol conversions that require additional processing time. Furthermore, such intermediaries can become communication bottlenecks if they are not properly designed (O'Brien, Merson & Bass, 2007).

High performance middleware technologies typically find a good balance between interoperability and performance efficiency. Designing high performance middleware typically makes use of concepts such as load balancing and message queueing and industrial grade solutions are available for such tasks. Section 10.3 provides more details about messaging solutions for different scenarios.

### 4.5.4 Performance efficiency vs data detail

In the context of modelling within DTs, high fidelity models require more detailed data and thus more processing, storage and network bandwidth than lower fidelity models (VanDerHorn & Mahadevan, 2021). As a result, more resources must be utilised to manage the increased data load and possibly the increased latency. Therefore, it is important to determine what level of data detail is required by models and services. In some cases it is useful to reduce data dimensionality to improve the comprehensibility of the data, even though some detail is lost (Fadlalla, 2005). Furthermore, structuring data to minimise the number of data queries is a way of retaining detail with minimal effect on time behaviour. Pre-computing computationally expensive queries and storing the results of those queries is also common (Fadlalla, 2005), but there is then trade-off between computational and storage resources that needs to be justified.

It is also important to consider data management aspects and how those influence the performance efficiency. For example, when strong data consistency is required, latency is often increased and data persistence management requires long-term storage. The trade-off of performance efficiency vs data detail and quality is further considered in Chapter 9.

### 4.5.5 Performance efficiency vs portability

In the context of distributed software systems, Lindsay *et al.* (2021) refer to the conflict between generalisation and specialisation. Generalisation of software refers to the software's ability to handle a variety of scenarios without having to change the software, whereas specialisation of software refers to the software's ability to perform a specific task exceptionally well. Generalisation prioritises portability, particularly adaptability, whereas specialisation prioritises time performance and resource efficiency (Lindsay *et al.,* 2021). In this dissertation, this issue is facilitated by the separation of concerns. Section 8 provides more details regarding services separation.

### 4.5.6 Security vs performance efficiency

There is always a trade-off between security and performance (Gadge & Kotwani, 2017) because security requires additional processing and possibly extra communication steps and storage. For example, encryption and decryption increase message size and overhead (O'Brien *et al.*, 2007), authentication and authorisation can cause large initial delays in communication while a secure connection is being established and security aspects such as non-repudiation and accountability require logging and thus additional storage.

Security is always a concern in software systems and should never be sacrificed. However, there are degrees of security and there are some methods of ensuring security that do not influence performance too significantly. Typically, this includes using industry standards for security, such as the SSL/TLS protocol and the OAuth protocol, and making use of reputable cryptographic libraries. Security issues are further discussed in Section 10.1 and the security design pattern (Section 11.6) provides some recommendations for security.

### 4.5.7 Security and maintainability

In the context of service-oriented architectures, O'Brien *et al.* (2007) mentions the trade-off between security and maintainability. Common methods of improving maintainability are by promoting a loose coupling between services and through the separation of concerns (Bachmann, Bass & Nord, 2007). However, security measures such as encryption, authentication and authorisation increase the interface complexity and responsibility of services, since the service must include logic to handle security certificates, security tokens, etc. Furthermore, authorisation and confidentiality concerns may cause some services to be restricted making it harder to debug and test such services.

As discussed in Sections 4.5.2 and 4.5.6, it is recommended to prefer security protocol standards for security and making use of reputable cryptographic libraries also greatly eases maintenance issues.

# 5 Aggregation

This Chapter extends upon the concept of aggregating DTs as presented in Sections 2.2 and 2.3. In particular, Section 5.1 introduces a definition and some core concepts related to aggregation. Section 5.2 then discusses hierarchical aggregation principles, while Section 5.3 discussed how the hierarchical aggregation principles can be applied to design DT aggregation hierarchies.

## 5.1 Aggregation definition and core concepts

The following definition for aggregation, suited to computing and software environments, is used here:

*Within a given domain, the process of collecting, and potentially contextualising, various independent (and possibly heterogeneous) entities, followed by the processing of those entities into a unified, coherent entity, where "entities" can be data, information, models, microservices, etc.*

This definition captures some specific aspects of aggregation that can be identified when investigating literature. These aspects are:

- Aggregation is tailored to intended purposes and a certain context.
- Aggregation entails the collection of independent or nearly independent entities (see Section 5.2.2 for a discussion on nearly independent entities). Independent or nearly independent entities are entities that can be considered as individuals, but they may interact and influence each other. Therefore, considering them as a collective can also be beneficial.
- Aggregation entails some processing to be performed on the combination of various entities to produce an aggregated entity.
- The output of the aggregation must be a unified, coherent entity, i.e., an observer (such as a user) can interact with the aggregated entity without knowledge about the entities being aggregated.

The definition of aggregation above, as well as the variety of processing approaches that can form part of aggregation, mean that a single DTI (as defined in Section 2.3) can already be considered an aggregation of, for example, multiple data sources (environmental sensors, machinery, historical data stores) from multiple lifecycle stages. Therefore, it is necessary to clarify what is meant by the aggregation of DTs. The definition proposed for the aggregation in the context of DTs is:

*The process of collecting entities from multiple DTs (the child DTs), followed by the processing of those entities into a single DT that represents an identifiable part of physical reality which encompasses the child DTs' physical realities.*

Therefore, a DTA (as defined in Section 2.3) is the single coherent result when DTs (which can be DTIs or DTAs) are aggregated.

Some further principles related to the aggregation of DTs are explained with reference to Figure 3. In Figure 3 the DTIs are labelled A, B and C and each of them represent a given reality and they encapsulate data, models and services related to that reality. Similarly, the DTAs are labelled AB, BC and ABC depending on which DTIs are aggregated.

The key characteristics of DT aggregation applied here, with reference to Figure 3, are:

- Aggregation follows a hierarchical structure, where DTIs make up the lowest level and DTAs make up one or more levels above the DTIs.
- Entities from one DTI can be aggregated by more than one DTA (B is aggregated by AB and BC) and a DTA can aggregate data from another DTA (AB is aggregated by ABC). For example, data from a robot represented by DTI B can be aggregated to the DTA the represents the cell containing the robot (AB) and to an aggregation of all the robots of one manufacturer (BC).
- If a DTA can be reached by more than one aggregation path, care should be taken to avoid redundant (and potentially conflicting) data. For example, ABC is an aggregate of AB and C, as opposed to being an aggregate of AB and BC. This restriction is intended to preserve data consistency, since AB and BC may manipulate the entities from B in different ways.
- A DTA should be introduced into the hierarchy only where that DTA can add value, such as by reducing data storage and flows by aggregating information not stored long term by lower-level DTs, or by providing models that rely on the data from multiple, lower level, DTs.

For the duration of this document, the term *aggregation* will refer to the process of aggregating DTs. The term *aggregation hierarchy* refers to the concept of dividing the system into subsystems with accompanying DTs and the aggregation of those DTs into different DTAs. The following sections will clarify the term aggregation hierarchy.

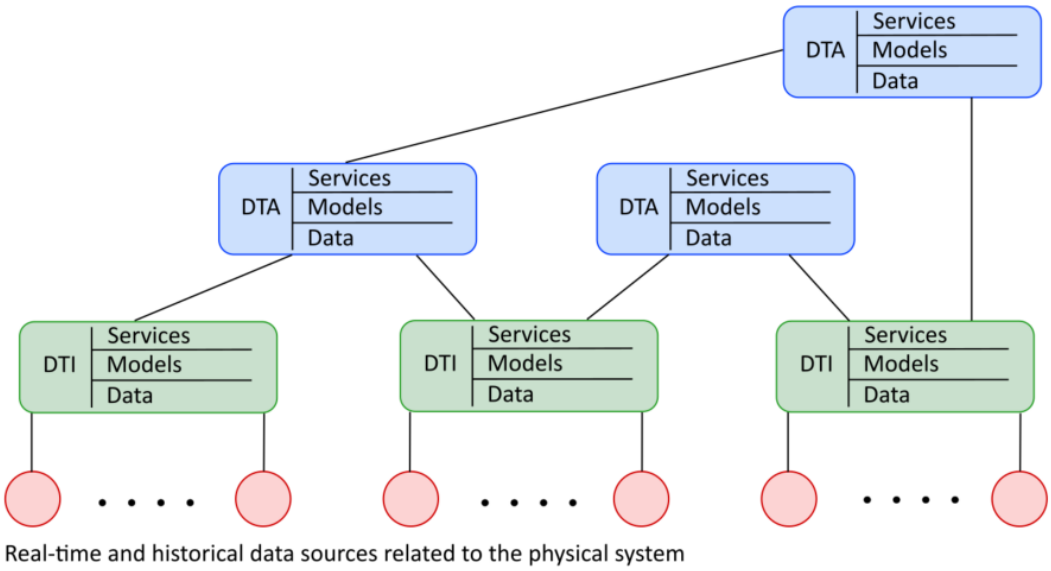



**Figure 3: Hierarchical digital twin aggregation**

## 5.2 Complexity, hierarchies and aggregation

Hierarchies are one of the central structural schemes of any complex system (Simon H.A., 1996). A hierarchy is a system composed of interrelated subsystems where each subsystem can further be decomposed into subsystem until a lowest level elementary element is reached. What constitutes the lowest level elementary element is often a subjective decision and dependent on the case. Hierarchical aggregation is the reverse of decomposition and it refers to the joining of subsystems to create larger subsystems that can also be aggregated to form even larger systems until the largest system perspective is reached.

WSNs and IoT networks along with concepts such as edge computing and fog computing often also make use of hierarchical arrangement and aggregation to manage complexity related to large amounts of data (Huang, Liu, Xiong, et al., 2020; Pastor, Chamizo, Hidalgo, et al., 2018; Bertoli, Cervo, Rosati, et al., 2021; Rajagopalan, Varshney, 2006; Pourghebleh, Navimipour, 2017). Hierarchies offer a structured method of distributing processing and decision-making responsibility across a network, making the system more manageable and comprehensible (Ciavotta *et al.*, 2018; Redelinghuys, Kruger, *et al.*, 2020; Villalonga *et al.*, 2020). Similarly, in SOAs, service hierarchies promote the separation of concerns, loose coupling and service reuse (Buenabad-Chavez, Kecskemeti, Tountopoulos, *et al.*, 2018).

Finally, in the context of DTs, hierarchical architectures provide a high degree of reusability and modularity which allows the DTs to adapt to different applications (Shangguan, Chen & Ding, 2019). Most existing DT architectures also make use of hierarchical models, where each level of the hierarchy has its own function and provides data or information to the level above (Villalonga *et al.*, 2021).

The following subsections introduce some dimensions and principles related to hierarchies in general and then translates those dimensions and principles to make them applicable to aggregation hierarchies in DTs.

### 5.2.1 Dimensions of an aggregation hierarchy

Simon (1996) introduces two dimensions to describe a hierarchy:

- The *span* of the system refers to the number of subsystems that the system is composed of. The span of the system is often limited by, and thus determined by, the system's *capacity for interaction*.
- Hierarchies are often defined in terms of their *intensity of interaction*. The intensity of interaction is typically quantified by the frequency of interaction between entities. In physical systems the intensity of interaction is often largely dependent on the spatial proximity of the subsystems, whereas for social systems only the frequency of interaction is an indication of the intensity of interaction.

These dimensions are discussed in the following subsections. For this discussion, consider the example of multiple machines (with machine twins) within a production line (with a production line twin) that forms part of a manufacturing plant (with a plant twin).

#### 5.2.1.1 Span of reality

To relate the "span" dimension of hierarchies to the aggregation of data and DTs, the term *span of reality* is defined as:

*The extent of the physical reality that is represented by a given digital twin, where a smaller span of reality means a smaller subset of the physical system and its environment is being represented.*

Considering the above-mentioned example, a machine twin has a small span of reality, whereas a production line twin has a larger span of reality and finally, the plant twin has the largest span of reality. Therefore, span of reality can be specified in part by specifying which DTs are being aggregated (and therefore which physical subsystem(s) is/are being represented). This is referred to here as the ***physical scope***. However, since subsets of data can be aggregated and processed in

different ways it is useful to define another dimension here, namely *data granularity*.

Data granularity refers to the detail of the data that is captured within a DTA, where a finer granularity refers to a higher level of detail. Data granularity specifies 1) the desired subset of data features and 2) the time interval between consecutively ingested data records of a feature (for batch data this refers to the amount of data compression and for real-time data this refers to the sampling frequency of the data). Choosing which DTs to aggregate from and defining a level of data granularity helps to determine the data scope of the DTs.

If a DT provides data to a service hosted outside of the DT or if the data ingestion is split from the data utilisation within a DT, then the *data update frequency* should also be determined. Data update frequency is the rate at which data must be output from the DT's database. Data update frequency is largely determined by the potential rate of physical system change, as well as the decision-making frequency. In some cases, the data sampling frequency may be the same as the required data update frequency, but they are not always the same. For example, for real-time control, the sampling frequency and data update frequency might be the same whereas for exploratory data analytics a fine data granularity may be required (many data features with many data records) but the data may only need to be updated once a day.

The Centre for Digital Built Britain (CDBB, 2018) mentions that a DT must provide for a variety of temporal scales, such as an operational timescale, reactive timescale, planned maintenance timescale, capital investment timescale, etc. Each of these requires different levels of data detail. Therefore, the data granularity is chosen according to the services' data requirements. Determining the desired data granularity will then inform what processing operations may be required to produce the desired subset of data.

Furthermore, when choosing the span of reality of a DT, it is important to consider the DT's capacity for interaction. The capacity for interaction of a DT is heavily influenced by the underlying hardware and refers to aspects such as:

- The number of concurrent connections that a DT can maintain.
- The amount of data that can be processed by the DT within a given timeframe.
- The storage capacity and memory allocated to the DT by the host machine.

Span of reality essentially describes the scope of the data that is available and sustainable within a DT. This is important for service-to-DT mapping since a service may require a particular set of data to perform its functions. Table 10 summarises the sub-dimensions of span of reality.

**Table 10: Sub-dimensions and influence of *span of reality*.**

| Sub-dimension of span of reality | Significance of sub-dimension |
|---|---|
| Physical scope | The scope of the physical reality of interest, i.e. which DTs are being aggregated. |
| Data granularity | Determines the scope of the data being aggregated and informs which processing operations are required. |
| Capacity for interaction | Determines the capability of the underlying hardware to support data granularity, which can be a requirement if new hardware will be obtained or a constraint if hardware is already provided. |

#### 5.2.1.2 Intensity of interaction

Based on the intensity of interaction dimension, the physical system will most likely be decomposed based on spatial proximity and interface complexity. Therefore, each DTI will represent an elementary element based on spatial decomposition (this may also be desirable for reconfigurability). However, DTs can form social hierarchies and thus an aggregate entity should be defined according to frequency of interaction rather than spatial proximity alone. As a result, aggregate entities can aggregate according to different types of relations.

There are two primary types of relations that are considered here: a spatial relation and a functional relation. A spatial relation refers to relationships based on physical proximity. For example, aggregating various machines within a production line into a production line twin is aggregation based on a spatial relation. DTAs typically aggregate based on a spatial relation because this is how humans naturally perceive reality and thus how DTs reflect reality.

Alternatively, aggregation can be based on a functional relation, where the relationship is based on functional similarity. For example, aggregating all the electrical information from various machines to determine energy consumption. Services that form part of an SOA or microservices architecture typically aggregate according to a functional relation, which is typically determined using domain-driven design principles (Aderaldo *et al.*, 2017; Salah *et al.*, 2016; Tovarnitchi, 2017)

Therefore, the intensity of interaction can be spatially focussed, in which case aggregation using a DTA may be preferred. Alternatively, the intensity of interaction can be functionally focussed, in which case aggregation using a service in a SOA may be preferred. Either way, the intensity of interaction is determined by the related service's required span of reality. There are also other

considerations when choosing an aggregate entity and these are discussed in Sections 8.3 and 9.3.4.

### 5.2.2 Near decomposability

In hierarchical systems, a distinction can be made between interactions within a subsystem and interaction among the subsystems. The interactions within a subsystem are typically an order of magnitude higher than interactions among subsystems. The principle of near decomposability states that the interaction among subsystems is weak but not negligible when compared to interactions within subsystems (Simon, 1996). This leads to two propositions that can be exploited when considering nearly decomposable systems (Simon, 1996):

- The short-term behaviour of the component subsystems is nearly independent of the short-term behaviour of other component subsystems. Therefore, short-term behaviour can often be closely approximated as independent of other component subsystems.
- The long-term behaviour of any one component subsystem depends on only the aggregate behaviour of another component subsystem, i.e. component subsystem A is not dependent on all the interaction within component subsystem B, but only on the aggregate result of those internal interactions within B. For example, consider two machines within a production line, machine A and machine B, where machine A places a label and machine B stamps the label placed by machine A. Machine B's throughput is not dependent on the power consumption, strain, temperature, etc of each motor in machine A, instead machine B's throughput is only influenced by the throughput of machine A.

In terms of an aggregation hierarchy, the principle of nearly decomposable systems relates to the concept of separation of concerns and the distribution of data and logic across the system. When considering decisions related to the short-term behaviour of a subsystem, only the data and logic internal to that subsystem is required. Alternatively, when considering decisions related to the long-term behaviour of a subsystem, only the aggregated data (that describes the net inputs and outputs) and the associated logic is required. Therefore, there is a natural separation of concerns within hierarchies allowing for distributed decision making.

The principle of near decomposability also highlights an important aspect within aggregation hierarchies, namely that higher levels of aggregation typically have reduced dimensionality and a higher level of abstraction than the lower levels (Engel *et al.*, 2018; Fadlalla, 2005). This means that higher levels of aggregation have more aggregated DTs (a larger physical scope) but with reduced data granularity (a smaller scope of data from each DT). This is how hierarchies manage

large data flows and how hierarchies improve the comprehensibility of complex systems.

### 5.2.3 Intermediary forms

Intermediary forms are stable subassemblies of a system that can exist as individuals and they can be assembled into larger systems (Simon, 1996). The number and distribution of potentially stable intermediate forms is a critical determinant of the time required for the evolution from simple elements to complex systems. Potential for rapid evolution exists in any complex system that consists of a set of stable subsystems, each operating nearly independently. Therefore, to improve the evolutionary ability of the system, i.e. the system's ability to adapt to changes, there must be stable intermediary forms.

Relating this to system of DTs, the existence of intermediary forms refers to the concepts of load distribution, incremental development and component reuse. Intermediary DTs can function as individuals and as part of a collective. As individuals they can be used to perform load distribution, where they are responsible for any pre-processing related to the data that is contained within themselves. As part of a collective they can then make the pre-processed data available to other DTs or services. Furthermore, the distributed nature of the intermediary DTs improves reliability and scalability since they allow for horizontal scaling, individualised vertical scaling and z-axis scaling (Section 4.3 provides a discussion on scalability).

Intermediary forms are also intended to be assembled into larger systems. This relates to the ability to incrementally build hierarchies, where elementary components are developed individually and then aggregated to form larger, more complex systems. This also relates to the divide-and-conquer design strategy, where a system is recursively subdivided into subsystems until a manageable unit complexity is reached for a subsystem. The manageable subsystems can then be developed separately and assembled to form the desired system (Bourque & Fairley, 2014).

### 5.2.4 Reoccurring patterns

Finally, hierarchical systems often contain reoccurring patterns. Hierarchical systems are usually composed of only a few different kinds of subsystems, but they are present in various combinations and arrangements (Simon, 1996). Similarly, in the context of complex production systems, Lutters (2018) mentions how a production plant is recursive and thus a plant can be built from only a very small number of typifications, even though there are a multitude of manifestations and instantiations.

The identification and exploitation of reoccurring patterns in a hierarchy relates to the concept of modularity and proposes that the ideal is to have only a few modules that can be arranged in various combinations. This suggests an optimal number of modules, where too few would mean a lack of generalisation and reusability, while too many modules would cause unnecessary confusion and complexity when ordering and integrating the modules. The intention of the modularity is to improve the reusability of the software, so finding the optimal number of modules could relate to a global optimal reuse rate

Simon (1996) also mentions that through appropriate “recoding”, the reoccurring patterns that are present, but not obvious in structure, can often be made clearer. This relates to the concept of self-similarity in hierarchies and is particularly useful when a mutual problem is “recoded” to allow for a single solution. For example, consider a software system with many interacting software programs but where these programs have different communication protocols. Each program has the problem of communication heterogeneity and instead of solving the problem for each program, a middleware or broker can be used as the single solution to the mutual problem.

Self-similarity also allows for module and solution reuse at various levels of the hierarchy, which allows for reduced development times. Consider again the example of multiple software programs that need to interact. However, in this case the communication protocol has not yet been determined. Then, because all the programs must communicate, a single communication module can be designed that can be used by each program.

## 5.3 How hierarchical aggregation helps to handle complexity

This section considers the benefits of the hierarchical principles discussed in Section 5.2. Table 11 provides a summary of the benefits and how they relate to the quality attributes. A short discussion follows after the table.

**Table 11: Relation of hierarchy principles to engineering design principles and their benefits.**

| Hierarchy principle | Related engineering design principle | Benefits | Quality attributes improved |
|---|---|---|---|
| Near decomposability | Separation of concerns | Allows for multi-stakeholder and multi-services provisioning<br><br>Enables parallel development<br><br>Reduces data flows<br><br>Eases integration | Maintainability<br><br>Portability<br><br>Performance efficiency |
| Intermediary forms | Load distribution<br><br>Incremental development | Allows for scalability (in all three axes)<br><br>Reduces unit complexity of individual DTs<br><br>Allows for multi-scale system representation<br><br>Eases integration | Maintainability<br><br>Portability<br><br>Performance efficiency<br><br>Reliability |
| Redundancy | Modularity<br><br>Self-similarity | Allows for larger, more adaptable systems<br><br>Improves expansibility and reconfigurability<br><br>Allows for solution reuse<br><br>Eases integration | Maintainability<br><br>Portability |

Through the separation of concerns and because of the distributed data acquisition and load distribution, the DT aggregation hierarchy can maintain a single source of truth in the DTIs while providing data to multiple services. Therefore, data-led decision making can be facilitated in multiple services and at multiple levels of the system, while maintaining a degree of data consistency. Furthermore, this also provides for the separation of data and service ownership concerns (Harper *et al.*, 2019). This is vital to building a multi-stakeholder ecosystem (Harper *et al.*, 2019) because requirement conflicts can often be

resolved by separating the system or subsystem and applying different solutions to the separate components (Poort & De With, 2004).

The DT aggregation hierarchy also enables a combination of distributed, clustered and/or centralised processing and/or decision making. This can be used to address issues, such as computational load balancing and latency concerns, while also allowing for centralised data visualisation, analytics and configuration management (Bertoli *et al.*, 2021). Furthermore, it allows for different data granularities in different parts of the aggregation hierarchy. This allows for a simple overview of one subsystem while also allowing for detailed data analysis on another subsystem (Brandenbourger & Durand, 2018). Allowing for different data granularities and levels of complexity is also advantageous for integration since subsystems can be integrated as primitive data sources or sophisticated DTs, depending on their capabilities.

The combination of distributed, clustered and/or centralised processing and/or decision making also improves resource efficiency by making use of multiple devices and computing processes (Villalonga *et al.*, 2020). This is related to the ability to scale horizontally, vertically and in the z-axis (O'Donovan *et al.*, 2015). Reliability is also improved through horizontal and z-axis scalability because they allow for replication and partitioning, respectively (Tovarnitchi, 2019; Villalonga *et al.*, 2020).

The separation of concerns also allows for different development teams (including third-party developers) to contribute to the different concerns in parallel, thereby also allowing for the integration of domain expertise by the different developers (Tovarnitchi, 2019). Furthermore, the different development teams can then independently deploy and maintain the DT(s) related to their concerns (Taibi *et al.*, 2018). The different development teams can also develop in different programming languages, provided that they use programming agnostic interfaces or a programming agnostic communication protocol (Balalaie *et al.*, 2018).

The modular design and self-similarity in DT aggregation hierarchies also allow for larger and more adaptable system representations and services (Ciavotta *et al.*, 2020). The modularity, separation of concerns and distributed architecture also allows for the easier integration of new technologies without disruption to other subsystems (Tovarnitchi, 2019) and these aspects make the system more reconfigurable (Adolphs *et al.*, 2015).

# 6 Overall reference architecture

This chapter presents a reference architecture that provides context for the design framework presented in Chapter 7. The design framework should produce an architecture that resembles the reference architecture presented in this chapter. However, when the design framework is applied to a specific case, variations of the reference architecture are expected. For example, not all the management services may be required or the services network may not need dynamic orchestration.

The overall architecture of the system of DTs is provided in Figure 4, where the DT aggregation hierarchy, the services network and the management services are encapsulated within the overall architecture.

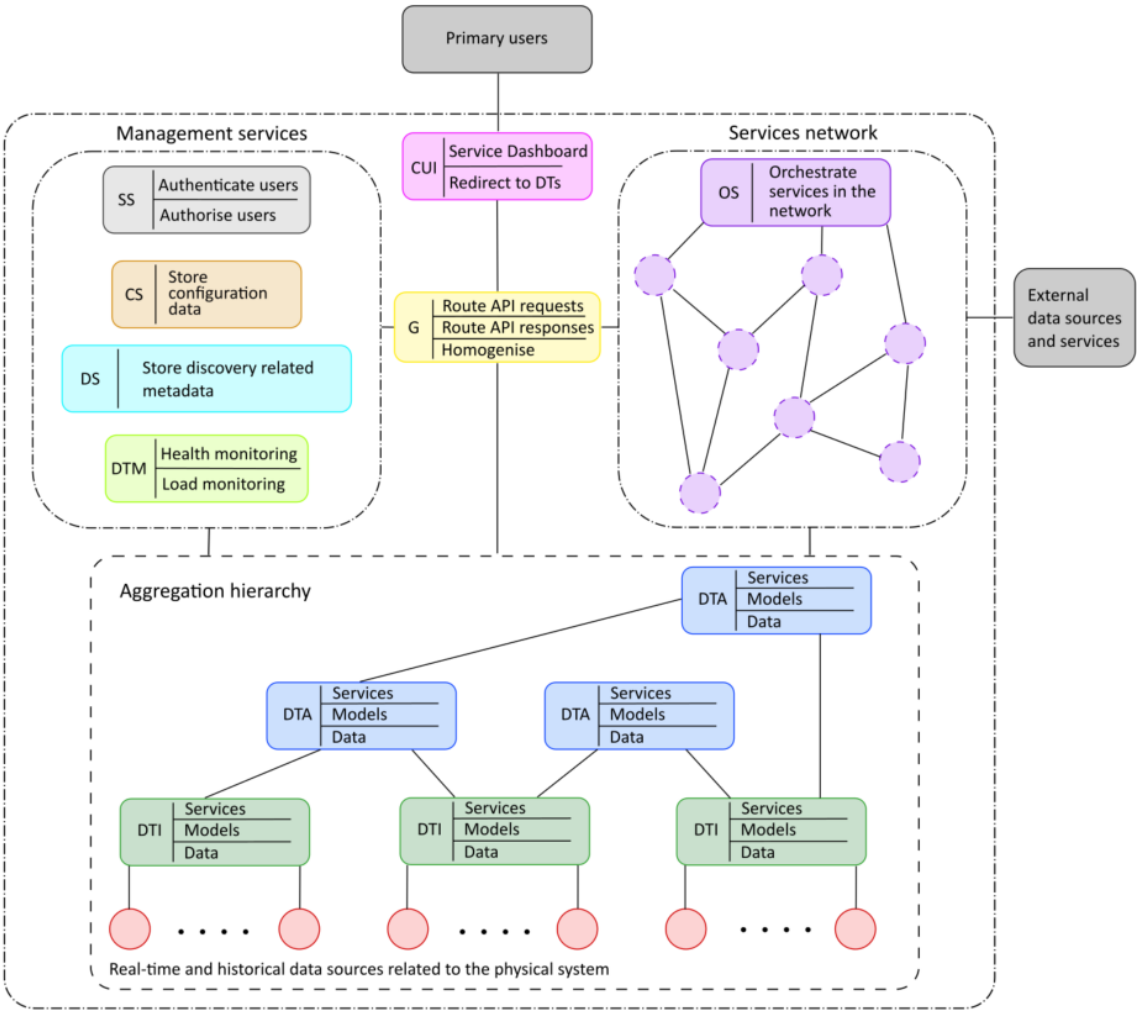



**Figure 4: Reference architecture for the system of digital twins.**

The roles of the main groupings in the architecture are as follows:

- The DT aggregation hierarchy represents the physical system in virtual space, including data capturing and system modelling, as well as providing services that directly pertain to the physical system and its data. The DT hierarchy is designed according to the principles discussed in Chapter 5 and typical services include physical system monitoring and physical system fault diagnostics.
- The service network is intended to provide a more general set of functionally decomposed services to further manipulate and interrogate the data. The service network will likely follow a service-oriented or microservices architecture. The services network also provides an entry point for external services and any data that does not originate from the physical system but that still has a bearing on the management of the physical system. For example, financial data can be incorporated using the microservices architecture and analysed together with production data to inform business decisions.
- The management services are services external to any DT but that still help to handle some of the complexity within the DT aggregation hierarchy. These services are discussed in Section 10.2.

The overall purpose of the system of DTs is to provide a stakeholder with the 1) appropriate level of interaction, from the 2) appropriate viewpoint, with 3) appropriate filtering of information, at the 4) appropriate time (Lutters, 2020). To fulfil its purpose, the system of DTs must 1) provide the right services with adequate security and regulation, 2) provide the right span of reality to the services, 3) aggregate the data appropriately and 4) respond to inputs in a timely manner.

Furthermore, the combination of the digital twin hierarchy and the services network is intended to support two (often conflicting) quality attributes, namely reliability and agility (Section 4.5.1 discusses the conflict between reliability and agility). The reliability of the system is supported by the digital twin aggregation hierarchy. The digital twin hierarchy represents a stable physical reality, of which the functionality, behaviour and possible interactions are likely to change slowly and infrequently – promoting the development of robust, reliable software. In contrast, the services network aims to satisfy dynamic user requirements and, as such, should be developed to be adaptable and agile. The services network enables faster provisioning of services as user needs change or as new users are integrated into the system. It also allows for optimized hardware utilization and dynamic resource allocation as services experience varying demand.

# 7 Design framework

## 7.1 Objectives of the design framework

The design framework presented in this dissertation aims to enable systematic, effective decisions when designing a system of DTs to represent a complex physical system. In particular, this framework adopts hierarchical aggregation as one of its primary enablers and combines it with a services network, as discussed in Chapter 6. However, the focus of the design framework is the design of the DT aggregation hierarchy according to the principles discussed in Chapter 5.

Furthermore, the framework aims to be broadly applicable to various DT application domains and, as such, does not focus on the needs of any particular complex system, but rather focusses on general complexity issues. The framework also aims to be vendor neutral, avoiding the prescription of any specific technologies, because complex systems are likely to involve multiple vendors and typically software related technologies change rapidly.

The framework enables traceability from user needs and complexity considerations to architectural and implementation decisions. By enabling this traceability, design choices can be mapped to the needs they intend to fulfil. This makes it easier to determine which needs have not been met and allows for better change management when a need, implementation technology, etc. changes. Finally, the framework also intends to provide a common set of terminology to allow for better development team cooperation.

## 7.2 Design framework overview

An overview of the system of DTs design framework is provided in Figure 5, where the high-level design steps are divided into various blocks and the design steps are further decomposed within each block. Furthermore, the outcomes of the design steps are summarised next to the arrows leading to the next step. Within each block, iteration and interaction should be expected. Although it is desirable to progress from the first to the last high-level step without returning to a previous step, such iterations are usually unavoidable in practice.

Each of the design steps are discussed in more detail in the following subsections. To help clarify the concepts and the intended outcomes of each step, the heliostat field case study will be used as a running example. The running example presented in this chapter is an excerpt from the heliostat field case study presented in Section 13.1.

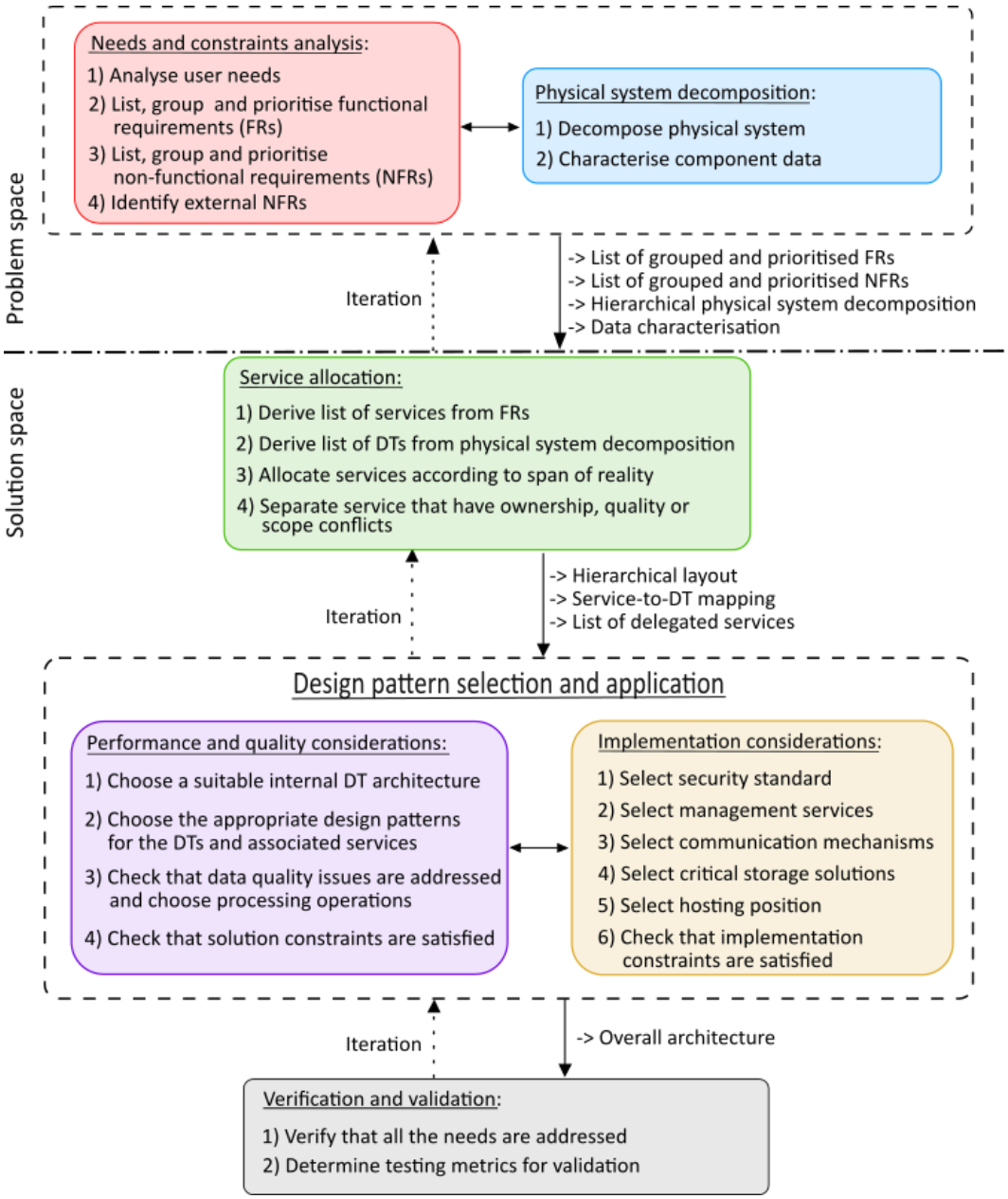



**Figure 5: Overview of design framework for complex DT system design.**

## 7.3 Needs and constraints analysis

This step involves the analysis and translation of user defined needs, as well as derived needs, into requirements. The requirements are then grouped. This step of the design framework relates to Chapter 4 which defines all the terminology related the requirements.

The needs and constraints analysis step is tightly coupled with the physical system decomposition since many of the derived needs are identified during physical system decomposition. The needs and constraints analysis along with the physical system decomposition are together considered the problem space. The problem space refers aspects of the design that are given and thus the system designers do not have control over such aspects. The decomposed design steps are:

- Analyse user needs to translate the user needs into FRs or NFRs. This step also involves the identification and translation of derived needs into requirements. Derived needs here refer to any needs that are not explicitly defined by the user but have been identified as necessary to satisfy the user defined needs. The physical system decomposition (discussed in Section 7.4) is a common source of derived needs.
- List the FRs and group them into primary and secondary functional requirements. Prioritise the two groups of functional requirements using the classification of mandatory, highly desirable, desirable or optional (Bourque & Fairley, 2014). The prioritisation of the functional requirements is important for project planning, management and deployment and it helps to make trade-off decision to, for example, save cost or development time (Poort & De With, 2004).
- List and group the NFRs into quality requirements and development constraints. Determine how the constraints may affect the final design and which of the quality attributes should be prioritised for the case.
- Determine what external NFRs, if any, should be considered.

Outcomes of design step: List of prioritised primary FRs, list of prioritised secondary FRs, list of prioritised quality attributes, list of development constraints with their implications, list of external NFRs and their implications.

With reference to the heliostat field, Table 12 is an example of a list of grouped and prioritised FRs. Similarly, Table 13 is an example of NFRs with their implications.

**Table 12: Functional requirements for the heliostat field (excerpt)**

| High-level functional requirements | Rationale | Group (Primary or secondary) | Priority |
|---|---|---|---|
| Remote monitoring | The status of individual heliostats, as well as the status of subsections of the field need to be presented to a user. | Primary | Mandatory |

| High-level functional requirements | Rationale | Group (Primary or secondary) | Priority |
|---|---|---|---|
| Event logging | Considering that the heliostat field can be controlled automatically or by a user, it is considered good practice to log events related to the heliostat field control. The best practise is related to maintenance for debugging purposes and to security for accountability and non-repudiation. | Secondary | Mandatory |

**Table 13: Non-functional requirements of the heliostat field (excerpt)**

| **Need** | **Provide for large amounts of data. (Related to N24)** |
|---|---|
| **Related NFR** | Performance efficiency |
| **Rationale for NFR** | Considering the size of the heliostat field, the amount of data generated by each heliostat and the potential resource constraints, there is a need to handle a large amount of data efficiently. Therefore, resource utilisation, scalability and high throughput are primary concerns and these are sub-characteristics of performance efficiency. |
| **NFR grouping** | Quality attribute |
| **Implication of NFR** | Use performance efficiency design pattern |
| **Need** | **Allow for retrofitting and integrate with existing information systems. (Related to N6 and N7).** |
| **Related NFR** | Solution constraint and implementation constraint |
| **Rationale for NFR** | Solution constraint - The consultants at STERG are responsible for designing the heliostat field and its accompanying control systems. Therefore, the digital twins must be able to integrate with the heliostat field as if it were being retrofitted onto an existing system.<br><br>Implementation constraints - The Helio100 field makes use of a local PostgreSQL database that serves as the current primary data source of all historical data. Therefore, there is a preference to use PostgreSQL because the current engineers are familiar with it. |
| **NFR grouping** | Development constraint |
| **Implication of NFR** | Some of the technologies related to the data acquisition are predefined and must be integrated with. There is a preference for PostgreSQL as a database. |

When the entire list of NFRs was analysed and the stakeholders were consulted again, it was determined which quality attributes are most important for the proper functioning of the heliostat field. The heliostat field does not have any identified external NFRs. However, an example of an external NFR could be the Department of Energy that requires the storage of data related to energy production for a minimum of seven years.

Furthermore, the examples presented in this dissertation are high-level needs and requirements for the sake of brevity. However, in practice it is likely that these needs and requirements may be defined in more detail, where each need can include more details and sub-needs. For example, the implication of the implementation constraint listed in Table 13 could include details of specifically which technologies must be integrated with. The need for remote monitoring could also be decomposed further if necessary.

## 7.4 Physical system decomposition

This step involves the decomposition of the physical system and the characterisation of the data that is expected to be available within the physical system. Furthermore, during physical system decomposition, it is likely that some needs will be derived to accommodate the complexity identified within the system. These derived needs typically resemble the needs presented in Chapter 3. The decomposed design steps are:

- Decompose the physical system, most likely according to spatial proximity and interface complexity to represent the system effectively.
- Characterise the available data of the decomposed elements and/or subsystems. This entails determining what data is available, in what format it is and how it can be accessed. This is intended to help with the contextualisation and integration of the data (Kuhn *et al.*, 2020; O'Donovan *et al.*, 2015). O'Donovan *et al.* (2015) warn that this data characterisation process can be challenging and the effort-to-benefit ratio is often perceived as low. However, this phase is necessary and, if done well, the effort related to subsequent data integration and contextualisation is greatly decreased.

Outcome of design step: Hierarchically decomposed physical system diagram and data characterisation

An example physical system diagram is given in Figure 12 in Section 13.1.3 and Table 14 is an example of the span of reality of a single heliostat. The span of reality includes a physical scope, a data characterisation, a communication mechanism and constraints and considerations. The data features within the data characterisation were defined according to the following schema: <data feature> - <number of observations if there is more than one>, <data type>, <data range>,

<(units)>, <frequency of observations>. This is an example of a highly detailed characterisation that would typically be used during implementation.

**Table 14: Span of reality of a single heliostat.**

| Physical component | Heliostat with local control unit (LCU) |
|---|---|
| **Physical system scope** | Individual heliostat. |
| **Data characterisation (Data granularity) of data recorded/ generated by physical component** | Stepper motor positions – two, int, between 0 and 200 000, (step count), generated every minute.<br>Battery value – float, between 5.5 and 8.2, (Volt), generated every minute.<br>Timestamp – datetime, N/A, (N/A), generated every minute. |
| **Data characterisation (Data granularity) of data sent to physical component** | Local coordinates of the sun – See CCU (presented in case study in Section 13.1.3).<br><br>Translated operator control commands – details unknown. |
| **Data format** | JSON formatted message. |
| **Communication** | Radio frequency (RF) communication using a serial bus. |
| **Considerations and Constraints (Capacity for interaction)** | LCUs are power constrained and thus the activity of the LCUs need to be minimised.<br><br>The LCUs can only support RF communication.<br><br>The design requires 10 002 individual heliostats and they may differ slightly in composition (e.g. newer heliostats make use of newer components and future heliostats may have more sensors). |

## 7.5 Services allocation

The services allocation marks the start of defining the solution space. The solution space refers to aspects of the design that are within the system designers' control. This step involves assigning services to DTs or to the services network based on the span of reality (discussed in Section 5.2.1.1) requirements of the services, as well as the intensity of interaction (discussed in Section 5.2.1.2). The service allocation step is discussed in detail in Section 8.

The decomposed design steps are:

- Derive a list of services from the functional requirements and determine the required span of reality of each service. It should be noted that a service can address more than one FR and more than one service can address an FR. The service patterns listed in Table 15 in Section 8.1 can be used as a reference for possible services.

- Derive a list of DTs from the hierarchically decomposed physical system diagram and span of reality characterisation. This provides a list of possible DTs that reflect the decomposed physical system. Section 8.2 provides guidelines when determining the scope of DTIs and DTAs.
- Using the list of services and the list of possible DTs, assign services to the DTs by mapping their span of reality to each other. Service should initially be assigned to the lowest-level DT that has the data required for the service. Services may also be assigned to the services network portion of the overall reference architecture (discussed in Chapter 6). Services that are assigned to the services network are referred to as the delegated services.
- Separate services that have ownership or quality conflicts or separate services when the DT's scope and complexity become difficult to manage.

Outcome of design step: The service-to-DT mapping, a hierarchical layout of DTs and a list of delegated services. The service allocation is primarily concerned with functional allocation, i.e. determining what services will be provided by which components.

In the example of the heliostat field, it was determined that a mirror service (as described in Section 8.1) is required to fulfil the remote monitoring FR. For the heliostat field, the mirror service can be applied to multiple levels of the system. The mirror service described here will only consider the span of reality required for individual heliostats, but the full span of reality description is available in Section 13.1.4. The mirror service's span of reality is:

**Mirror service:**

Description: The status of individual heliostats must be presented to a user.

Related primary functional requirements: Remote monitoring.

Related secondary functional requirements**:** Log files of events.

Required physical scope**:** Individual heliostats.

Required data granularity:

Individual heliostat scale:

- Data features: LCU level - Motor position values, battery values. CCU level – heliostat status values.
- Timescale: All data features should be measured at one-minute intervals.

Service characteristics:

- Required data update frequency: Real-time
- Degree of user interaction: Remote monitoring – periodic user interaction
- Intensity of interaction: Spatially focussed service
- Persistence: Persistent data gathering.

Constraints and considerations: Remote monitoring requires access from an external network and thus the service must either be cloud hosted or it must allow for direct local network access, such as through a VPN or SSH connection. The data throughput may become a critical factor in a large field.

Furthermore, based on the physical system decomposition it is determined that the DTIs should represent CCUs because the LCUs of the individual heliostats are resource constrained and thus do not have capacity for interaction. A DTA would represent the FCU. Both a DTI and DTA would have the right span of reality to host the mirror service detailed above. Therefore, the mirror service is initially assigned to each DTI, since the DTI is the lowest level DT that has the right span of reality. In this example, there is no separation of services because there is only one service.

## 7.6 Performance and quality considerations

This step involves making decisions regarding the aggregation and architecture to achieve the data quality requirements, as well as the desired system-level and service-level quality attributes. This step is closely linked to the implementation considerations discussed in Section 7.7. The performance and quality considerations are discussed in detail in Section 9. The design patterns presented in Chapter 11 are intended to simplify the architectural choices.

The decomposed design steps are:

- Choose a suitable internal architecture for the DTs. This dissertation makes use of SLADTA as discussed in Section 2.3.
- Determine what the dominant quality attribute(s) is/are for each DT and its associated service(s) and consider performance related architectural choices according to the identified quality attribute. Section 9.1 discusses aspects related to performance, while the design patterns (presented in Chapter 11) are intended to simplify the aggregation and architectural decisions to provide for the identified quality attributes.

- Check that the implemented architecture accommodates any data quality and management issues as described in Section 9.2.1 and choose the required granularity related processing operations as described in Section 9.2.2.
- Check that the solution constraints (identified during the needs analysis step) are satisfied.

Outcome of design step: Architectural design choices. The performance considerations step is primarily concerned with determining whether the quality requirements are being met by the architecture.

For the example heliostat field, the mirror service requires high data throughput and thus the performance efficiency design pattern is chosen. This includes choices such as distributing the processing load, using pre-storage aggregation, using local network aggregation and performing stream processing. Data persistence is likely to be an issue in the heliostat field and thus in addition to a local data store (for operational performance) a cloud-based data store is also advised. Furthermore, the system of DTs must be retrofitted onto the heliostat field and thus some infrastructure is already specified. No replication or partitioning is required yet, but the DTA of the FCU might need to be partitioned.

## 7.7 Implementation considerations

This step involves making recurring and important implementation decisions that significantly affect the data and system quality. This step is closely linked to the quality and performance considerations discussed in Section 7.6. The implementation choices are discussed in more detail in Chapter 10. The design patterns presented in Chapter 11 are intended to simplify the implementation choices. Furthermore, such implementation choices are usually made with reference to an internal DT architecture. In this dissertation the SLADTA was chosen for the internal DT design as discussed in Section 2.3.

The decomposed design steps are:

- Select the security standards (this choice will impact all following choices).
- Select management services.
- Select communication mechanisms and standards.
- Select important storage solutions (solutions where a specific type of storage is important).
- Select hosting position.
- Check that implementation constraints are satisfied.

Outcome of design step: Implementation decisions

With regards to the heliostat field, the performance efficiency design pattern is being used. For the heliostat field, standard TLS security is advised and no management services are recommended. A message-oriented middleware is recommended for communication, a time-series database would be well-suited for operational storage and a NoSQL database provides scalability for the long-term storage. Local hosting with containers is preferred where possible.

However, the heliostat field has resource constrained devices and thus a more lightweight communication mechanisms is preferred. Therefore, a lightweight publish-subscribe protocol would be well suited because publish-subscribe is also highly scalable. Furthermore, the heliostat field engineers are already using PostgreSQL as a database and thus a PostgreSQL database is used instead of the recommended timeseries database. Therefore, in context of SLADTA, the short-term (Layer 3), local data repository is a PostgreSQL database, while the long-term database should be a NoSQL database. The IoT gateway (Layer 4) will make use of a publish-subscribe messaging protocol with SSL/TLS for security.

## 7.8 Verification and validation

This step is concerned with verifying that all the needs have been addressed and validating that the system does indeed satisfy those needs. The decomposed design steps are:

- Verify that the implemented system or subsystems addresses all the needs that were identified in the needs and constraints analysis.
- Determine quantifiable metrics (technical performance measures) for the systems and subsystems to allow for validation after implementation.

Outcome of design step: Detailed system architecture and accompanying documentation for traceability of the needs and validation of the requirements.

# 8 Services allocation

This chapter discusses the services allocation step (introduced in Section 7.5) of the design framework in more detail. The following sections each discuss a step within the services allocation.

## 8.1 Service patterns

When considering what services can derived from the user needs, the eight *service patterns*, identified by Erikstad & Bekker (2021), can be considered. The term service pattern is used by the authors because they describe general aspects of the services which can be applied to numerous contexts. These service patterns, listed in Table 15, have been included here to help with the service identification step within the services allocation.

**Table 15: List of service patterns for DT services. (Adapted from Erikstad & Bekker, 2021)**

| Name | Description |
|---|---|
| Virtual sensor | Infer sensor feeds from digital model to provide data that is not captured in the real world. This provides compensation when sensor placements are limited due to cost, access, hazardous environments, etc. |
| Context sensor | Provide insights into operational context by inverse inferences from asset response measurements. This allows for load estimations where load is not measured directly because of limitations. |
| Fingerprint | Recognise operational response of real asset based on a catalogue of behavioural patterns that were pre-generated within the digital model. This can be used to pre-empt failures of critical assets. |
| Anomaly | Detect abnormal behaviour by contrasting data from live sensor feeds with data from trusted digital models. This can provide notifications in case of anomalies and help to understand what behaviour should be expected. |
| Root cause | Determine the reason for asset response deviation. This is typically done through a combination of physics-based simulations, as well as real-time sensor readings. |
| Scout | Simulate future behaviour of an asset to inform decision making. This typically uses physics-based simulations, as well as data-based models, such as machine learning models. |
| Life counter | Track stresses incurred by an asset to determine remaining useful life and/or to prescribe maintenance. This can be used to reduce uncertainties related to prognostics. |
| Mirror | Manage assets remotely through an immersive operators' experience. This allows for more immersive and informed decision making from remote locations. |

## 8.2 DTI and DTA scope identification

This section provides guidelines to determine the scope of a DTI and a DTA (as defined in Section 2.3). Therefore, this section helps identify applicable DTs to adequately represent the physical reality of interest. This relates to the DT identification step of the services allocation.

Drawing from the discussion of span of reality in Section 5.2.1.1, the DTI has the finest grained data about a certain subset of the physical reality, but also the smallest physical scope. DTIs are therefore located at the lowest level of the aggregation hierarchy. A DTI may reflect a complex physical entity that forms part of a larger complex system, but it should remain the source of the finest grained data of that physical entity. Therefore, a DTI can reflect any physical entity (simple or complex), where the scope of the DTI is determined by the granularity of the data required by the models and services within the DTI. Furthermore, the physical entity being reflected may include environmental data, such as ambient temperature, provided the environmental data is not already being captured elsewhere.

In continuous systems where there are no physical divisions in the physical system, such as water distribution systems or railway systems, the physical system should be divided according to the respective concerns of the end-users or manageable parts, such as district water distribution networks or sectors of railway line. These continuous physical systems can then be represented by a number of DTIs that represent similar, but still distinct, physical realities. These DTIs will be more interdependent and exchange data among themselves, either directly or through an aggregate.

In contrast, a DTA can be located at various levels within an aggregation hierarchy, but not at the lowest (DTI) level. As a DTA is located higher in an aggregation hierarchy, its physical scope increases, but typically its data granularity decreases (refer to Section 5.2.2 for a discussion of this trend). Different DTAs on the same level of aggregation can also reflect different aspects of reality by 1) aggregating different DTs, 2) by aggregating different features from DTs and 3) by processing the data differently. The span of reality of a DTA is determined according to the data requirements of the associated models and services, where services are typically focussed on a spatially distinct subset of the physical system (refer to Section 5.2.1.2 for spatially versus functionally focussed services).

Furthermore, the more assets are represented by a DT, the more complex the problems that can be addressed by the DT (Kuhn *et al.*, 2020). Aggregation allows for the combination of DTs and thus DTAs are generally used for more complex decision making with regards to the physical system. Therefore, when models or services require such a combination of other DTs, a DTA should be used. This is

likely to become increasingly applicable as the system evolves and needs to host new services with their associated DTs. However, to keep the DTAs manageable, Moyne *et al.* (2020) suggests that aggregation membership should be specified in terms of purpose and the associated span of reality. This means a DTA must have a clear purpose and span of reality and should only aggregate the necessary data for decision-making, with the required sampling frequency, to meet its purpose (Villalonga *et al.*, 2021). When the scope of an DTA's purpose becomes too large and general, a DTA can quickly become too complex due to the amount of data available from the lower levels of aggregation.

The number of aggregation levels depends on the complexity of the system, where higher levels of complexity generally require more layers to represent reality efficiently (Villalonga *et al.*, 2021). The level of aggregation depends on the degree of detail required for decision making and it depends on the efficiency requirements, such as storage space and response times (Fadlalla, 2005).

In summary, the scope of a DTI or DTA is limited by its unit complexity. DTIs and DTAs may contain multiple models and services, but if maintaining the DT becomes too difficult, it may be desirable to partition the DT. Furthermore, if the DT's capacity for interaction is too limited, the performance of the DT will degrade. Therefore, the scope of any DT must be large enough to accommodate the related models and services, but small enough to remain responsive and manageable.

## 8.3 Services in digital twins vs the services network

This section provides some considerations when allocating services to DTs and to the services network. These considerations are in addition to the intensity of interaction as discussed in Section 5.2.1.2.

In systems where different span of reality requirements exist, two broad approaches can be followed to gather the data for the right span of reality: data warehousing or data federation (Pathak, Jiang, Honavar, *et al.*, 2006). Data warehousing refers to the collection, transformation and storage of the relevant data in a common format, that can then be queried for decision making. Data federation refers to the collection and transformation of the relevant data as a query is made by a service.

DTs tend to build a data warehouse and thus a span of reality related to a particular asset or a group of spatially related assets. However, services in an SOA are functionally focussed and thus data federation and orchestration are often used to gather data as needed and sent to the appropriate services to be processed as required. Therefore, services within DTs tend to have a specific purpose with a specific data requirement and typically with stricter quality

requirements, while services in an SOA have a more general purpose and are generally also more adaptable to user needs.

Domains such as manufacturing tend to prefer the services in DTs approach because DTs are persistent and more dedicated to an asset or group of assets, i.e. they are useful for real-time and persistent services, such as automatic process control and fault detection (Ciavotta *et al.*, 2020; Moyne *et al.*, 2020; Therrien *et al.*, 2020). On the other hand, services in SOAs tend to be preferred for more general and longer-term decision making that is only periodically required. For example, long-term business planning using general data analytics. The services mentioned in Section 8.1 are strongly related to physical assets and thus they are also considered spatially focussed services. DTs are well suited to hosting spatially focussed services because DTs also follow spatial decomposition.

In summary, DTs are generally the preferred host for services when services 1) are spatially focussed, 2) have a specific purpose, 3) are persistent or periodically invoked, 4) require real-time data and 5) have strict latency, throughput or reliability requirements. If services are 1) functionally focussed, 2) more general, 3) periodically invoked (typically less frequently than periodically invoked services in DTs) or event-driven and 4) only require historical data with no strict service requirements, then the services network may be preferred.

## 8.4 Separation of conflicting services

This section discusses the separation of services according to ownership (8.4.1), scope complexity (8.4.2) and dominant quality attributes (8.4.3). These three reasons for services separation are part of the final sub-step of the services allocation step.

### 8.4.1 Separation according to ownership

Separation according to ownership is the simplest reason for separation to understand, but it can be complex to handle. The premise is to separate DTs according to ownership of the data, models and services. For example, in the context of city management, the Centre for Digital Built Britain (CDBB, 2018) state: “Each infrastructure owner or operator is likely to want DTs to improve the management of their own assets.” Therefore, DTs and their associated services are separated according to ownership and appropriate data sharing is facilitated between these DTs. However, challenges arise when issues related to intellectual property and competitive advantage hinder data sharing (this issue is discussed as a multi-stakeholder complexity need (N2) in Section 3.2)

### 8.4.2 Separation according to scope complexity

The goal of a service is to provide a comprehensive set of data related to an area of decision making, while the service must remain easy to use (Therrien *et al.*, 2020). Therefore, one of the primary goals of the DT is to provide the service with the data that it needs. Some use cases require high levels of detail and high fidelity models for decision making, while other use cases may not (VanDerHorn & Mahadevan, 2021). Therefore, tailoring the span of reality of the DT to the service can help make the service comprehensive while also maintaining usability. The intention is to meet the intended outcomes without adding unnecessary complexity or cost that may compromise the feasibility of the DT (VanDerHorn & Mahadevan, 2021).

For example, in the context of manufacturing, Villalonga *et al.* (2020) classified DT modelling and decision making into three main levels of detail: 1) local, 2) system and 3) global, according the system being represented. Local represents the dynamics of the equipment pieces in the production lines; system considers the interaction between the equipment pieces that make up the production line; and global replicates the behaviour of the entire shop floor production. Only data needed for the decision making at the upper levels are sent to those levels.

The required timeframe and data update frequency are components of a service's span of reality. The timeframe and update frequency of various data features is largely determined by the potential rate of physical system change captured by the data feature, as well as the decision-making frequency. Typically, the physical system change is described using models and decisions are made based on such models. Therefore, the update frequency is determined by the DTs associated models. Lamb (2019) also distinguishes between dynamic digital models and static digital models. Dynamic digital models capture and react to real-time data to, for example, perform control functions. Static digital models periodically update long-term data and are typically used for strategic planning.

Furthermore, the decision-making frequency can be classified as real-time, periodic or event-driven. It is also important to recognise that 'real-time' is context specific and it depends on the frequency of data required to make effective decisions. For example, for control applications such as motor speed control, this frequency is very high (data needs to be sampled multiple times per second) because the physical system can change rapidly and the related control decisions must react faster than the change of the system. However, for the control of a heliostat's position, this frequency is very low (only one sample per minute) because heliostats only adjust position once per minute. Further, in the context of power transformers, operational decisions are made on a hour to week timeframe, maintenance decisions are made on a week to year timeframe and planning decisions are made on a year to 10 year timeframe (Pathak *et al.*, 2006).

Ultimately, the services form part of the scope of a DT and if the DT's scope becomes too complex, it may be desirable to partition the DT. Section 8.2 discussed DT scoping considerations.

### 8.4.3 Separation according to dominant quality attributes

Different services may have different quality requirements and often these requirements can be conflicting. One of the best ways to deal with such requirement conflicts is to separate the services (Poort & De With, 2004).

For example, consider a high-value or critical asset within a physical system. There may be a fault detection service related to the asset that needs to be reliable. There may also be a need for data analytics related to the high value assets which emphasises agility. Even though these services rely on the same span of reality, their quality attributes are in conflict and thus is may be better to separate the services. Section 4.5 provides a discussion on some of the most cited quality attribute conflicts.

# 9 Performance and quality considerations

This section discusses the performance and quality considerations step (introduced in Section 7.6) of the design framework in more detail. Specifically, this section considers aspects related to the aggregation hierarchy's performance in Section 9.1, as well as aspects related to the management and quality of the data within the hierarchy in Section 9.2. Some aggregation alternatives are then provided in Section 9.3 to help manage the trade-off of performance versus data detail and quality. This trade-off is discussed in Section 4.5.4 and the trade-off is also captured within the span of reality where performance efficiency is related to the capacity for interaction, whereas the data detail and quality are related to the physical scope and data granularity.

## 9.1 Performance efficiency considerations

Regardless of the dominant quality attribute of a DT and its associated services, the performance of the DT must still be acceptable for a good user experience. Therefore, this section considers aspect related to the performance of a DT.

Performance efficiency is sub-divided in the ISO 25010 standard (BSI *et al.*, 2011) into time behaviour, capacity and resource utilisation. These sub-divisions can be quantified as latency, throughput and infrastructure measures, respectively. Table 16 provides a summary of the performance efficiency sub-division and the associated performance measures, which are further considered in the subsections that follow the table.

**Table 16: Performance efficiency breakdown**

| **Performance efficiency sub-division** | **Performance measures** | **Dimensions of performance measures** |
|---|---|---|
| Time behaviour | Latency | Intermediary communication time: Transmission, I/O and processing time |
| | | Number of intermediary communications |
| Capacity | Throughput | Message size |
| | | Message frequency |
| | | Number of parallel message streams |
| Resource utilisation | Infrastructure measures | Network: network speed, network bandwidth, network availability |
| | | Computation: CPU, GPU, memory |
| | | Storage: I/O speed, storage capacity |

### 9.1.1 Latency considerations

Latency is the amount of time it takes for a message to be captured, transmitted, processed and received when sent from a source to a destination.

When considering communication from a sensor (source) to where a decision can be made, such as a user interface within a cloud platform (destination), the communication can be broken into intermediary communications (O'Brien *et al.*, 2007), where each intermediary communication consists of data input/output (I/O), transmission and processing operations. For example, intermediary communications can be from sensor to local storage, from local storage to cloud storage and then from cloud storage to user interface. There might also be additional communications involved, such as communication through a service gateway or intermediary communications between services. Therefore, when considering latency between a source and a destination, the intermediary communication time and the number of intermediary communications should be considered.

The latency related to each part of an intermediary communication is tightly coupled with the underlying hardware. The I/O latency is related to the storage device's read/write speed, the transmission latency is related to the network speed, as well as the network bandwidth and availability and the processing operation latency is related to the CPU, GPU, and RAM. These hardware aspects are further discussed in Section 9.1.3.

Furthermore, some intermediary communications may be more time critical than others. For example, transfer time from sensor to cloud database may be less important than minimising the transfer time from cloud database to user application. Therefore, optimising the transfer time from database to user application at the cost of transfer time between sensor and database is sometimes desirable. This would typically be achieved by pre-processing the data before it is stored so that less processing is required when the data is queried.

### 9.1.2 Throughput considerations

Throughput is a measure of the amount of data that is transferred between source and destination within a given time interval.

Throughput is influenced by multiple factors such as message size (further influenced by the size of the datapoints and the number of datapoints), message frequency and the number of message streams. The configuration of these factors is often influenced by the latency requirement and the limitations imposed by existing infrastructure, as well as the number of data sources.

### 9.1.3 Infrastructure considerations

The computing and network infrastructure can be considered a limitation in cases where the infrastructure is already installed and when DTs are being retrofitted onto the physical system. In other cases, however, the DT developers may have freedom to choose some or even all the infrastructure components. Therefore, it is important to establish what infrastructure is already available and what infrastructure needs to be added. The considerations in each case, however, largely remain the same and thus this section provides considerations for retrofitting onto existing infrastructure and when choosing new infrastructure.

Infrastructure considerations have here been divided into three subsections: network, computational resources and storage. Alternative methods of infrastructure hosting, such as local hosting vs cloud hosting and the effects of virtualisation, are discussed in Section 10.5.

Network considerations include:

- Network bandwidth: The upload and download capacity. This is the maximum amount of data that the network can transmit per second, typically measured in megabits per second (Mbps).
- Network speed: The transmission speed between network nodes, which has a significant impact on transmission latency. Network speed is typically measured through ping messages. Network speed is dependent on factors such as (Kajati, Papcun, Liu, *et al.*, 2019): 1) transmission medium, 2) the physical distance between nodes (this plays a significant role when using cloud services), 3) the number of network relays (such as routers or switches), 4) internet service provider (for external connections) and 5) cloud platform and cloud offering (e.g. throttling limits imposed by a certain cloud service). Time of day and the day of the week have negligible impact on network latency (Kajati *et al.*, 2019). Another factor to consider is the repeatability of the messaging latency, where the standard deviation of the latency values may be too great even though the mean value may be acceptable (Kajati *et al.*, 2019). Standard deviation is particularly applicable when considering applications that require high reliability.
- Network availability: The fraction of time that the network can be used by the various nodes within a given timeframe. A stable and continuous network connection may not always be available, for example when using wireless connections or when devices are mobile and move between connection points. Therefore, the amount of time that a device is unable to connect to the network contributes to the latency of the communication. Intermittent network connections also require additional provisioning for asynchronous communications, such as message queues and message patterns that acknowledge message delivery, to prevent data loss.

Network connectivity may also be challenging when designing for physically dispersed systems that require multiple access points. In such cases, network availability may be a limiting factor and it may also limit some of the aggregation options. For example, local network aggregation may not be feasible for physically dispersed systems.

Computational resource considerations include CPU, GPU and memory considerations. The computational resources are related to the processing latency between source and destination as mentioned in Section 9.1.1. Therefore, when dealing with processing heavy workloads, such as workloads that require extensive image processing, computational resources become a critical factor to reduce latency. Furthermore, processing latency can be significantly increased by data format conversions, for example when parsing, validating and transforming text-based data formats (O'Brien *et al.*, 2007) and when encrypting and decrypting data.

Storage considerations include the read/write speed of the storage drive and storage space. I/O latency is heavily influenced by the storage drive's read/write speed as well as the database type and the database management system. Furthermore, the required storage capacity must also be considered.

## 9.2 Data quality and detail considerations

The data detail is described by the data granularity and it is directly proportional to the physical scope being represented. However, having detailed data that is of poor quality does not help and thus data quality is also considered here. Section 9.2.1 highlights some data quality and management related considerations that influence how the data may be aggregated, while section 9.2.2 considers processing operations that form part of the aggregation process to produce the desired data granularity.

### 9.2.1 Data quality and management considerations

The data quality and management considerations are subjective by nature and thus they are case specific. However, there are some general guidelines about how these considerations can influence the aggregation strategy, as well as the performance and data quality within the DTs.

Data veracity refers to the trustworthiness of the data. It is generally unwise to aggregate data that is trustworthy and accurate with data this is low quality. For example, in healthcare, Lutze (2019) mentions that digital trust should be established between user and digital system and thus it is important to determine if high-quality clinical data should be combined with behavioural and biometric data from smart wearables.

Furthermore, the difference in veracity as a result of the level of processing of the dataset should be considered. For example, a raw data dataset, a cleaned dataset and a summarised dataset have different levels of veracity. The data veracity is also highly influenced by the method of collection and if the data collected is of poor quality it will be detrimental to the service(s) being rendered (Therrien *et al.*, 2020). The pre-processing of data can help to mitigate shortcomings of poor collection, but only to a point (Therrien *et al.*, 2020).

Data worth refers to the importance of the data. Data with high worth may justify more data reliability measures, such as duplication and redundancy, within the aggregation hierarchy. Factors that influence the worth of the data could be aspects such as 1) number of sources (data with fewer sources may have higher worth), 2) the frequency at which the data is recorded (less frequently recorded by have higher worth), 3) the likelihood of a particular data point value (in some services such as anomaly detection, anomalous values should not be aggregated away) and 4) long-term availability of the data (when data is collected through a third party, it may be useful to duplicate the data for long-term availability).

Data accuracy refers to how closely the captured data reflects the actual data about the system and this is an important aspect of DT fidelity, i.e. how closely the DT reflects reality. Fidelity is significantly influenced by data granularity (Brandenbourger & Durand, 2018), but finer detail does not always produce more accurate models/ predictions. For example, aspects such as sensor measurement accuracy influence the fidelity as well as modelling aspects, such as overfitting or underfitting data.

The Centre for Digital Build Britain (CDBB, 2018) states that a DT must represent reality at a level of accuracy suited to its purpose and this depends on 1) the accuracy of the data, 2) the fidelity of the models (including the validity of the algorithms and assumptions) and 3) the quality of the visualisation and presentation. Furthermore, Moyne *et al.* (2020) suggests including a parameter for prediction uncertainty and simulation accuracy, such as a probability value.

Data consistency refers to the consistency of the data across multiple instances of the data in use. Without proper data consistency, different stakeholders may receive different data values for the same features of the physical system. Data consistency can refer to strong consistency, where any update to a partition of the dataset is immediately reflected in any subsequent access, or weak consistency, where updates may experience a delay before being propagated through the system (Lindsay *et al.*, 2021).

Ensuring strong data consistency will likely cause an increase in latency because data changes must be propagated through the distributed instances before the data can be accessed again. The severity of the increase in latency is dependent

on the time it takes the data to propagate through the system and thus how many distributed instances there are, how physically far apart the instances are and how the instances are connected.

Data persistence refers to the management of long lived data that also relates to the need to store massive amounts of data (Bourque & Fairley, 2014; Pan *et al.*, 2020). In some domains, the data persistence and life-cycle management of data are dictated by domain related policies and governance (Ismail *et al.*, 2019). Therefore, adequate storage capacity and scalability are important factors when data persistence is important.

Villalobos, Ramírez-Durán, Diez, *et al.* (2020) present a three-level hierarchical architecture to manage persistent data effectively. The architecture follows the multi-temperature data management paradigm, which is a paradigm that tailors storage hardware choices according to the frequency with which data is accessed. An example is to use high performance SSDs for hot (real-time data that requires high read and write speeds) and standard HDDs for warm data (frequently accessed but older data). Furthermore, the architecture proposes the use of a pre-processing service to clean data being transferred from the hot storage to warm storage and a data reduction service is proposed to reduce data when transferring data from warm storage to cold storage. A data management layer is also used to manage the data flows between the different storage units as well as between the storage units and a user.

Data synchronisation refers to the matching of data values as they were recorded. Synchronisation is applicable when data features are generated at different rates. The need to cross-validate sensor readings of the same parameter and the need to draw correlations between parameters means that the data must be synchronised. The synchronisation of data can cause increases in latencies because when data needs to be matched in time, the slowest sampling frequency dictates the latency of the synchronised data.

Data inference and interpolation relates to the observation that all automatically captured and treated data is likely to contain gaps and dealing with these gaps is important (Therrien *et al.*, 2020). There are various strategies, such as interpolation techniques or model-based gap filling. Depending on the methods used to infer data, the computational demand placed on the system may be significant.

Data heterogeneity relates to the heterogeneity of the data with regards to various aspects. These aspects are:

- Data structure - Data can be structured, semi-structured or unstructured and some processing may be required to make data more structured.

- Data features - Data features can differ in data type, as well as the units that they use and some conversion may be required. This also includes heterogeneity of the level of processing of the data points. For example, sensor signals (voltage value), sensor readings (bit-value, such as an 8-bit value of 0-255) sensor unit value (e.g. temperature in Degrees Celsius).
- Data format - Data format conversion may be necessary, especially when working with different types of databases. For example, converting data between XML, JSON, CSV or text formats.
- Communication protocol - Some nodes within the network might need to utilise more than one communication protocol or, alternatively, an intermediary may be required to translate messages from one communication protocol to another.

### 9.2.2 Granularity related processing operations

The desired data granularity is achieved during aggregation through a combination of the following processing operations:

- Data removal such as the removal of redundant data and invalid data: Data removal helps alleviate data persistence and storage issues (Pan *et al.*, 2020) and contributes to smaller transmission payloads. Data removal also helps alleviate network strain and downstream processing strain (Huang *et al.*, 2020).
- Data cleaning such as missing value handling and outlier detection and handling: Data cleaning methods typically increase the data veracity but at the cost of increase processing time.
- Data homogenisation: This includes aspects such as structuring data into a common format (Pan *et al.*, 2020) or receiving data from multiple protocols and outputting it in one protocol (Huang *et al.*, 2020). Data homogenisation requires data and protocol conversions that increase the processing time.
- Data summation, such as calculating descriptive statistics (minimum, maximum, average, and standard deviations etc.) (Huang *et al.*, 2020): Data summation can also entail the structured summation of unstructured data, for example, provide structured, text-based data that describes the contents of an audio file. Data summation can significantly reduce downstream latencies related to processing, transmission or I/O operations and it reduces the storage requirements of downstream entities.
- Data selection and filtering based on certain criteria or thresholds (Huang *et al.*, 2020): Similar to data summation, selection and filtering can significantly reduce the latencies and loads placed on downstream entities.

The processing approaches above are often split into two groups. The first group is data removal, data cleaning, and data homogenisation. The second group of approaches is data selection, data filtering, and data summation. The first group of approaches is typically done before medium to long-term storage and is used when dealing with relatively smaller amounts of data and real-time data (like within edge devices). The second group of approaches is usually done after storage and is used when dealing with relatively large amounts of data, such as data from large historical datasets or data from many child entities (like fog servers and data mining practices within big data pipelines).

## 9.3 Aggregation alternatives

The aggregation alternatives refer to various architectural decisions that influence, on the one hand, performance efficiency requirements discussed in Section 9.1 and, on the other hand, the data quality and management requirements discussed in Section 9.2.

### 9.3.1 Processing batch density

Processing batch density differentiates between stream processing, micro-batch processing and batch processing, where the batch density is influenced by:

- Datapoint size: The size of individual datapoints of a given data feature. For example, a single sensor reading is typically a small datapoint, whereas a single photo is typically a large datapoint.
- Number of datapoints per message: This refers to the number of data features in the message, as well as the number of datapoints per data feature.
- Message frequency: The rate at which messages are being transmitted.
- Number of messages per aggregation: This refers to the number of data sources that are being aggregated from.

Furthermore, these choices are mutually exclusive for a single intermediary communication, but consecutive intermediary communications may utilise different processing batch densities.

In stream processing, single data points from multiple sources are processed together. This type of processing happens frequently (relative to the frequency of micro-batch and batch processing). Therefore, this entails the frequent processing of small bits of data and is typically used when processing real-time data and when processing few data points from many sources. In some cases, stream processing is also used to break up a batch of data that is too large to process effectively. For example, when transmitting and processing a batch of 1000 photos, it may be

more effective to send and process the photos as individual messages as opposed to a single, large message.

Stream processing is typically used for applications that demand low latency. The small batch densities require less computational power, but the network infrastructure can be significantly strained. Stream processing is also not typically suited to data synchronisation or data inference applications. Furthermore, data cleaning or data summation would not typically coincide with stream processing unless there are pre-determined rules that can be applied to single datapoints.

In micro-batch processing, small collections of data points from multiple sources are processed together. This type of processing occurs less frequent than stream processing, but more frequent than batch processing. Micro-batch processing is typically used when processing relatively few data points from multiple sources that also require data synchronisation or in cases where network strain needs to be reduced (such as when sensors collect data at a very high frequency).

As with stream processing, micro-batch processing can be used to break up a batch of data that is too large to process effectively. Micro-batches are also used to break-up a batch of data for fault tolerance and diagnosability reasons. For example, when a fault can cause an entire batch process to fail, the batch can be divided into parts where only certain parts may incur faults. Therefore, only a part of the batch needs to be debugged and processed again, as opposed to the entire batch.

Batch processing entails the processing of a large collection of data, at a low frequency. Typically, batch processing combines fewer data sources, but many data points are collected per source, such as when aggregating large amounts of historical data from two or three datasets. The difference between micro-batch and batch processing is not clearly defined but is rather a difference of degree.

Batch processing can be very computationally intensive and thus is not preferred when low latencies are important or when computational power is limited. Furthermore, processing large batches of data can cause data inconsistencies when failures cause the processing to terminate early (this scenario is the premise for the Atomicity principle in database management systems that follow the ACID principles; refer to Section 10.4 for a discussion on database management systems). On the other hand, data inference also typically makes use of large batches of data and it is easier to discern difference in data veracity when large batches of data are compared.

### 9.3.2 Pre-storage vs post-storage aggregation

Aggregation before the data has been placed in medium-term to long-term storage is referred to *as pre-storage aggregation*, while aggregation after

medium-term to long-term storage is referred to as *post-storage aggregation*. For example, in SLADTA (Section 2.3), medium-term to long-term storage corresponds to Layer 5 of a DT and thus when data is aggregated before it is stored in a DTI's Layer 5 (through Layer 4) it is classified as pre-storage aggregation. Conversely, aggregation that aggregates data from Layer 5 through Layer 6 is classified as post-storage aggregation. It is possible to split the aggregation responsibilities across pre-storage and post-storage aggregation. For example, high frequency sensor data can be pre-storage aggregated, whereas low frequency sensor data can be post-storage aggregated.

Pre-storage aggregation is common when aggregating real-time data since it allows for reduced storage requirements by, for example, removing duplicate sensor values before storing the data. Pre-storage aggregation also has lower latencies than post-storage aggregation because data is exchanged before the database transactions need to be executed. Pre-storage aggregation is also more suited to multi-cloud environments because data can be aggregated before it is stored in numerous repositories in numerous cloud platforms.

Pre-storage aggregation is typically more applicable when considering network and resource constrained devices that are not able to maintain their own repositories or when aggregating data with a high sampling rate. However, these benefits may be negated if each of the aggregated entities still maintain the aggregated data in their own data repositories. Typically, pre-storage aggregation makes use of stream or micro-batch processing and is not suited to batch processing.

Furthermore, pre-storage aggregation is not typically applied to data that has high veracity or high worth. Pre-storage aggregation also typically entails data removal, data homogenisation and in some cases data cleaning and selection using pre-determined rules.

Post-storage aggregation is common when aggregating larger batches of data and when aggregating low frequency or historical data. Post-storage aggregation allows for more asynchronous and selective aggregation, but at the cost of higher latencies. Asynchronous aggregation here refers to the ability to wait for and aggregate a batch of data, such as aggregating historical data. Selective aggregation refers to the ability to select certain data features over certain timeframes without losing the original data.

Furthermore, post-storage aggregation also allows for more redundancy of the data when data has a high level of worth or high level of veracity, since the original data is not lost when aggregating. Typically, post-storage aggregation makes use of micro-batch or batch processing.

### 9.3.3 Local-network aggregation vs cloud-based aggregation

Aggregation of data within a common, local network is referred to as *local-network aggregation*, whereas the aggregation of data from different networks within the cloud is referred to as *cloud-based aggregation*. In cases where data is aggregated in the local network, but the data sources are from different networks, the characteristics of the aggregation may resemble either local-network or cloud-based aggregation. The distinction depends on 1) the number of external network connections, 2) the physical distance between aggregated and aggregating entity and 3) the capacity of the local infrastructure.

For clarification, local-network aggregation is considered to have far fewer external connections, much shorter distances between entities and sufficient computing capacity. In contrast, cloud-based aggregation has many external connections, there are large distances between entities and there is abundant computing capacity. Therefore, local-network and cloud-based aggregation are not all-encompassing, but they provide two distinct reference points. As with pre-storage and post-storage aggregation, the aggregation responsibility can also be split between local-network and cloud-based aggregation in a given DT and, even more so, in an implementation of a DT hierarchy.

Local-network aggregation is common in environments requiring low latencies and high throughput or it can be used to accommodate network and resource constrained devices. Aggregating within the local network also means that fewer external network connections are required which is also beneficial for privacy and security. Furthermore, local-network aggregation is typically more reliable (in terms of message loss) and typically has less latency variability. These qualities make local aggregation useful when dealing with high veracity and high worth data. However, local-network aggregation requires more and better local infrastructure compared to cloud-based aggregation. Finally, in multi-cloud environment, local-network aggregation is preferred because the data is aggregated before it enters multiple cloud platforms.

Cloud-based aggregation provides scalability in terms of computing resources and storage, making it easier to manage large processing loads and persistent data. Cloud-based aggregation also reduces the amount of on-premises computing and storage infrastructure required and thus the expertise required to design, install and maintain the infrastructure. However, cloud-based aggregation is very dependent on the network infrastructure and typically displays lower message transport reliability and higher network latencies and latency variability.

### 9.3.4 Aggregate entity

Section 8.3 discussed when it is appropriate to allocate service to a DT or to a service. This section further explains the differences when aggregating with a DTA

as opposed to a service in the services network. The distinction is intended to promote the separation of concerns.

DTs in the DT hierarchy typically share a messaging mechanism internal to the DT hierarchy, as opposed to making data available through a service or API. Therefore, DTAs can make use of this common communication mechanism for lower latency, pre-storage aggregation that is beneficial for near real-time applications, such as operational control scenarios. DTAs can also pre-process and pre-structure data that can then be stored for faster querying ability.

Services in a service network are typically cloud hosted and thus typically make use of data sharing mechanisms, such as APIs. As such, the services in the services network are not afforded access to the messaging mechanism internal to the aggregation hierarchy. Therefore, these services cannot make use of pre-storage or local aggregation. This restriction is intended to limit communication through the DT hierarchy messaging mechanism, to allow for simpler devices with lower resource requirements and to ensure more reliable and secure aggregation and communication within the DT hierarchy.

# 10 Implementation considerations

This chapter discusses the implementation considerations step (introduced in Section 7.7) of the design framework. This chapter discusses some of the most often encountered and most important implementation decisions when designing individual DTs, as well as an aggregation hierarchy. In particular regarding security, support services, messaging mechanisms, storage solutions and hosting options.

## 10.1 Security

Security is generally associated with the following principles (BSI *et al.*, 2011; O'Brien *et al.*, 2007):

- Confidentiality: Degree to which the system ensures that only authorised entities (either human users or other software programs) have access to data.
- Authenticity: Degree to which the identify of entities can be confirmed.
- Integrity: Degree to which unauthorised access and/or modifications to data or programs can be prevented.
- Accountability: The degree to which the actions of an entity can be traced back to that entity.
- Non-repudiation: The degree to which events can be proven to have taken place.

Furthermore, some developers also consider availability to be an aspect of security since some security attacks (such as denial of service attacks) seek to disrupt the normal functioning of the system, as opposed to stealing data.

Security is often divided into security in transit and security at rest. Security in transit refers to the security of data when it is sent between nodes in a network, whereas security at rest refers to the security of data when it is stored on a device.

Ensuring that systems have adequate security is a continual and changing process, but some security standards are provided here.

Federated identity (such as single sign-on) is the preferred method of authentication and authorisation with an API Gateway in a services network (Gadge & Kotwani, 2017). This approach allows for the decoupling of the authentication and authorisation functions. It also makes it easier to centralise these two functions, to avoid a situation where every service must manage a set of credentials for every user. There are three major federated identity protocols: OpenID, SAML and OAuth (Gadge & Kotwani, 2017). Figure 6 provides a diagram of the OAuth2.0 protocol flow where a client must request access from a resource

owner and then be authorised by a server before the requested resource can be acquired from the resource server (IETF, 2012). OAuth2.0 is industry-standard protocol for authorisation at the time of writing.

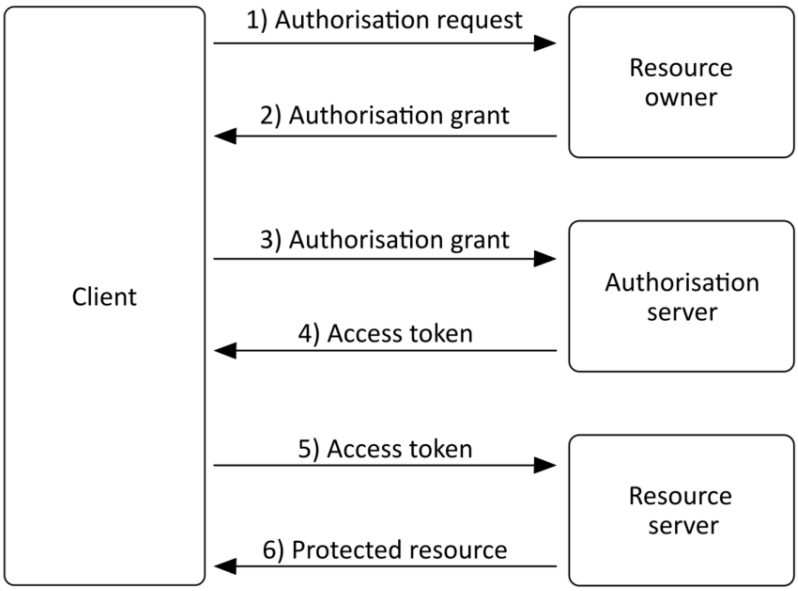



**Figure 6: OAuth2.0 abstract protocol flow. (Adapted from IETF, 2012)**

To prevent message tampering and thereby improve integrity and confidentiality, encryption is standard practice (Gadge & Kotwani, 2017). The SSL/TLS protocol is a standard for authentication and encryption during transit and at the time of writing TLS 1.3 is the latest version. TLS works by using asymmetric cryptography (also known as public key cryptography) to authenticate one or both connected parties. Once the parties have been authenticated, they exchange a symmetric encryption key that is valid for that session (Dierks & Rescorla, 2008).

Furthermore, additional security measures can also be undertaken to further improve the security of the system. For example, hosting services in a subnet that can only be made accessible through a proxy, such as an API gateway (Gadge & Kotwani, 2017) or making use of multi-factor authentication.

## 10.2 Management services

This section lists some management service that are often used in distributed computing environments. The management services mentioned in this section were derived from Ciavotta *et al.* (2017, 2020), Gadge & Kotwani (2017), Kuhn *et al.* (2020), Taibi *et al.* (2018). The management services discussed in this section were introduced as part of the overall architecture in Chapter 6.

### 10.2.1 Central user interface

The purpose of the *central user interface* (CUI) is to provide a single entry-point into the system of DTs. The CUI can serve as a flexible dashboard to display data for other services. However, it is expected that some DTs and their associated services may have their own dashboards and thus the CUI can help discover and redirect to other dashboards too.

Furthermore, when considering a single DT or a small group of DTs and/or services, it is feasible to have a separate interface for each DT or service. In complex systems, however, it may be more reasonable to have a single entry-point into the system which can serve as a single flexible dashboard or as a directory for users to find the appropriate DT or service for their needs.

### 10.2.2 Security service

The purpose of the security service (SS) is to fulfil the role of the authentication server when protocols such as OAuth2.0 are used. By having a central security service, other services can delegate authentication and authorisation functionality. This also promotes the separation of concerns since each service does not have to implement its own security. Furthermore, updates in security information, such as changes to the user roles, do not have to propagate through the system. However, having a central security service is a trade-offs that benefits performance efficiency and usability more than security (Gadge & Kotwani, 2017). It is often considered an acceptable trade-off since there are many other methods of further strengthening security and it can be very cumbersome to implement security in every individual service.

### 10.2.3 Gateway service

The purpose of the gateway (G) is to route service requests to the appropriate services or DTs and it is closely linked with the CUI. The CUI allows users to make requests, whereas the gateway transforms and directs those requests as necessary. The gateway can also be used by services to establish connections with each other. Gateways are further often used to transform data formats or communication protocols between internal and third-party provided services.

The gateway is intended to simplify communications (since services do not have to implement service discovery logic) and this reduces the number of requests made by a client. The gateway can allow for better service interoperability since the services only have to interface with the gateway as opposed to interfacing with each other. However, the API gateway can become a bottleneck and it can become complex when load balancing and multiple interfaces for different services are

considered. Server-side discovery is preferred with gateways because discovery logic is removed from services and it makes maintenance easier.

### 10.2.4 Directory service

The purpose of the directory service (DS) is to serve as a central metadata repository that can be queried for discovery information about DTs and services within the system. This typically includes metadata about the DTs such as a short description of the DTs' contents (such as what physical subsystem is being reflected, what models are available, what services are offered, etc.) and how to contact the DT (such as an IP address or related messaging topic). Similarly, for the service network, the directory provides metadata about the services' functionality and connection details.

To implement the directory, the DTs and services must register themselves when they start up or a log must be manually maintained that provides all the relevant metadata.

### 10.2.5 DT monitoring service

The DT monitoring (DTM) service is responsible for monitoring individual DTs (and other software components within the system) to ensure maximum availability of the DTs. Typically monitoring is split into two categories, i.e. health monitoring and load monitoring.

Health monitoring checks the connection status of DTs and resource availability within DTs to determine whether they are functioning as expected. Recommended measurements for health monitoring are (Gadge & Kotwani, 2017): 1) CPU, memory and thread usage, 2) network connectivity, 3) security alerts and 4) maintenance of logs.

Load monitoring of DTs is intended to provide data on DT performance and it allows for load-balancing between replicated DT instances. Recommended measurements for load monitoring are (Gadge & Kotwani, 2017): 1) number of service requests, 2) performance statistics and 3) success and exception messages

Furthermore, the DT monitor should keep interactions with the DTs to a minimum and should rather interact with the platform or host that the DT is running on, when possible.

### 10.2.6 Configuration Server

The configuration server (CS) is a central server that contains the start-up and operation configuration settings for the different digital components (DTs, services, brokers, etc.) of the system. Non-volatile operational configuration settings, such as user-specific settings, preferences, etc. can also be captured

within the configuration server. Having a centralised configurations server allows for the automatic deployment and redeployment of digital components and it makes the reconfiguration of distributed systems much easier.

### 10.2.7 Orchestration service

The purpose of an orchestration service (OS) is to logically sequence other services to achieve the desired data transformation and application functionality. This is likely only applicable to the services network where multiple functions need to be performed consecutively to achieve an outcome. Services hosted within DTs are expected to encapsulate all the functionality required to achieve the purpose of the service. There may be cases where the DT delegates functionality to the services network, but in such cases the DT is regarded as a client and not part of the services network.

## 10.3 Messaging mechanisms

Messaging mechanisms form a critical part of any distributed system and they can be complex in their own right. This section provides guidelines when considering alternative messaging and communication mechanisms.

### 10.3.1 Communications middleware

The purpose of a message-oriented middleware is to help manage heterogeneity and high message loads from many sources. A middleware can be used to facilitate communication and aggregation within or between DTs and middleware should be agnostic to the contents of the message and thus no processing should be performed on the contents of the message.

Furthermore, the middleware decouples communication in time (allows asynchronous communication) and space (allows software to run in separate processes), in addition to decoupling communication from a specific protocol. A middleware can also implement certain communication patterns, such as a circuit-breaker pattern, to improve the reliability of the connected services (Santana, Andrade, Delicato, *et al.*, 2021).

Message-oriented middleware is typically preferred in large systems with many concurrent users and requests and for asynchronous communication that involves large data loads. Some message-oriented middleware technologies also have support for multiple communication protocols and they are commonly required to have low latencies and high reliability (Karabey Aksakalli *et al.*, 2021; Tovarnitchi, 2017).

### 10.3.2 Messaging patterns

Messages can be exchanged according to different patterns and these patterns have different benefits and drawbacks. The messaging patterns can be broadly categorised into synchronous and asynchronous. Synchronous typically refers to request-response (such as APIs that make use of the RESTful approach) and it is commonly used for inter-platform communication (Bertoli *et al.*, 2021). Asynchronous communication typically refers to publish-subscribe communication or asynchronous request-response. Publish-subscribe makes use of a broker or other message-oriented middleware, while asynchronous request-response typically makes use of message queue.

Publish-subscribe also allows an entity to simultaneously be a data provider and a data receiver and this is often used for inter-service communication within the same platform (Bertoli *et al.*, 2021; Tovarnitchi, 2019). Publish-subscribe is typically preferred for one-to-many or many-to-many messaging and it is also very beneficial with regards to fast changing systems since the broker decouples publishers and subscribers, allowing for easy reconfigurations (Karabey Aksakalli *et al.*, 2021).

Table 17 provides a conceptual comparison between request-response and publish-subscribe messaging based on certain characteristics. The event-bus is conceptually very similar to publish-subscribe and is thus not considered on its own in Table 17.

**Table 17: Conceptual comparison of request-response and publish-subscribe messaging.**

| **Characteristic** | **Request-response** | **Publish-subscribe** |
|---|---|---|
| **Pattern** | One-to-one | One-to-many |
| **Participating entities** | Client, server | Publisher, broker (usually), subscriber |
| **Data flow** | Bi-directional | Uni-directional (per topic) |
| **Coupling** | Tighter<br>Client and server must be acquainted | Looser<br>Subscriber can be anonymous |
| **Synchronous/ Asynchronous** | Synchronous or asynchronous | Asynchronous |
| **Typical usage** | Inter-platform communication | Intra-platform communication |

### 10.3.3 Messaging performance parameters and solutions

This section provides an overview of how a messaging solution can be chosen and provides some of the most popular messaging protocols and technologies used industry and in research.

Choosing the most appropriate protocol depends on several characteristics of the use case, but the most important are: environmental conditions, network characteristics, the amount of data transferred, security levels and quality of service (QoS) requirement (where QoS in the context of messaging refers to the level of guarantee that a message is delivered) (Ferrández-Pastor, García-Chamizo, Nieto-Hidalgo, *et al.*, 2018). Based on the use case characteristics, certain performance parameters will be prioritised when considering a messaging solution. Table 18 provides some common performance parameters used to compare messaging solutions.

**Table 18: Descriptions for performance parameters with regards to messaging protocols**

| Parameters | Description |
|---|---|
| **Latency** | The amount of time it takes for a message to be captured, transmitted, processed and received when sent from a source to a destination. |
| **Transport reliability** | The ability of a protocol to transport data with minimal to no data loss under given conditions and for a given time interval. This includes a protocol's ability to compensate for faulty networks with differing QoS levels, message queues and similar functions. |
| **Security** | Security refers to authentication, integrity and encryption, as described in Section 10.1. |
| **Interoperability** | The ability of a protocol to allow communication and data transfer between functional units in a manner that requires minimal to no knowledge of the unique characteristics of the involved units. |
| **Resource usage** | The use of the elements of a processing system that are required to perform an operation. This includes the required bandwidth for communication, as well as the CPU and memory usage of the protocol. |
| **Throughput** | A measure of the amount of data that a message protocol is transferring within a given time interval. |
| **Scalability** | A protocol's ability and capacity to adapt to changes in network size and scale. |
| **Usability (Support/ existing technologies)** | The provisioning of services and materials that allow for the use and improvement of a protocol. This includes support for multiple programming languages and documentation to help with implementation. |

| Parameters | Description |
|---|---|
| **Complex communication suitability** | The ability of a protocol to allow for different communication architectures and data exchanges, such as publish-subscribe, request-response and contract exchanges. |
| **Message order** | A protocol's ability to ensure correct message order on the recipient's side. |
| **Message priority** | For asynchronous communication, the ability of a protocol to prioritise certain messages in the message queue over others, depending on a priority score. |

HTTP based protocols such as RESTful APIs have good interoperability and are thus good for cross-platform integration (Bertoli *et al.*, 2021; Longo *et al.*, 2019). HTTP is a synchronous, request-response protocol but, for asynchronous request-response, HTTP based protocols are often combined with message queues (Karabey Aksakalli *et al.*, 2021).

RabbitMQ is cited as a good option for a message-oriented middleware (Karabey Aksakalli *et al.*, 2021; O'Donovan *et al.*, 2015) because of its scalability and reliability and it supports multiple protocols and messaging patterns. Apache Kafka is often cited as a good technology for unidirectional data streaming because of its scalability and low latencies (Ciavotta *et al.*, 2020; Ismail *et al.*, 2019). MQTT and CoAP are popular lightweight and open protocols typically used in IoT where power and resource efficiency are important (Tovarnitchi, 2017). MQTT follows a publish-subscribe pattern, whereas CoAP follows a HTTP compatible request-response pattern.

Binary protocols such as gRPC and Apache Thrift are good for large, heterogeneous service environments because they require the user to publish interface definitions using a Interface Definition Language (IDL) (Protobuf and Thrift for gRPC and Apache Thrift, respectively) (Karabey Aksakalli *et al.*, 2021). Binary protocols also allow for polyglot programming.

## 10.4 Storage

Storage solutions are required in every DT, as well as in some of the services and thus this section provides some considerations when choosing data storage solutions.

### 10.4.1 SQL vs NoSQL

#### 10.4.1.1 SQL (relational database)

Relational databases are databases that store data in two-dimensional tables with rows and columns according to a schema-on-write model. The schema predefines

the type of data that can be inserted into a column with strict rules to ensure that the data is consistent. Relational databases use structured query language (SQL) to perform queries.

SQL is preferred for highly structured data that requires consistency (relational databases are good at ensuring consistent data across multiple instances of a database for multiple applications), complex queries (multiple operations can be performed using a single transaction) and no duplication (because of relational propagation of data as opposed to writing to multiple tables) and is often used for cases like banking information and order logs. SQL is also very reliable, durable and it follows the ACID (atomic, consistent, Isolated, durable) model.

Relational databases also allow for stored procedures, which are functions internal to the database, that ensure consistent functions and processes for multiple instances of the same database. Stored procedures also allow connecting tables to one another (create relations) to further improve consistency. Examples of relational databases are MySQL and PostgreSQL.

#### 10.4.1.2 NoSQL (non-relational database)

NoSQL databases typically follow the BASE paradigm (Basically Available, Soft state, Eventually consistent) meaning that they typically lose consistency to improve availability and performance (Bonnet, Laurent, Sala, *et al.*, 2011). This tends to make NoSQL more scalable (Ismail *et al.*, 2019) and better for cross-node operations and thus they are good for large volume data processing in distributed environments (Bonnet *et al.*, 2011).

Furthermore, NoSQL is schema-less which makes the database more adaptable to evolutional change and better suited to handling heterogeneous data (Ismail *et al.*, 2019). However, this does come at the cost of data consistency as mentioned above and the lack of a schema can cause data to become excessively unstructured which negatively impacts performance and usability. Some NoSQL systems do employ Multi-Version Concurrency Control (MVCC) to mitigate this issue by proving “weak consistency” (Bonnet *et al.*, 2011). Non-relational or NoSQL databases can be further divided into document stores and wide column stores.

Document stores (e.g. MongoDB) are intricate key-value stores where data is saved in parts known as *documents*. These documents typically contain relatively small amounts of semi-structured data in the form of JSON or XML. Documents typically represent a single data entry and thus documents are grouped into *collections* to represent multiple related data entries, such as multiple data entries from a single source. Documents do not have any set schema and thus entries do not necessarily have the same features etc. This is good for reconfigurability and adaptability, but data can become excessively unstructured if not managed well.

Document stores are also well suited to read intensive workloads (Bonnet *et al.*, 2011).

Wide column stores (also known as column-family databases, such as Cassandra or Bigtable) are data stores that still use a table-row-column format but different rows can have different column formats, i.e. there is some structure, but there is no strict schema. Wide column stores typically have fast write speeds, allowing for high throughput and low latencies but they are not as consistent or predictable as relational databases (Bonnet *et al.*, 2011). Furthermore, some wide column stores, such as Apache Cassandra, have good decentralisation support which is well-suited to distributed environments (Ciavotta *et al.*, 2020).

#### 10.4.1.3 Specialised datastores

There are some other storage options, often grouped with NoSQL databases, that have more specialized functions. Some of these more specialised datastores are listed below along with their typical usage.

Key-value data stores (e.g. Redis or Memcached) are a temporary data storage solution often used for caching. Key-value stores use system memory and thus have exceptionally low latencies but the storage is volatile, i.e. the storage is short-term and temporary. They also only support simple queries.

Object or file storage (e.g. Google Cloud Storage or Azure Blob Storage) solutions are typically used for file sharing and it uses a hierarchical, tree-like, file storage format. They are often used when dealing with large amounts of unstructured data such as images, videos, audio files and large csv files.

Graph databases (e.g. Neo4J) are databases that consist of nodes and edges, where nodes represent entities and edges represent relationships between entities. This is commonly used when the relationships between nodes is an important factor in the data. For example, graph databases are popular for fraud detection because relationships between people and places can be identified more easily.

Specialised time-series datastores (e.g. InfluxDB) are available that are specifically designed to collects large amounts of real-time data from multiple sources. Typically, time-series datastore are optimised for many small data entries that are rarely updated or deleted.

Full-text search engines (e.g. Elasticsearch) are another type of data store that is specifically used for text-based searches, but they are not durable and should not be used as a primary data store.

### 10.4.2 Operational and transactional vs analytical datastores

Operational datastores (technical operations data) and transactional databases (business data) are datastores that record the day-to-day data generated within a company. This often entails many simple read-write operations that require low latencies so that multiple concurrent applications have access to real-time data. This type of datastore is primarily used for dynamic real-time data used to answer short-term questions (Therrien *et al.*, 2020).

In contrast, analytical datastores are data storage systems that specialize in storing data for data analytics. There are various types of analytical datastores but the most common is a data warehouse. Other types analytical datastores include data lakes and data marts. Analytical datastores are typically concerned with high volumes of historical data that rarely changed after entry and, thus, reading and querying ability is often emphasised over writing speeds with analytical datastores. Analytical datastores are often classified according to the order of extraction, transformation and loading. *Extract* refers to the gathering of data from multiple sources, *transform* refers to the transformation of data into a more useful structure and format and *load* refers to writing the data to the store for further use.

A data warehouse is subject-oriented, integrated (data is transformed and stored in a standard format) datastore, that typically organises data by time period. In data warehouses, data is not updated in real time but rather updated periodically because they typically contain data for data analytics and long-term strategic decision-making (Suba, 2018). Data warehouses conform to an extract, transform, load (ETL) method of data entry, meaning a schema is enforced when writing data. Therefore, data warehouses typically makes use of a relational database model to store data in a structured format (Therrien *et al.*, 2020).

An alternative analytics datastore to a data warehouse is a data lake, which conforms to an extract, load, transform (ELT) model of data entry. Data lakes do not enforce a schema when writing data. This makes them easy-to-use with unstructured data but this also holds the risk of the data becoming so unstructured and lacking in metadata that data becomes overlooked, misused, corrupted and ultimately unusable (Therrien *et al.*, 2020).

Furthermore, data warehouses often make use of data marts, where a data mart contains a subset of data that is applicable to a particular group of people within the company. The top-down approach of designing a data warehouse entails the analysis of business requirements and then designing and implementing the data warehouse according to those requirements. Dependent data marts (dependent because data comes from data warehouse) are then linked to the data warehouse to provide specific data. This generally ensures better consistency and

standardisation but the design stage may take longer and cost more to deploy (Suba, 2018).

The bottom-up approach of designing a data warehouse implements independent data marts (data marts do not get their data from the data warehouse) as quickly as possible. The data warehouse is then built by integrating *conformed dimensions* from the data marts (Suba, 2018). This generally provides data storage solutions quickly but special care must be taken to enforce standards for seamless integration (Suba, 2018).

## 10.5 Hosting options

Hosting options refer to the hardware or platform that the software is hosted on as well as virtualisation techniques that can help to better exploit the potential of the hardware. Practical solutions would most likely consider a combination of hosting positions as well as making use of virtualisation.

### 10.5.1 Hosting positions

#### 10.5.1.1 Overview

Hosting positions are here divided into three categories: local device, fog server and cloud. Local devices (also known as edge devices) refer to any device that is directly connected to the sensors and actuators that are being monitored. For example, a Raspberry Pi or a PLC can be local devices because they receive, interpret and respond to signals from sensors.

Fog server refers to dedicated servers at the network edge that are typically used to provide additional computational and storage capacity within the local network. Fog servers are typically used when the local devices do not have sufficient computing capacity or power to perform all the tasks required of the digital system within the local network.

The cloud refers to any computing infrastructure that is managed and maintained by a recognised third-party provider, where the computing infrastructure is in an off-premises, specialised facility. Examples of cloud providers are Google Cloud Platform (GCP), Amazon Web Services (AWS) and Microsoft Azure. The cloud is often associated with exceptionally accessible, scalable and “serverless” computing resources (serverless meaning the physical servers are managed and maintained by the cloud provider).

#### 10.5.1.2 Comparison of hosting positions

Local devices are typically used for low latency, real-time decision making within the local data context (Villalonga *et al.*, 2021), such as health monitoring of a

critical asset. The local device can also be used for time synchronisation and data stream buffering to, for example, manage a few high-frequency sensors (Karanjkar *et al.*, 2018). Local devices can also be used to compensate for other resource constrained devices by providing network connectivity and relatively simple processing (in comparison to the processing done using fog servers or the cloud) (Kuhn *et al.*, 2020). This is typically the case in wireless sensor networks, where resource constrained sensors transmit their data to the local device using short-range communication protocols, such as Bluetooth Low Energy.

Fog computing was designed to overcome the shortcomings of cloud technology with respect to real-time applications and physically distributed systems (Bertoli *et al.*, 2021), while also overcoming the problem of limited resources found in local devices. Therefore, fog servers are suited to applications such as near real-time control of multiple devices and low latency local analytics, which require more computing resources than a local device typically has available (Ullah *et al.*, 2021). Fog servers are also often used to reduce network congestion (Ferrández-Pastor *et al.*, 2018) and to ensure reliable data transfer within the local network as well as to external networks (Ciavotta *et al.*, 2020).

Furthermore, fog servers are also intended to improve security and interoperability. Security is improved because fog servers can host sophisticated security software, such as firewalls, antivirus and anti-malware software, and can then act as proxies for other local devices to external networks (Cisco, 2015; VanDerHorn & Mahadevan, 2021). Security policies can also be met using fog servers since they provide location awareness that allows for specific security measures in accordance with local government (Bonomi, Milito, Zhu, *et al.*, 2012). Fog servers are also used to improve interoperability within the local network by hosting sophisticated middleware (Ullah *et al.*, 2021).

The cloud offers accessibility and scalability in data storage and processing power (Harper *et al.*, 2019; Therrien *et al.*, 2020; VanDerHorn & Mahadevan, 2021) and it has exceptionally high availability (Tovarnitchi, 2017). The cloud also consists of more than just hardware and typically there is also a selection of service that help manage the cloud platform as well as help utilise its hardware more quickly and efficiently. Using cloud infrastructure also saves the cost of server acquisition, maintenance and eventual upgrade, but fast and reliable internet access is required (Therrien *et al.*, 2020).

The cloud can be divided into public cloud and private cloud. The private cloud is not accessible through the internet but is only accessible from the local network (and using VPNs). This offers better reliability, control, performance, privacy and security but at the cost of lower accessibility and interoperability (Givehchi, Imtiaz, Trsek, *et al.*, 2014). Private cloud is typically used in industrial environments where

services do not focus on the user, but rather on managing a device on behalf of a user.

In practice, a hybrid between public and private cloud is often used. Hybrid cloud combines the easier access of public cloud with the reliability and performance of private cloud. Generally, applications with inconsistent rising and falling demand for network resources are best served by the public cloud, whereas applications that require consistent, high levels of network resources are better served by the private cloud. (Odun-Ayo, Ananya, Agono, *et al.*, 2018)

### 10.5.2 Virtualisation

Virtualisation refers to the creation of virtual hardware to allow for more versatile and efficient use of the actual hardware. Two forms of virtualisation are considered here: virtual machines (VMs) and containers.

VMs partition a part of the hardware resources to host a separate operating environment. Therefore, VMs provide resource and operating environment isolation between software components and VMs in the cloud are also able to take advantage of some cloud support for load balancing and scaling to improve availability (Karabey Aksakalli *et al.*, 2021). VMs are good for implementing security measures around a service. However, VMs are relatively slow to deploy in comparison to containers and, furthermore, VMs employ static resource partitioning.

Containers also partition a part of the hardware resources to host a separate operating environment, but the method used to do this allows for dynamic resource allocation and faster deployment than VMs (Karabey Aksakalli *et al.*, 2021). Containers are typically deployed using a cluster manager (such as Kubernetes or Docker Swarm) to help monitor and manage them (Akbulut & Perros, 2019).

Both VMs and containers are good for reliability since they allow for environment reproducibility (where dependencies are packaged with the software) and failure isolation (Santana *et al.*, 2021). VMs enforce a more severe partitioning and thus greater isolation and security, but at the cost of resource elasticity and deployment speed.

# 11 Design patterns

The preceding sections give a large number of design considerations that often have to be balanced in compromises. The number of choices can be daunting. However, in practice, many applications are characterised by having one high-priority (even dominant) quality attribute. This section provides six design patterns, where each design pattern is focussed on a different quality attribute. It should be noted that the quality attributes are interdependent and thus changes to one quality attribute will likely also influence others.

The quality attributes that the design patterns focus on are performance efficiency, reliability, maintainability, compatibility, portability and security. Usability and functional suitability do not have design patterns because they are too dependent on the use case.

In the context of the research presented here, the design patterns serve as abstract case studies that demonstrate the application of the information presented in Chapters 9 and 10.

In practical applications, the benefits of a pattern-based software architecture include ease of maintenance and reuse (Aderaldo *et al.*, 2017) and they provide common solutions to common problems that can be applied to a given context (Bourque & Fairley, 2014). Various DTs in the aggregation hierarchy can make use of different design patterns. Furthermore, it is likely that more than one design pattern would be applied to a given DT, to iteratively improve a particular quality attribute.

For the sake of brevity, the core aspects of each design pattern are not worded in full sentences.

## 11.1 Performance efficiency

Related priorities:

Responsiveness, scalability, timeliness, capacity, resource utilisation.

Related needs:

The related complexity needs are: N18, N24, N25, N27. The performance efficiency design pattern is also applicable when the latency, throughput and resource usage of a particular service needs to be improved.

Performance metrics:

Latency, throughput, infrastructure measures. See Table 16 in Section 9.1.

Conflicts:

Data detail and DT fidelity, security, portability and interoperability (see Section 4.5).

Recommended aggregation choices:

- Design for the separation of concerns and the distribution of load across multiple DTs to improve the scalability of the architecture. DTs can also be replicated for horizontal scalability, partitioned for z-axis scalability or provided with more resources for vertical scalability. This scalability is enabled by the hierarchical aggregation architecture as discussed in Section 5.3. Furthermore, the separation of concerns reduces the amount of data that needs to be exchanged between subsystems.
- The span of reality should be narrowly defined and should serve a specific purpose to improve query response times. This includes the following:
    - Data granularity should be chosen to include only the necessary data for the intended purpose.
    - Typically, a DTI or DTA is used, depending on the physical scope being considered, because they are more dedicated to an asset and they encapsulate functionality as opposed to orchestrating functionality as done in services networks.
    - As the physical scope gets larger, the data granularity should get coarser and the processing operations should be more extensively applied.
- Pre-storage aggregation should be used where possible to reduce latencies and/or to reduce storage requirements. This includes:
    - Pre-processing and pre-structuring data before storage for faster response times to queries. This may include performing data homogenisation and data synchronisation before storage.
    - Exchanging data through a messaging mechanism internal to the aggregation hierarchy for more rapid data to propagation to the higher levels of aggregation.
    - Reducing the data (e.g. by removing duplicate values) before storing the data.
- Local aggregation should be used where possible to increase throughput and to reduce latencies. This includes:

  - Making use of local infrastructure (i.e. local devices and fog servers) for reduced latencies and increased throughput.
  - Reducing the physical distance between DTIs and aggregates for reduced latencies
  - Avoiding limitations imposed by cloud platforms such as throttling.
- Aggregate through stream processing or micro-batch processing. Furthermore, it may be beneficial to implement the aggregation processing operations' logic in the DTIs to prevent the DTA becoming a bottleneck.

Recommended implementation choices:

- Minimise the number of intermediaries between data source and data destination (for automatic decision making) or requester and responder (for human decision making).
- Minimise the physical distance between source and destination to minimise network latencies.
- Minimise the required amount of data formatting and protocol conversions that need to be made, by using a commonly agreed upon format and protocol.
- Request-response messaging patterns tend to have lower point-to-point latencies, but publish-subscribe is more scalable. Some message-oriented middleware may allow for both.
- NoSQL databases can support higher throughput and lower latencies than SQL databases and NoSQL databases are more scalable. Using a specialised storage solution such as a time-series data store or a key-value store for temporary storage may also be useful.
- Make use of local infrastructure or private cloud offerings where possible. Furthermore, hosting DTs and services in containers will allow for elasticity which is good for resource efficiency and it makes load balancing easier.

Further recommendations:

- Multi-threading can reduce I/O latency and multi-processing can reduce computational latency. However, this is also dependent on the resources that are available in the DTIs and DTAs.
- A component can save resources by delegating some functions to other components, particularly processing intensive workloads. For example, a local device can save resources by using a secure fog server as a proxy where the fog server performs security services.
- Caching frequently queried data as opposed to accessing the database for every query can greatly reduce latencies but at the cost of higher memory usage.

## 11.2 **Reliability**

Related priorities:

Maturity, availability, fault tolerance, recoverability, resilience, robustness.

Related needs:

The related complexity needs are: N15, N18. The reliability design pattern is also applicable when network availability is uncertain, when messages have high veracity or worth, when power outages may cause disruptions, when applications are safety critical, etc.

Reliability metrics:

The length of downtime after system or component failure. The number of faults, failures or error messages within a given timeframe. The percentage of time that a system or component remains available within a given timeframe. The number of downstream failures (the number of failures caused by an initial failure).

Conflicts:

Agility (See Section 4.5).

Recommended aggregation choices:

- Design for separation of concerns (distribution of functional logic) and the distribution of load to allows for fault isolation. For example, service such as fault monitoring can be implemented on locally hosted, distributed DTIs (as opposed to a central DTA) so that even if the external network connection fails or if there are failures in an aggregate, the DTIs remain operational. Furthermore, replication and partitioning also contribute to reliability by improving scalability which is complementary to reliability since it allows for a better response to load disruptions.
- The span of reality should be defined to leave reserve capacity for interaction, i.e. the full infrastructure capacity of the system should rarely be utilised. This essentially serves as a safety factor so that when there is some disruption to the system, it has additional capacity to compensate. Therefore, often a reliable upper bound of resource utilisation is identified that allows for stable hardware functioning and allows for some disruption.
- Typically, DTIs and DTAs are used to host services with high reliability requirements because querying data from a single source, such as a data warehouse within a DT, is more predictable than data federation. The encapsulation of functionality within a persistent and dedicated DT is also more reliable than service orchestration.

- Post-storage aggregation is generally better for reliability because data is stored before it is transferred or manipulated, which allows for some redundancy. If latency is also important, a key-value store with periodic persistence, such as Redis, can be used as low latency temporary storage before aggregation.
- Local aggregation is generally more reliable because the low latencies and high throughput of a local network allows for a higher capacity for interaction.
- Stream processing and micro-batch processing should be used for aggregation to prevent large batch failures. Furthermore, it may be beneficial to implement the aggregation processing operations' logic in the DTIs to prevent the DTA becoming a bottleneck. However, this is also dependent on the resources that are available in the DTIs and DTAs.

Recommended implementation choices:

- A DT monitoring service's primary focus is to ensure maximum DT uptime. Therefore, a DT monitoring service should be used and the extent of the monitoring is dependent of the level of reliability required. However, at the very least, a heartbeat monitor or watchdog service should be implemented. Furthermore, if the middleware makes use of load-balancing or the circuit breaker design pattern, the monitoring service will have to capture performance metrics to enable those functions.
- Message-oriented middleware is generally beneficial for reliability because middleware typically makes use of short-term and temporary storage, which ensures some redundancy. Middleware also decouples components in time, space and protocol, i.e. asynchronous communication and processing are enabled and interoperability between components is improved. Furthermore, middleware can employ load balancing and/or reliability focussed software design patterns such as the circuit breaker pattern.
- In resource constrained environments, publish-subscribe messaging is typically more reliable than request-response because the broker decouples publishers and subscribers in time and space.
- For persistent storage, SQL databases are typically more reliable and durable than NoSQL and they have better data consistency.
- Private cloud hosting is the most reliable hosting position for service with constant loads, whereas the public cloud is the most reliable option for highly dynamic loads and when the service must be publicly available. However, transmission within the local network is more reliable than transmission across networks. Therefore, if reliability is required to prevent message loss, the services should be hosted in the cloud with short-term local storage for data redundancy. However, if reliability is required because of intermittent

network availability, then it may be preferable to host services locally on a fog server.

- Virtual machines provide the best resource and operating environment isolation. However, they do take longer to deploy which means that when they do fail, they will have a longer downtime than containers.

Further recommendations:

- Digital twins should employ default safe state logic in case of external failures such as network or communication failures.
- When components need to be replaced within the system, introduce new components using a split load approach, where possible. For example, when a new version of a service is available, the load can be split between the old and the new version until the new version proves adequately reliable. Cloud platforms, such as Google Cloud Platform, provide services that enable this type of load splitting.
- Make use of physically and logically partitioned communication for different types of communication. For example, the aggregation communication can be separated from the digital twin's internal communication, where different hardware modules are used by different OS processes.

## 11.3 Maintainability

Related priorities:

Modularity, reusability, analysability, modifiability, testability, reconfigurability.

Related needs:

The related complexity needs are: N6, N9, N14, N16.

Maintainability metrics:

The time required to add, remove or rearrange system components without introducing failures. The reuse rate of software modules. The time it takes to identify the cause of a failure.

Conflicts:

Security (see Section 4.5). Performance efficiency and reliability (because testing and maintaining distributed software logic can be harder than maintaining centralised logic).

Recommended aggregation choices:

- The separation of concerns increases the modularity and reusability of software, which makes it easier to maintain the system. Modularity also improves the scalability and expansibility (as defined in Section 4.3) of the system and makes it more adaptable. However, replication and partitioning of DTs may cause data consistency and code versioning issues (e.g. where one DT gets updated but its replicant does not).
- The span of reality should serve a clear purpose and should be defined to produce self-contained and modular DTs with few, well-defined and simple interfaces. Therefore, the more complex the system, the more aggregates and levels of aggregation there are likely to be.
- Spatially focussed services would be more maintainable in a DT whereas a functionally focussed service may be better maintained in the services network.
- The choice of pre-storage vs post-storage aggregation is not likely to affect maintainability much. However, it may be beneficial if DTAs establish the aggregation communication. This means that the DTA contains all the aggregation metadata, such as which DTs are aggregated and which features are relevant. Furthermore, it would be easier to maintain the aggregation processing operations' logic if it is implemented in the DTA, but this may cause the DTA to become a bottleneck.
- Cloud-based aggregation could be better for maintainability because of the additional support services that are provided by most cloud platforms.
- The processing batch density is not likely to affect the maintainability of the DTs much, if at all.

Recommended implementation choices:

- Making use of a central user interface, security service, gateway service, directory and configuration server should be beneficial for maintainability, where more complex systems would benefit more. Each of these management services encapsulate a particular functionality that helps to separate concerns. This separation of concerns allows DTs and services to be focussed on a single responsibility by delegating functions to the management services.
- Message-oriented middleware would be test best option with regards to maintainability. Message-oriented middleware decouples entities in time and space, which reduces dependencies and thus simplifies interfaces. Furthermore, some message-oriented middleware, such as RabbitMQ, can also support multiple communication patterns and protocols, which also simplifies interfacing. In resource constrained environments, the publish-subscribe pattern is likely to be better than request-response because the broker still decouples entities in time and space.

- The best storage solution for maintainability depends on the data. Structured data may be more maintainable in an SQL database or wide column store, whereas semi-structured or unstructured data may be better maintained in a wide column store or document store. Furthermore, NoSQL datastores are generally better in distributed environments and some datastores such as, Apache Cassandra, employ multi-version concurrency control which eases the management of distribute data.

  Specialised datastores may also be preferred. For example, a graph database, such as Neo4j, provides a visual data interface that can make some data more comprehensible and thus manageable.
- The cloud offers “serverless” computing which means that the maintenance related to hardware is delegated to the cloud provider. Furthermore, cloud providers often offer additional support services to help deploy and manage software in the cloud.
- Containers are widely regarded to be beneficial for maintenance since they provide resource and operating environment isolation. Containers are usually deployed using a cluster manager, which makes aspects such as software performance monitoring and load balancing easier.

Further recommendations:

- Code versioning services, such as GitHub, are widely recognised as a means to manage code versioning more effectively, particularly when multiple developers are contributing to the code.
- Check that software is backward compatibility and designing new systems with backward compatibility in mind can greatly contribute to maintainability.
- Incrementally develop modular and testable software features with accompanying reusable unit tests.
- Implement good document management practices so there is a dependable reference for system components, functions, relationships, etc.
- Allow for over the air (OTA) programming for easier reprogramming of distributed devices.

## 11.4 Compatibility

Related priorities:

Interoperability, co-existence.

Related needs:

The related complexity needs are: N3, N5, N6, N7, N23, N27.

Compatibility metrics:

The time required to add a new component and/or to integrate with external services. The processing time required to transform heterogeneous data formats and protocols.

Conflicts:

Security, performance efficiency (see Section 4.5).

Recommended aggregation choices:

- The separation of concerns and modularity of the architecture simplifies the interfaces with the DTs and services, making them more interoperable. The degree of isolation provided by the separation of services across DTs and the services network also enables easier replacement and expansion of the system's functionality. However, replication and partitioning may cause resource contention and thus poor co-existence if it is not managed appropriately.
- The span of reality should serve a clear purpose and should be defined to produce self-contained and modular DTs with few, well-defined and simple interfaces. Therefore, the more complex the system, the more aggregates and levels of aggregation there are likely to be.
- The services network is intended to interface with external data source and services as discussed in Chapter 6.
- Pre-storage aggregation may be beneficial for interoperability because data can be homogenised and structured before storage, making later querying easier.
- Local aggregation is likely to be better for interoperability is cases where multiple cloud platforms are present (e.g. as when multiple DT owners use different cloud platforms) because data can be exchanged between entities before entering the different cloud platforms.
- The processing batch density is not likely to make a difference in terms of interoperability.

Recommended implementation choices:

- A gateway service and directory service can both be beneficial for interoperability. The gateway service simplifies interfacing between services because services can be designed to interface with the gateway as opposed to multiple other services. In some cases, gateways are also used to convert between communication protocols. The directory service can be queried by DTs to retrieve interfacing information about other DTs. For example, when

multiple DT owners form part of the system, the DT directory can be queried to determine what data is available in which DTs.

- Openly available message-oriented middleware is good for interoperability because it decouples the communicating entities. Furthermore, some message-oriented middleware can support multiple communication protocols and programming languages. Furthermore, binary protocols, such as gRPC, can also support multiple development languages for better interoperability.
- In some cases, having a strict data schema can be beneficial for interoperability and in such cases an SQL database may work best. However, when dealing with semi-structured and unstructured data, it would likely be better to use NoSQL databases or specialised databases since they are better suited to handling heterogeneous data.
- Hosting software entities in the cloud will ensure the best co-existence. Furthermore, hosting software entities in containers and VMs provide resource partitioning for better co-existence, where VMs implement stricter partitioning. Containers and VMs also help with versioning control which is important since not all versions of software are backward compatible. Furthermore, hosting databases and services in the public cloud makes them more accessible to clients (whether the clients are humans or other software entities).

Further recommendations:

- The data format should strive to be syntactically and semantically consistent across different DTs and metadata should be incorporated to provide context.
- Making use of open industry standards for data formats, communication protocols and security protocols can greatly improve interoperability.

## 11.5 Portability

Related priorities:

Adaptability, installability, replaceability.

Related needs:

The related complexity needs are: N6, N9, N14, N16.

Portability metrics:

The time it takes to install new components. The time it takes to replace components. The time it takes to transfer a software package from one platform to another or from one operating environment to another.

Conflicts:

Reliability (See Section 4.5).

Recommended aggregation choices:

- The separation of concerns and modularity help make components more replaceable and interchangeable. The ability to aggregate existing DTs into new DTs also allows for the adaptable and incremental development of increasing more complex DTAs and services.
- Span of reality should be defined to be modular and may be defined with a more general purpose than some of the other design patterns. More DTs and more levels of aggregation may also be preferred to increase the pool of DTs that can be aggregated from.
- Services in a service network are generally preferred for adaptability because functionality can be orchestrated as required.
- Post-storage aggregation may be preferred since services in the services network make use of post-storage aggregation. Post-storage aggregation also allows for more asynchronous and selective aggregation.
- Cloud-based aggregation may be preferred since cloud platforms are highly elastic and can thus adapt more easily to varying processing loads.
- The processing batch density is not likely to make a difference to portability. However, services in the services network are more likely to make use of batch processing.

Recommended implementation choices:

- A directory service allows for automatic lookup of relevant information that can be useful for adaptability. An orchestration service may be required for services in the services network.
- A message-oriented middleware and publish-subscribe messaging with a broker are beneficial for adaptability and replaceability since both patterns decouple the communicating entities.
- NoSQL databases are more adaptable than SQL databases since they do not enforce a strict schema. Furthermore, the JSON type data format of document stores makes them particularly adaptable to changing data features. Object datastores are also well suited to storing large, unstructured data, such as images and videos.

- The cloud is the most adaptable hosting position since it is highly elastic and provides a multitude of hardware configurations to choose from. This also makes it easy to replace and upgrade hardware as demand on the system increases.
- VMs and containers are both very beneficial for portability since they allow for environment reproducibility across multiple platforms and types of hardware. Containers are particularly popular for portability since they also allow for dynamic resource allocation as well as fast deployment and redeployment.

## 11.6 Security

Related priorities:

Confidentiality, integrity, authenticity, authorisation, accountability, non-repudiation.

Related needs:

The related complexity needs are: N2, N4. Security become increasingly important when working with confidential information and often governments enforce data privacy requirements as well.

Security metrics:

The security of the system is hard to quantify but some metrics that can be used are the number of vulnerabilities that have been identified and the percentage of those that have been patched. The impact of a vulnerability being exploited, typically rated on a low to high scale. The attack area - the number of open external network connections within the local network.

Conflicts:

Interoperability, maintainability, performance efficiency.

Recommended aggregation choices:

- Distributing services across multiple DTs and the service network decreases the impact of a vulnerability exploit since only a subdivision of the data and functionality would be exposed. DTs can be replicated and partitioned to separate data processing over multiple instances. If one instance is compromised, other instances can remain unaffected and continue activity. Furthermore, if one instance of a partitioned DT is compromised, only the data within the DT instance is compromised and not all the data.

- The capacity for interaction should make allowance for security related processing and storage. For example, encryption and decryption require some processing, while storage is required to log activities and to keep track of whitelisted connections, such as approved IP addresses.
- Services in DTs are likely to be more secure than services in the services network since DTs encapsulate the functionality and data required for a service and thus DTs require fewer external connections.
- Post storage aggregation may be beneficial to create data redundancy.
- Local network aggregation is more secure since the network can be closely monitored for suspicious activity and making use of firewalls, antivirus and anti-malware software can further secure the local network. The local network can be further isolated from the internet by using proxies.
- Stream processing and micro-batch processing are preferred to allow for data partitioning to reduce the severity of exposure.

Recommended implementation choices:

- Standard security protocols should be employed for all communication. This may include authentication, authorisation and encryption between all entities as opposed to using federated security. This may also include implementing individual security measures in each DT and/or service, such as password protected access and multi-factor authentication measures.
- A DT monitoring service can be used to monitor DT activity. This can help by detecting slow connections, by providing notifications about access by suspicious IP addresses and other security alerts, etc.
- Messaging protocols and messaging technologies should support the use of security protocols. For example, MQTT provides easy integration with the SSL/TLS protocol and AMQP is a protocol that is known for its security provisioning. Furthermore, message-oriented middleware and brokers can apply additional security restrictions such as regulated access to topics and resources.
- DTs should never allow direct access to a database but should rather provide a regulated service through which database queries can be made. Furthermore, any reputable database management system, whether it is an SQL or NoSQL database, should provide multiple features related to security. This typically includes provisioning for multiple users with differing levels of access. Encrypting data before storage in another means of securing data within a database.
- Fog servers are often used to provide security. Fog servers form part of the local network and can thus communicate within a local private network. Fog servers also have substantial computing resources to implement security

measures. The private cloud is also preferred for security over the public cloud because the private cloud is linked to the local network and not the internet.

- For virtualisation, VMs are preferred in terms of security because they provide better support for security and isolation than containers.

Further recommendations:

- DTs and services can be hosted in a private subnet that can only be accessed through a secure proxy with a list of whitelisted connections.
- Vulnerable legacy systems should be retrofitted with new technologies that implement security measures.
- Multi-factor authentication can be applied in individual DTs.

# 12 High-level case studies

This chapter presents two high-level case studies. These case studies focus on architectural aspects, with limited implementation detail. The detailed case study presented in the next chapter does include more implementation details.

The purpose of the case studies presented here is to demonstrate the systematic approach of the design framework in two different case studies. The case studies each present unique challenges and considerations and by applying the design framework to each of these cases the general applicability of the design framework is also demonstrated.

## 12.1 Water distribution system

### 12.1.1 Scenario

A water distribution system (WDS) is responsible for the transportation of water from storage sources to consumers with appropriate quality, quantity and pressure. To achieve this, the physical integrity, hydraulic integrity and water quality integrity of the WDS must be monitored and maintained (Van Zyl, 2014). Physical integrity refers to the ability to have correctly functioning components that also maintain a barrier between the water in the network and the external environment. Hydraulic integrity refers to the ability of the distribution system to meet all the users' demand (domestic, commercial, industrial, etc.) while ensuring desirable pressures, velocities and water age in the system. Water quality integrity refers to the distribution system's ability to deliver water of acceptable quality to its users.

The scope of the WDS considered in this case study includes the infrastructure systems that are intended to deliver water to consumers with appropriate quality, quantity and pressure. This includes water storage systems, pipe networks, pumping stations and water treatment facilities. For this case study, a Kentucky WDS, KY12, that forms part of a set of benchmark WDSs will be used (Jolly, Lothes, Sebastian Bryson, *et al.*, 2014). The KY12 WDS is the largest of the set of 12 WDS defined in Jolly *et al.* (2014) and some metadata about the data captured within the WDS is freely available on the University of Kentucky website (University of Kentucky, n.d.). Therefore, it serves as a good representative WDS for this case study.

Concerns raised by researchers in the water sector include:

- Increases in urban populations cause increases in demand, as well as water quality degradation (Butler, Farmani, Fu, *et al.*, 2014).

- Changing weather conditions and changing rainfall patterns are creating uncertainty with regards to water supply. In South Africa, this has placed tremendous pressure on water supply and infrastructure management (Archer, Landman, Malherbe, *et al.*, 2019).
- Infrastructure is aging and, in many cases, has not been maintained properly. Therefore, some infrastructure needs to be replaced and demand for water must be managed during the development of the new infrastructure (*SAICE*, 2017).
- There is a lack of water engineering and artisanal expertise within South Africa and there is a lack of resources to support the expertise that is available (Brown, Keath & Wong, 2009; Sharma & Vairavamoorthy, 2009). This means that there is insufficient expertise and capacity to undertake the infrastructure development mentioned in the previous point.

Researchers in the water sector have turned to digitisation to alleviate some of the concerns mentioned above. However, current monitoring and control technology can vary significantly in terms of communication methods and capabilities. For example, some subsystems make use of SCADA systems, while others make use of varying IoT communication methods such as 3G-based protocols, Sigfox or LoRaWAN. Furthermore, much of the sensor and control infrastructure is installed and maintained by third-party providers that provide access to the data through webhooks or APIs.

In this context, digital twins are being proposed as a potential method to integrate and manage the data to facilitate services that can help alleviate some of the challenges. Owen (2018) provides a list of potential services that may be applicable to WDSs, including:

- Active leak control prioritisation: Prioritisation of leak repairs, as well as pressure management in response to leaks, to minimise water loss.
- Benchmarking: Comparing operational zones within a WDS and comparing various WDSs.
- Anomaly detection: Detecting abnormal operation and irregular performance within the distribution network.
- Burst awareness: Generating real-time alerts in response to bursts along with locational pinpointing for improved remediation.
- Works optimisation: Optimised scheduling of maintenance to balance effort with impact.

### 12.1.2 Needs and constraints analysis

This section presents the needs and constraints analysis for the WDS case study. The high-level FRs are listed in Table 19 along with a rationale for the FR, a grouping and a prioritisation (as discussed in Section 7.3).

**Table 19: Functional requirements for the water distribution system**

| High-level functional requirements | Rationale | Group (Primary or secondary) | Priority |
|---|---|---|---|
| Demand estimation | The growing population is causing increased demand for water supply and this demand must be accurately estimated so that adequate water supply can be ensured. | Primary | Mandatory |
| Supply estimation | There is considerable uncertainty with regards to the supply, including uncertainty about rainfall and evaporation losses, uncertainty about the infrastructure's ability, the inability to perform effective maintenance, etc. | Primary | Mandatory |
| Pressure management | To supply sufficient pressure and thus quantity of water, the pumps and pump stations must be scheduled appropriately. | Primary | Mandatory |
| Leak/ burst detection | Considering the uncertainty with regards to water supply and the aging infrastructure, it is important to detect leaks and burst quickly and accurately to minimise losses. | Primary | Mandatory |
| Demand characterisation | For certain areas, particularly industrial areas, there is a need to characterise the demand to allows for services such as anomaly detection. | Primary | Desirable |
| Maintenance scheduling | Effective maintenance and repairs scheduling is required to fully utilise the available resources and expertise, while also maintaining adequate supply. | Primary | Desirable |
| Reporting | The government expects regular (weekly to monthly) updates of the water quality to ensure that it is within acceptable bounds. Therefore, having a reporting service would save managers' time so that they can focus elsewhere. | Primary | Highly desirable |

Furthermore, the NFRs are listed in Table 20, where the needs are given in the grey rows, followed by the related NFR, a rationale, an NFR grouping and an implications description for the NFRs (as discussed in Section 7.3).

**Table 20: Non-functional requirements for the water distribution system**

| | |
|---|---|
| **Need** | **Provide for proprietary technologies, integrate information silos and integrate with existing data systems. (Related to N3, N7 and N23 )** |
| **Related NFRs** | Compatibility, composition constraint, service provider support. |
| **Rationale for NFRs** | The variety of monitoring and control systems that have been implemented by different third-party providers have resulted in data silos. The DTs must integrate with the third-party technologies to create a holistic view of the WDS and this requires interoperability. |
| **NFR grouping** | Quality attribute, development constraints. |
| **Implication of NFRs** | Use the compatibility design pattern to help overcome heterogenous communication mechanisms and data formats. The composition of the DTs must provide for the inclusion of APIs and webhooks to interface with the third-party providers. |
| **Need** | **Provide for legacy systems and facilitate effective system maintenance and extension by allowing for efficient reconfiguration, possibly through some system automation. (Related to N6, N9, N14, N16)** |
| **Related NFR** | Maintainability, solution constraint. |
| **Rationale for NFR** | The WDS consists of old and new infrastructure while the variety of monitoring and control systems also differ in their technological maturity. The differences in maturity are a result of periodic sensor installations and system replacements etc. over the lifetime of the WDS. Therefore, the DTs must allow for such differences in technological maturity by facilitating easy system maintenance and by allowing for easy reconfigurations. |
| **NFR grouping** | Quality attribute, development constraint. |
| **Implication of NFR** | Use the maintainability design pattern. The system must also make provision for legacy systems that do not have internet connectivity (many of the subsystems use short range communication). |
| **Need** | **Provide reliable and highly available services. (Related to N15, N18)** |
| **Related NFR** | Reliability. |
| **Rationale for NFR** | The WDS is a critical piece of infrastructure and the fast and efficient identification of leakages or other anomalies is important. |
| **NFR grouping** | Quality attribute. |
| **Implication of NFR** | Use the reliability design pattern. |
| **Need** | **The system must provide for data veracity, data accuracy and data persistence management. (Related to N20)** |
| **Related NFR** | Reliability, solution constraint. |

| | |
|---|---|
| **Rationale for NFR** | The DT must ensure that the services and models receive all the relevant data for reliable supply and demand estimates. Therefore, the DT must ensure high veracity data, accurate data and the data must be managed for long-term use. |
| **NFR grouping** | Quality attribute, development constraint. |
| **Implication of NFR** | Use the reliability design pattern. The solution space is constrained in its ability to aggregate since high veracity and high accuracy data is usually post-storage aggregated. Persistent data must be managed and periodically reduced. |
| **Need** | **The distributed nature of the system results in the use of wireless networks. (Related to N26)** |
| **Related NFR** | Composition constraint. |
| **Rationale for NFR** | WDSs often make use of WSNs to capture data and these networks typically make use of short-range communications to send data to a sink node (a sink node collects the data from multiple sensors and sends it to a server or to the cloud). Therefore, the system of DTs must provide for the use of wireless networking infrastructure. |
| **NFR grouping** | Development constraint. |
| **Implication of NFR** | The reliability design pattern should be used to compensate for the uncertainty of the network ability. The network infrastructure is largely constrained to mobile network connections. |
| **Need** | **Provide for resource constrained devices. (Related to N25)** |
| **Related NFR** | Performance efficiency, solution constraint. |
| **Rationale for NFR** | The sensors and devices that take measurements on the pipe network are typically battery powered and thus they are limited in terms of their processing ability. |
| **NFR grouping** | Quality attribute, development constraint. |
| **Implication of NFR** | Use the performance efficiency design pattern for efficient resource utilisation and limit the load placed on the sensors and the battery powered devices. |

### 12.1.3 Physical system decomposition

The basic components of a WDS are presented in Figure 7, where the elements are arranged in a simple diagram. These elements are typically used within hydraulic modelling software, such as EPANET, to build hydraulic models. Junctions are links in the network that join pipes or that mark a point where water enters or exits the network. Reservoirs are water storage units and typically they also have water treatment facilities, tanks are simple water storage units, pumps are links that impart energy to the water and valves are links that limit pressure or flow. Pipes are the conduits that transport the water and there are multiple types, such as transmission lines, arterial mains, distribution mains, etc.

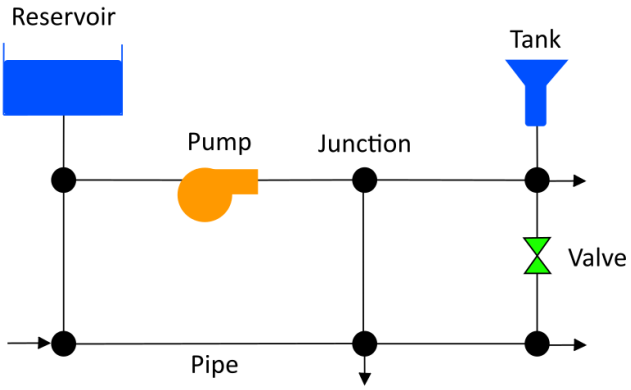



**Figure 7: Basic water distribution system elements. Adapted from (Rossman, 2000)**

The KY12 WDS physical system diagram is presented in Figure 8 where all the hydraulic modelling elements are present except that the junctions are not explicitly shown. In Figure 8, the “pump” symbol represents a pump station which consists of more than one pump. In total the KY12 WDS consists of one reservoir with a water treatment facility, seven water tanks, 21 pressure reducing valves, 16 pumps spread across four pumping stations, 2262 junctions and 649,4 km of piping.

Furthermore, the WDS has been divided into operational zones which are arbitrary divisions used to manage the size of the WDS. Operational zones are commonly used in practice, although other terms, such as district metering area, are used in similar contexts. In practice it is also common to use pressure flow zones to divide the WDS into areas according to similar pressure requirements (these areas are typically at the same elevation). However, this case study adopts the operational zones concept because it allows for divisions according to parameters other than pressure.

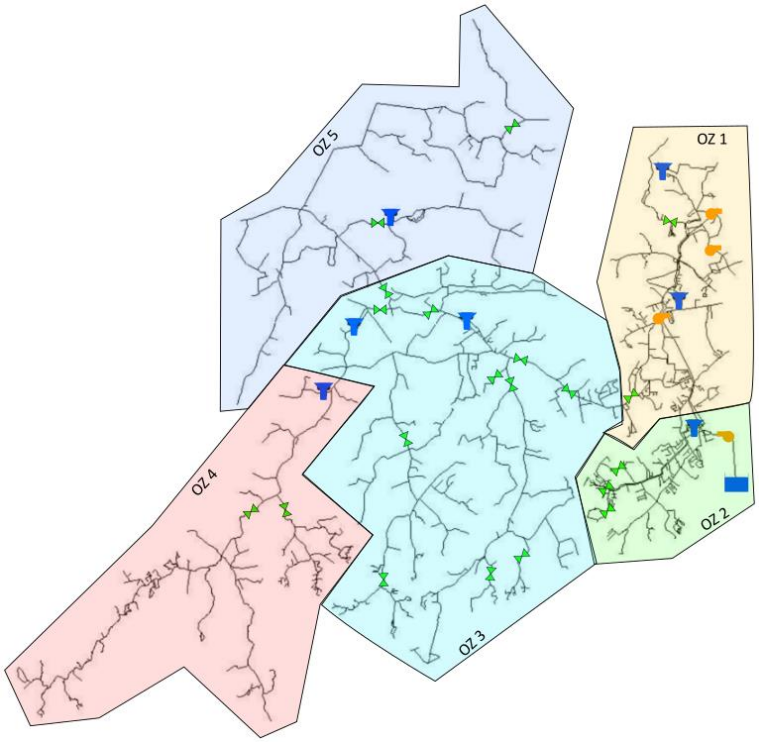



**Figure 8: KY12 water distribution system decomposition. (Adapted from Hoagland, n.d.)**

Table 21 presents the formal span of reality of an operational zone, but for the sake of brevity, the other physical components are only briefly discussed. The elementary components are the pipes, values, pumps and tanks. Tanks are relatively simple and typically only have a water level measurement. Pipes (and junctions), as well as pumps, can be fitted with pressure meters, flow velocity meters and vibration sensors.

A reservoir with a water treatment facility is a complex system and many measurements can be taken to monitor and control the treatment process and water quality throughput the process. This case study does not consider the water treatment facility in detail and is only concerned with the reservoir's water level. Pump stations typically consist of a collection of pumps, pipes and valves and the measurements that go along with them. The various sensors are also scattered across the pipe network and are typically located in key areas, such as areas where the elevation changes significantly. Furthermore, all the components also have static data such as longitude, latitude and elevation values.

The frequency of data capturing within the WDS can typically be between a measurement every ten minutes to one measurement a day. However, for some components, such as the pumps, measurements can be taken more frequently.

**Table 21: Span of reality of water distribution network operational zone.**

| Physical component | Operational zone (OZ) 1 |
|---|---|
| **Physical system scope** | Two water tanks, two pressure reducing values, three pumping stations, unknown length of pipe, unknown number of junctions. |
| **Data characterisation (Data granularity) of data recorded/ generated by physical component** | None |
| **Data characterisation (Data granularity) of data within physical component** | Pump station data: Number of pumps, number of valves, pipe data, valve data.<br><br>For each pump: pressure at inlet (kPa), inlet flow velocity (m/s), pressure at outlet (kPa), outlet flow velocity (m/s), power usage (kWh), status.<br><br>Water tank data: Water level (m), volume (kL).<br><br>Valve data: Pressure at inlet (kPa), pressure at outlet (kPa), status.<br><br>Pipe data: Length (km), diameter (cm), material, pressure at node (kPa), flow velocity at node (m/s), vibrational data. |
| **Data format** | Unknown |
| **Communication** | Sigfox and LoRaWAN. Other protocols not known |
| **Considerations and Constraints (Capacity for interaction)** | OZ 1 shares a boundary with OZ 2, where two pipes are bisected. OZ 1 also shares a boundary with OZ 3 where one pipe is bisected. |

### 12.1.4 Services allocation

**Service identification and characterisation:**

Many services can be developed with the data from the WDS. The services suggested for the WDS are briefly discussed below except for the virtual sensor service, which is used as an example for the extensive service characterisation. The suggested services were identified from the scenario in Section 12.1.1, as well as by using Table 15 in Section 8.1.

- Benchmarking (which is similar to the fingerprint service in Table 15) is a service that compares an operational zone's demand and supply ability with

its history, as well as with other operational zones, to better estimate demand and to detect trends.

- Anomaly (which is a service in Table 15) is a service to detect water leaks and burst pipes, as well as other irregular component operations. Examples of irregular operations include irregular pressures at nodes or in pumps, low water tank or reservoir levels, unsafe water quality, the detection of cavitation, etc. The benchmarking service is similar to the anomaly service, but the anomaly service is used to detect irregular operation, whereas the benchmarking service is used to detect irregular demand or supply.
- Scout (which is a service in Table 15) is a service that simulates future behaviour of the WDS or the behaviour of a sub-network. Simulation programs, such as EPANET, are used to design WDSs, where the supply of a WDS can be estimated to determine if a specified demand can be met. The scout service would likely just build upon an existing simulation application, such as EPANET, to make the simulation more applicable the WDS being represented.
- Optimised maintenance scheduling is a service intended to schedule maintenance to make optimal use of the available resources. This service will likely require information generated by the other services, as well as information about the available personnel, costs, etc. For example, the anomaly service could be queried to identify where maintenance is needed, while the scout service could be queried to determine which repairs would be most beneficial. The maintenance scheduling service can then use this information to assign the appropriate technician for the job, record that the job was executed and verify that the repair was effective.
- Reporting is a service that is intended to automate the reporting of data to the government. The reporting is primarily concerned with water quality data.
- Pump scheduling is a service that schedules pump operation to maintain adequate pressure throughout the WDS.

**Virtual sensor service:**

Description: The virtual sensor service can make use of existing modelling frameworks, such as EPANET, to infer various values at any point within the system based on current sensor inputs. This is required since fitting sensors to all 2396 pipes and all 2262 junctions is not feasible. The virtual sensor service will likely serve as an input to the benchmarking, scout and reporting services.

Related primary functional requirements: Supply estimation, demand estimation, pressure management.

Related secondary functional requirements: None identified yet.

Required physical scope: Operational zone with known boundary node values or full WDS.

Required data granularity:

- Data features: From pipes and junctions – size, pressure and flow velocity values. From pump stations – pressure and flow velocity values. From valves – pressure and flow velocity values. Form tanks – water level and volume. From reservoir – water level and volume.
- Required sampling frequency: For pipes and junctions, pump stations and valves – measurement once every ten minutes is the norm. For tanks and reservoir – measurements once per day is the norm.

Service characteristics:

- Required data update frequency: Near real-time (which is the same rate as the sampling frequency for this case).
- Degree of user interaction: Periodic and possibly infrequent. The virtual sensing service serves more as an input to other services.
- Intensity of interaction: Spatially focussed.
- Persistence: Periodic or event-based

Constraints and considerations: The virtual sensing service makes use of the current sensor measurements to infer measurements at other points in the WDS. Therefore, if the service is provided with or has access to the historical sensor measurements, the inferred sensor measurements can be recalculated as opposed to stored. This approach will further help to ease the data persistence management required for the actual sensor measurements. However, if latencies become a problem within the user interface, the virtual sensor values may need to be periodically pre-calculated and stored. This may be applicable if virtual sensing is done for a lengthy historical period.

**DT identification:**

Considering the physical system decomposition, the following digital twins may be feasible:

- DTIs of pumps. The pumps are critical components within the WDS and their operation is closely monitored using pressure meters, flow velocity meters and vibrations sensors. Therefore, each pump may need its own DTI for reliable representation and high fidelity.
- DTIs of clusters of pipes, junctions, valves and tanks. The geographic distribution and prevalence of short-range wireless communications likely means that there are multiple clusters of sensors spread across the WDS.

Therefore, DTIs can be designed to represent the elements being represented by such clusters, where a cluster does not necessarily have to contain all the elements. However, the exact distribution of the sensors is unknown and this may need to be verified.

- DTAs of pump stations. The pump stations are a collection of pumps, pipes, junctions and valves that must work in unison. Therefore, a DTA of the pump station, which aggregates from the individual pump DTIs and an internal cluster DTI, is recommended.
- DTA of the reservoir and treatment facility. The reservoir and treatment facility are together likely to be a DTA depending on the complexity of the treatment facility. However, the KY12 WDS dataset does not provide data or metadata about the water treatment facility, which makes is difficult to determine the complexity and composition of the treatment facility.
- DTAs of the operational zones. The operational zones are arbitrary divisions of the WDS, but the divisions help manage the data and, importantly, help with supply and demand estimations through benchmarking. The operational zone divisions are chosen to minimise the interconnections between zones, i.e. the divisions were chosen to cross the least number of pipes possible. Furthermore, the dividing lines are drawn close to potential sensor clusters so that the boundary values can be determined by sensor values.
- DTA of the full WDS. The WDS DTA provides a full overview of the WDS, including the supply and demand of the entire WDS, as well as representing the interconnections between the operational zones.

**Services allocation:**

Table 22 summarises the services allocation, where all potential hosting position for each service are indicated.

**Table 22: Potential services allocation, for WDS, based on span of reality.**

| Services | Pump DTI | Cluster DTI | Pump station DTA | Reservoir DTA | OZ* DTA | WDS DTA | Services network |
|---|---|---|---|---|---|---|---|
| Virtual sensor | | | | | X | X | X |
| Benchmarking | | | | | | X | X |
| Anomaly | X | X | X | X | X | X | X |
| Scout | | | | | | X | X |
| Optimised maintenance scheduling | | | | | | | X |
| Reporting | | | | X | | | X |

| Services | Pump DTI | Cluster DTI | Pump station DTA | Reservoir DTA | OZ* DTA | WDS DTA | Services network |
|---|---|---|---|---|---|---|---|
| Pump scheduling | | | X | | X | X | X |

*OZ: operational zone

Based on Table 22 and considering that initially services are allocated to the lowest level DT that can host the service, the services allocation is:

- Anomaly -> Pump DTI
- Anomaly -> Cluster DTI
- Pump scheduling -> Pump station DTA
- Reporting -> Reservoir DTA
- Virtual sensor -> OZ DTA
- Benchmarking and scout -> WDS DTA
- Maintenance scheduling -> Service network

Furthermore, the only DT that is hosting more than one service is the WDS DTA. However, these services do not have any obvious conflicts and thus no separation is required.

### 12.1.5 Design pattern selection and application

As discussed in Section 2.3, the SLADTA is used as the reference architecture for the internal structure of each DT. Based on the needs and constraints analysis, the quality attributes that are most important for this case study are maintainability, reliability and compatibility. Of these, maintainability is considered the most important because the exact data characteristics of the system are not known and thus future changes will be necessary. Furthermore, the WDS has a history of incremental expansion and development and thus the system must make provision for such changes.

Reliability is also an important factor in the lower-level WDS components and their accompanying services. The pumps and pump stations, the tanks, critical junctions and the transmission and arterial pipes are all important components and thus the services related to their real-time operation must also be reliable.

Finally, compatibility is important because of the variety of communication mechanisms that are used within the WDS. For the communication heterogeneity, the DTIs would have to implement some homogenisation logic before sending the data further.

The proposed internal architectures for the DTs are presented Figure 9 and the aggregation and communication architecture is presented in Figure 10. With regards to the internal architectures, the pump DTIs' anomaly detection service has been separated into anomaly training (hosted in the cloud portion of the DTI) and anomaly model (hosted on the local portion of the DTI). Anomaly training refers to data exploration and training functions related to building an anomaly detection model. The anomaly detection model is the model implemented to identify anomalies during operation. This separation has been made to ensure that the anomaly detection has enough storage and processing power to perform training, while also ensuring reliability of the service execution during operation despite network failures.

Furthermore, the pumps and pump stations each have locally hosted publish-subscribe (pub-sub) clients to allow for local network communication between the pump DTIs and pump station DTA. The pump station also has a watchdog service (as proposed by the reliability design pattern) that periodically pings the pump DTIs and the pump scheduling service to ensure that all the components are active. The middleware client (present in all DTs except the pump DTI) is the common communication mechanism within the DT hierarchy to ensure interoperable and reliable messaging within the DT hierarchy. Furthermore, according to the SLADTA, where a DT has layers hosted locally and in the cloud, data transfer from local layers to cloud layers is handled through Layer 4.

The cluster DTI also has a webhook component within its Layer 4. The webhook is an HTTP message that gets pushed to a pre-defined endpoint based on the occurrence of events. This mechanism follows an asynchronous request-response pattern (discussed in Section 10.3). This is the pattern that some of the pre-defined communication mechanisms, such as Sigfox, use to make their short-range communication data available over the internet and thus it is mirrored here.

In terms of the separation of concerns, the generated architecture spreads the services across multiple DTs, where each DT has a clear purpose in fulfilling the needs of the services attached to it. There are also multiple levels of aggregation which help make the architecture more maintainable by lowering the unit complexity of each of the DTs and services (as discussed in Section 5.2.3).

In terms of the services hosting, most of the services (except for the maintenance scheduling) are primarily focussed on the physical system and thus they can be encapsulated in DTs. The encapsulation should allow for more predictable functioning, which is good for reliability, while also reducing the need for service orchestration. Furthermore, the maintenance scheduling service requires data that is not captured by the DTs, such as personnel and financial data, and thus this service is located within the service network.

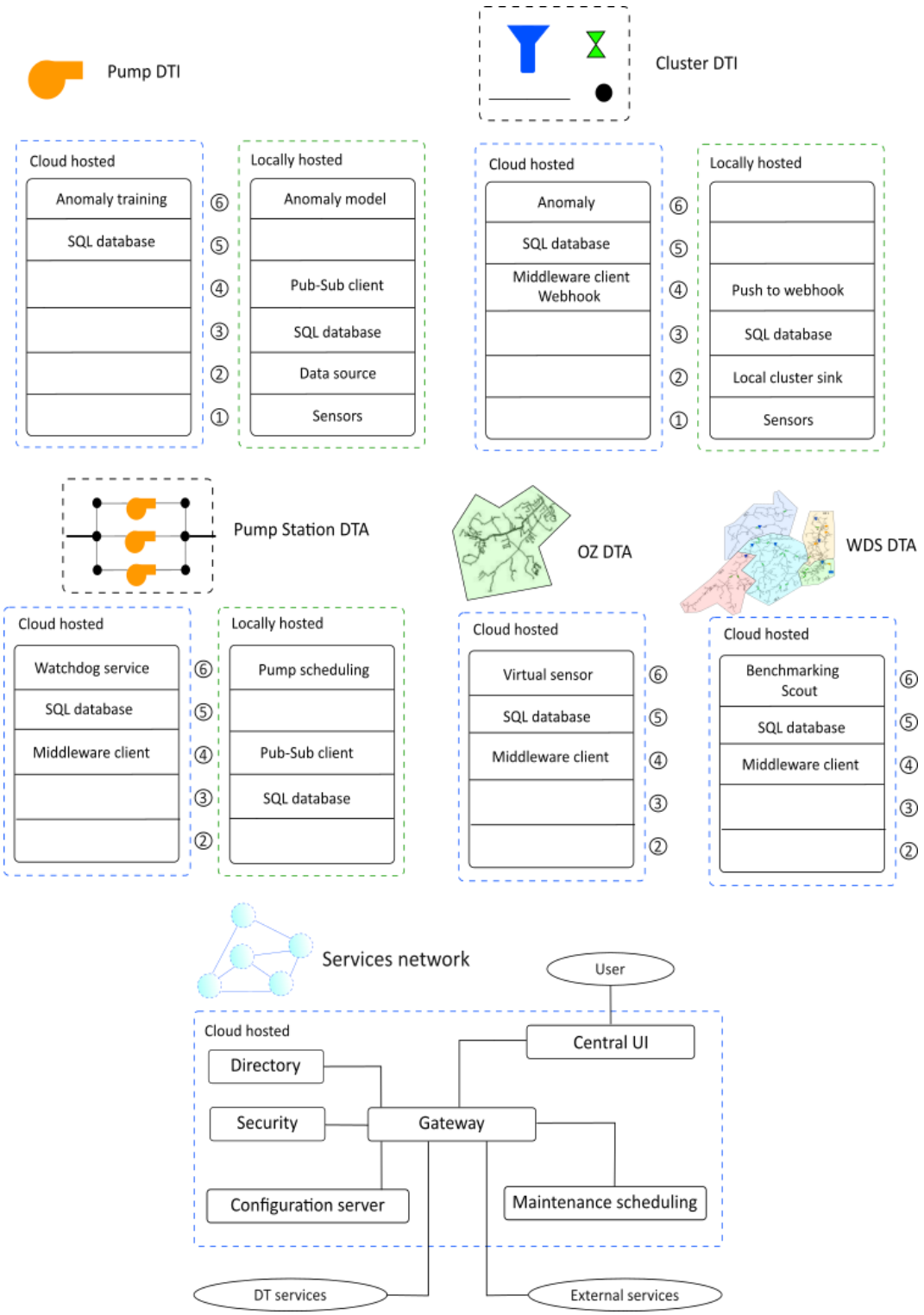



**Figure 9: Internal architectures for the water distribution system DTs.**

For the communication architecture in Figure 10, there are three primary communication facilitators: the local broker, the message-oriented middleware and the gateway. The local broker is a locally hosted, lightweight, publish-subscribe broker which is solely dedicated to facilitating communication within a pump station. The pump DTIs and the pump station DTA all have local network presence. Therefore, pre-storage aggregation is sensible to allow for reliable transmission, while the short-term local storage can used to capture, for example, two weeks' worth of data to still provide data redundancy.

The message-oriented middleware and gateway are both cloud hosted communication mechanisms. The message-oriented middleware is responsible for reliable and secure communication within the DT hierarchy, while the gateway serves as a proxy for external services and service within the services network. It would be possible to join them, but this would be less secure, less reliable and more prone to becoming a bottleneck. Furthermore, the message-oriented middleware is expected to facilitate publish-subscribe messaging, whereas the gateway is expected to facilitate service requests through REST APIs.

The local broker and message-oriented middleware, both of which are in the DT hierarchy, make use of periodic micro-batch processing. This is because the frequency of sensor readings can vary significantly depending on the phenomena being measured and many of these are relatively infrequent (measurements are in the order of minutes to tens of minutes). The gateway, on the other hand, makes use of event-driven batch processing, where events are requests from other service or from a primary user.

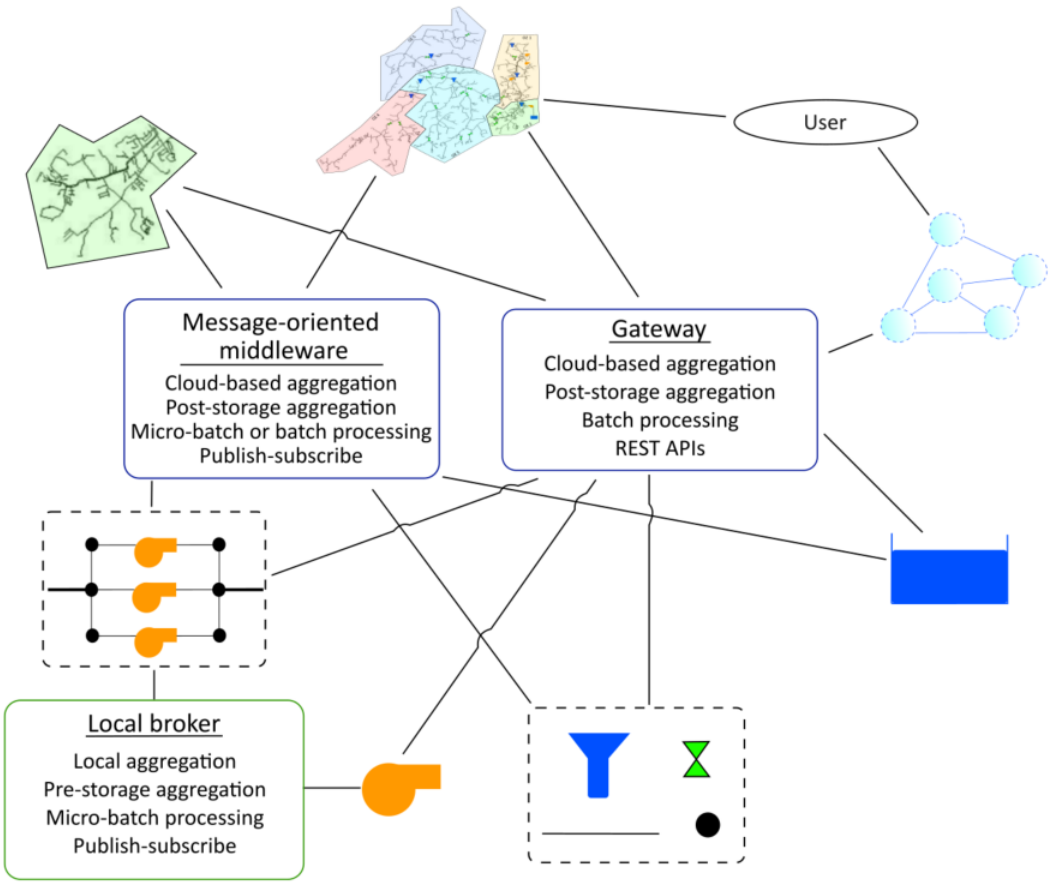



**Figure 10: Communication architecture for the water distribution system's DTs.**

### 12.1.6 Discussion

The proposed WDS architecture separates the service across all six DTs, as well as delegating some services to the services network. This is in accordance with the reliability and maintainability design patterns to distribute the service offerings across multiple DTs to allow for better fault isolation and separation of concerns. Reliability is also improved by hosting most of the services in DTs, which encapsulate the data and functionality required for the services, to minimise external dependencies. The minimised dependencies are also in accordance with the maintainability design pattern.

Furthermore, local, pre-storage aggregation is facilitated between the pump DTIs and the pump station DTA using a local broker. The local network communication provides reliable communications, while the broker decouples the pump DTIs and pump station DTA to further improve fault isolation. Although pre-storage aggregation is used, data redundancy is ensured through the local, short-term storage solution. Making use of the short-term storage for data redundancy is

feasible in this case study because the data sampling frequency is relatively slow. The pump station DTA also makes use of a watchdog, service to further improve reliability.

The message-oriented middleware is hosted in the cloud, using a container, to provide secure and reliable communication for all the DTs in the hierarchy. Hosting the middleware in a container in the cloud ensures scalability and easier replication with load balancing. This is in accordance with the maintainability design pattern, which suggests making use of cloud support for easier management. The message-oriented middleware also decouples the DTs to provide fault isolation.

Finally, the individual DTs use an SQL database for better reliability and consistency, as recommended by the reliability design pattern. The services network includes central UI, directory service and configurations server to further improve the maintainability of the system of DTs.

## 12.2 Smart City

### 12.2.1 Scenario

The concept of smart city refers to the combining of information and communication technology (ICT) and other technologies to improve the quality of life (QoL) for the citizens in the city, to improve the competitiveness of the city and to improve the operational efficacy of urban services, while ensuring the availability of social, economic and environmental resources for the current generation and for generations to come (Kondepudi, Ramanarayanan, Jain, Singh, Agarwal, Kumar, *et al.*, 2014). Another description of smart cities is: to seek to improve the institutional (governance), social (intellectual and human capital and QoL), economic (economic and job growth) and physical (natural and manufactured resources) infrastructure of the city, in a sustainable way, to ultimately improve the wellbeing of the citizens in the city despite increases in population (Mohanty, Choppali & Kougianos, 2016).

Smart cities are typically composed of other smart systems and these smart sub-systems can vary from city to city depending on how well developed the city is. Some of the most common smart sub-systems are (Silva, Khan & Han, 2018):

- Smart energy which refers to the ability to produce enough energy in a sustainable manner with minimal adverse effects to the environment and community.
- Smart buildings which primarily refers to the management of buildings to reduce energy consumptions and to improve the physical security of the building and thus the people in it.

- Smart waste management which refers to the effective collection, disposal, recycling and recovery of waste within the city.
- Smart transport which refers to the effective management of traffic flow within the city.

Other smart sub-systems that have been mentioned in literature include smart healthcare, smart water management, smart warehouses and smart supply chains.

Examples of smart cities include Smart London (United Kingdom) and Smart Santander (Spain). Smart London is one of the top ranked smart cities according to Berrone, Ricart, Carrasco, *et al.* (2018). Some of the most renowned initiatives that Smart London have employed include traffic congestion management through street monitoring and number plate recognition (for traffic modelling purposes) while smart public transport schemes are also being heavily invested in. Furthermore, Smart London also gets the citizens involved by providing platforms where citizens can comment and provide feedback. Large amounts of data captured within Smart London are also made available to citizens through London Datastore (Greater London Authority, 2021) to stimulate application development and innovation among citizens.

Smart Santander has implemented the Smart Santander testbed (Smart Santander, n.d.) which is a large-scale IoT infrastructure deployment available to researchers. The Smart Santander testbed is primarily focussed on environment and traffic monitoring. It includes 2000 IoT devices for environment monitoring, 150 mobile environment monitoring units on public vehicles, 400 parking sensors, 60 traffic monitoring sensors and 50 irrigation related sensors for the parks.

The Smart Santander IoT infrastructure deployment makes use of three types of devices: 1) *IoT nodes* which have various sensors built in, 2) *repeaters* which receive data from IoT nodes and forward them to the *gateway* to improve the range of the WSN and 3) the gateway which is a WiFi, ethernet or GPRS enabled device that either forwards the data to a central command unit or stores the data and makes it available through a web interface. Static deployments are typically located on streetlights and building facades, whereas mobile deployments are located on public transport vehicles.

For this case study, as hypothetical smart city will be considered based on Smart London and Smart Santander. The hypothetical case study will be considered at a higher level than the WDS case study presented in Section 12.1. The WDS would typically be a sub-system within a smart city and thus the smart city case study will focus on demonstrating the design framework on a system-of systems.

### 12.2.2 Needs and constraints analysis

The FRs mentioned in Table 23 are general FRs that would each likely be part of multiple services. For example, remote monitoring allows an operator to recognise traffic congestion, which can then be acted on by sending notifications to public servants, such as police officers, while sending directed messages at an ambulance to help them follow an alternative route. Furthermore, these are considered to be fundamental functions that will allow for a multitude of other functions.

**Table 23: General functional requirements for a smart city**

| High-level functional requirements | Description | Group (Primary or secondary) | Priority |
|---|---|---|---|
| Remote monitoring | Allow users to monitor activity and conditions within the city from a central command position. | Primary | Mandatory |
| Notification | Notify citizens, workers or authorities about activities or conditions within the city that are relevant to them. | Primary | Mandatory |
| Directed communication | Allow for directed communications to selected users to enable particular services. For example, notifying an ambulance about an alternative, less congested route. | Primary | Mandatory |
| Exploratory analytics | Allow various users to analyse data about various parts of the city for various services. | Primary | Highly desirable |

Furthermore, Table 24 provides some of the NFRs that are expected to be most applicable in the smart city context.

**Table 24: Non-functional requirements for smart cities.**

| **Need** | **Accommodate multiple stakeholders with different levels of access to different subsets of data. (Related to N2)** |
|---|---|
| **Related NFRs** | Security. |
| **Rationale for NFRs** | A smart city is expected to consist of multiple DTs that represent multiple infrastructure systems that are each managed by a different group of people. Furthermore, some data is expected to be available to the public, some data will only be for internal use and some data may even be restricted to individuals. Secure data sharing and data privacy are some of the primary concerns in smart cities. (Silva *et al.*, 2018) |
| **NFR grouping** | Quality attribute. |

| | |
|---|---|
| **Implication of NFRs** | Use the security design pattern to ensure secure data sharing and storage. |
| **Need** | **Integrate with new and existing data systems, allow for retrofitting and allow for the integration of humans. (N6, N7, N8)** |
| **Related NFRs** | Compatibility, security. |
| **Rationale for NFRs** | Interoperability is one of the most cited concerns in smart city research because of the variety of data, the variety of data acquisition techniques and the variety of vendors involved in the system composition (Silva *et al.*, 2018). Furthermore, smart cities can also collect data from participating citizens which further emphasises the need for data privacy and security. |
| **NFR grouping** | Quality attributes. |
| **Implication of NFRs** | Use the compatibility and security design patterns. |
| **Need** | **Allow for easy system maintenance and extension. (N14)** |
| **Related NFRs** | Maintainability. |
| **Rationale for NFRs** | A smart city deployment is expected to be extended over the span of years. Furthermore, the city itself will also change and thus the system of DTs must be able to change along with the city. |
| **NFR grouping** | Quality attribute. |
| **Implication of NFRs** | Use the maintainability design pattern. |
| **Need** | **Data management issues that are applicable within the smart city are veracity, consistency, persistence and synchronisation. (Related to N20)** |
| **Related NFRs** | Solution and implementation constraints. |
| **Rationale for NFRs** | Given the variety of contributors and the variety of data acquisition methods, ensuring the trustworthiness of the data may become challenging. Some data will also likely be shared amongst multiple users and it is important that these users receive the same data. The volume of the data will make data persistence a pressing issue and given that variety of sensors, as well as the intermittent network availability, data synchronisation will be required. |
| **NFR grouping** | Development constraints. |
| **Implication of NFRs** | To ensure that data is trustworthy, certain measures will need to be put in place. Such measures may include imposing a data standard, implementing cross-validation of sensor values (comparing sensor values with the values from other nearby sensors) and implementing rule-based data cleaning.<br>The data consistency issue will require database support for consistency, the persistence issue may require automated data reduction on older data (such as the multi-temperature data management paradigm) and data synchronisation will limit the data transfers to micro-batch transfers. |
| **Need** | **Facilitate heterogeneous data handling. (Related to N23)** |

| | |
|---|---|
| **Related NFRs** | Compatibility. |
| **Rationale for NFRs** | Smart cities can collect data from sensors, cameras, mobile devices and from people who comment (Smart London provides multiple methods for people to comment on their experiences within the city). This means there will be highly structured data from the sensors, as well as highly unstructured data in the form of user comments. |
| **NFR grouping** | Quality attribute. |
| **Implication of NFRs** | Use the compatibility design pattern. |
| **Need** | **Provide for resource constrained devices. (N25)** |
| **Related NFRs** | Solution constraint, Performance efficiency. |
| **Rationale for NFRs** | Many of the sensing devices used in Smart Santander are battery powered devices. This is covered in more detail in Section 12.2.3. |
| **NFR grouping** | Development constraint, quality attribute. |
| **Implication of NFRs** | Resource constrained devices are limited in their processing ability and typically have very infrequent sensor readings. Furthermore, in Smart Santander, these devices also make use of power efficient short-range communications that are designed to the IEEE 802.15.4 protocol. |
| **Need** | **Provide for intermittent network availability. (Related to N26)** |
| **Related NFRs** | Solution constraint, reliability. |
| **Rationale for NFRs** | The mobile devices and sensors in Smart Santander either make use of General Packet Radio Service (GPRS), which is a mobile data standard that makes use of 2G and 3G cellular communication, or mobile devices connect to a gateway when within range of one. |
| **NFR grouping** | Development constraint, quality attribute. |
| **Implication of NFRs** | Mobile sensing devices must store data so that when network connectivity is not available, the data does not go lost. This also means that micro-batch transfers are common for such mobile devices. |

This case study will focus on the compatibility and security design patterns because interoperability and security are two of the most cited concerns in smart cities (Silva *et al.*, 2018). Further consideration will also be given to maintainability of the smart city system of DTs.

### 12.2.3 Physical system decomposition

The physical system decomposition of the city, can be done in various ways, as is the case for most complex systems. However, what makes the decomposition of the smart city difficult is that some sub-systems are spatially concentrated and static, whereas other sub-systems are widely dispersed and others are mobile.

For the physical decomposition in this case study, a spatially focussed decomposition will be followed, but keeping in mind the functional decomposition as well. This case study considers the following high-level functional sub-systems of a smart city:

- Energy systems: Infrastructure related to energy provision. This includes distribution infrastructure, sub-stations, metering systems, etc.
- Water distribution systems: Infrastructure related to water provisioning. This includes water storage, pump stations, piping networks, etc.
- Transport systems: Infrastructure related to the transportation of people and goods. This includes roads, railways, public transport mechanisms, etc.
- Waste management systems: Infrastructure related to the collection, disposal, recycling and recovery of refuse (municipal solid waste) within the city. This includes refuse removal trucks and teams, landfills, recycling plants, etc.
- Sanitation systems: Infrastructure related to the removal of sewage. This includes the piping infrastructure, treatment plants, etc.
- Emergency systems: Infrastructure related to emergency response and physical safety and security. This includes health services, law enforcement and fire and rescue services.
- Goods distribution systems: Infrastructure related to the distribution of goods. This includes warehouses, shopping centres, retail outlets, etc.

As for the physical decomposition, cities are typically divided into arbitrary sections by municipalities to help with governance. For example, Stellenbosch municipality is divided into 22 wards and the city of Cape Town is divided into 116 wards, where each ward has a different councillor that presides over it. These wards are spatially divided sections that can be large (more than 100 $km^2$) or small (0,30 $km^2$) and they are related to parameters such as the population density. Within wards there are various combinations of the following components:

- Citizens: The people within the city are ultimately the most important part of the city. People are a dynamic, multifaceted and fully integrated part of any city. The city is there to help serve the needs of the people, but the people are also largely the means through which a city achieves its functions.
- Buildings: Smart buildings are an important component in smart cities and smart buildings are primarily focussed on optimised power consumption and improved physical safety and security. Furthermore, some buildings have specialised functions, such as hospitals, university buildings, shopping centres, etc.
- Vehicles: Smart vehicles are cars, busses, taxis, etc. that capture and transmit data. In Smart Santander, public transport vehicles are used to capture data

about the environment (such as ambient temperature, luminosity, air quality, etc.), as well as to capture statistics about public transport usage. As with buildings, some vehicles can be specialised vehicles, such as ambulances or police cars, in which case their contents may also need to be logged. Furthermore, vehicles can monitor and keep track of their own usage statistics, such as speed travelled, distance travelled, maintenance history, etc.

- Static infrastructure components: Static infrastructure refers to small infrastructure components such as streetlights and traffic lights.
- Land use areas: An area whose components share a purpose, such as a housing estate, a golf course or a business park. Such an area can consist of multiple citizens, vehicles, buildings, etc. but they are typically still smaller than a ward.
- Utility networks: Distinct infrastructure systems that perform core functions related to cities, such as water distribution systems, sewage systems, power distribution systems, urban transport systems etc. These are infrastructure systems that form continuous networks across a city and thus they can span across multiple wards.

For the span of reality, the Smart Santander IoT deployment has been used as a reference (Smart Santander, n.d.), while further possibilities are also considered based on literature. A map of the Smart Santander IoT infrastructure deployment is available at: https://maps.smartsantander.eu. For brevity, only a high-level data characterisation of a hypothetical ward will be considered here.

**Table 25: High-level span of reality of the data available in a city ward.**

| Physical component | Ward |
|---|---|
| Physical system scope | Citizens, buildings, vehicles, static infrastructure components, land use areas and a subsection of a utilities network. |
| Data characterisation (Data granularity) of data within physical component | Citizens – GPS coordinates, comments and feedback, smart wearables data.<br><br>Buildings – Water usage, electricity usage, security data (such as CCTV video), occupancy measurements, HVAC system information, temperature, luminosity, noise.<br><br>Vehicles – Vehicle related data (such as speed and distance travelled), public transport related data (such as passenger count), environment data (such as temperature, luminosity, noise, air quality).<br><br>Static infrastructure components – Electricity usage, environmental data, traffic data (such as parking space availability, traffic volume, road occupancy, vehicle speed or queue length).<br><br>Land use areas – Environmental data, irrigation data (in parks and gardens), traffic data (in parking areas or large roads), occupancy measurements, power consumption.<br><br>Utility network subsystem – Depends on the infrastructure but power consumption and water usage are likely measurements. |
| Data format | Unknown |
| Communication | (Based on the Santander IoT infrastructure for environment and traffic monitoring)<br><br>Sensing devices – IEEE 802.15.4 based protocols (e.g. Zigbee, 6LoWPAN, Thread)<br><br>Repeater devices - IEEE 802.15.4 based protocols<br><br>Gateway devices - IEEE 802.15.4 based protocols and WiFi or ethernet or GPRS. Specific protocols not defined. |
| Considerations and Constraints (Capacity for interaction) | None |

### 12.2.4 Services allocation

**Service identification and characterisation:**

The potential for services within a smart city is substantial and covering each possibility here would not be feasible. Therefore, the service characterisation here is rather going to focus on core services that would enable the creation of other services. The core service suggested here are:

- A mirror service: The mirror service here refers to the ability of the DT to capture, manage and present data in a meaningful way for decision-making regarding the physical system being represented. It also includes the ability to send information or commands back to the physical system if applicable. Furthermore, the mirror service is intended to allow for remote monitoring and control.
- An anomaly service: The anomaly service may not be applicable to all DTs, but it is expected to form part of most DTs and thus it is included in this list. The anomaly service, in this context, is a service that helps operators identify potential unusual activity and may also include an automated notification service. Some examples of what an anomaly service could be used for is to help identify traffic congestion, help to identify excessive power consumption or to help identify faulty hardware.
- System directory and navigation: The system directory is part of the management services mentioned in Section 10.2 but it is mentioned here because navigation between DTs will be important for a good user experience.

**DT identification:**

Based on the physical system decomposition the following DTs are identified:

- Citizens may each have their own DTs, but the composition of such a DT is uncertain. Citizens can form part of various other systems. For example, their data can be tracked as a worker or their medical data can be recorded for hospital use, etc. Furthermore, there would be strict security and privacy concerns, assuming the citizens provide permission to track the data. Therefore, this case study will not consider DTs of citizens, but will rather consider citizens as voluntary data sources to other DTs.
- Specialised buildings are likely to consist of multiple levels of DTs. For example, a hospital contains complex machinery, as well as specialised rooms and floors, that can each be represented if the need arises. Therefore, DTs of such buildings will be DTAs. Some less sophisticated buildings may only consist of a single DTI.

- Static infrastructure is likely to form part of a utilities network or land use area and thus this case study does not consider separate DTs for each static infrastructure component.
- Vehicles are each likely to have their own DT. As in Smart Santander, DTs of vehicles can capture data about the vehicles, as well as the environment. Furthermore, vehicle DTs can be designed to allow an automotive manufacturer to monitor vehicle performance to allow for better future designs. Some vehicles, such as ambulances and fire trucks, may also have important contents that need to be monitored, such as medicines or rescue supplies. Therefore, similar to citizens, vehicles are very versatile and a single vehicle can be applicable to many stakeholders.
- Land use areas may be represented through a DTI or DTA depending on the size of the area and the amount of data captured within the land use area. However, it is likely that a land use area would be a DTA that aggregates from multiple buildings, vehicles, citizens, etc. as well as other data sources, such as irrigation system data, static infrastructure data and environmental data.
- Utility networks are likely to have their own DTs, such as the WDS presented in Section 12.1. Therefore, there is likely to be a DTA of the utility network that would be integrated with for the smart city representation.
- Wards would be represented by a DTA that could aggregate data from various land use areas, buildings, vehicles, citizens, etc.
- The Smart City DTA will be a high-level DT that aggregates the data from various ward DTs to represent the city.

**Services allocation:**

This section does not explicitly assign services to DTs, but instead makes some recommendations that may be applicable within the smart city context. Services that are focussed on operational decision-making, such as services related to traffic congestion or irrigation control, would likely be better served within DTs. Furthermore, service that require confidential data are also suggested to form part of the DT hierarchy. Services that citizens can interact with, such as parking spot directories or citizen notification services, would likely be better served in the services network.

In terms of service separation, separation according to ownership is highly likely. In particular, the respective utility network systems are likely to be owned and managed by different stakeholders. Some buildings are also likely to separate services according to ownership. For example, a hospital may have some services for doctors and others for administrators. An office block that accommodates more than one business would also likely have separation according to ownership. Furthermore, the different wards each have a different councillor and each

councillor would likely be interested in different aspects of their ward. For example, one ward might experience heavy traffic congestion, another ward may be more concerned with water usage and another ward may be more concerned with crime statistics.

Separation according to scope complexity will also play role in a system such as a smart city. To allow for minimal interface complexity and interchangeability of components, the number of services assigned to a DT should be limited. Some DTs may even be dedicated to a single persistent service. This separation also creates a greater degree of service isolation which is beneficial for security. However, it should be noted that any DT is expected to be able to share data to authorised clients (people or programs) despite the number of services hosted within the DT.

Furthermore, the compatibility and security design patterns suggest a span of reality that has a specific purpose and thus a higher degree of separation is beneficial. The span of reality should also be defined to allow for additional processing and storage related to security functions.

### 12.2.5 Design pattern selection and application

This case study will focus on the compatibility and security design patterns because interoperability and security are two of the most cited concerns in smart cities (Silva *et al.*, 2018). However, the design should also allow for maintainability and system extension.

This case study will not consider the internal architecture of the DTs since there are many possibilities and will rather focus on the communication between the DTs, as well as the services in the services network. The proposed architecture is presented in Figure 11.

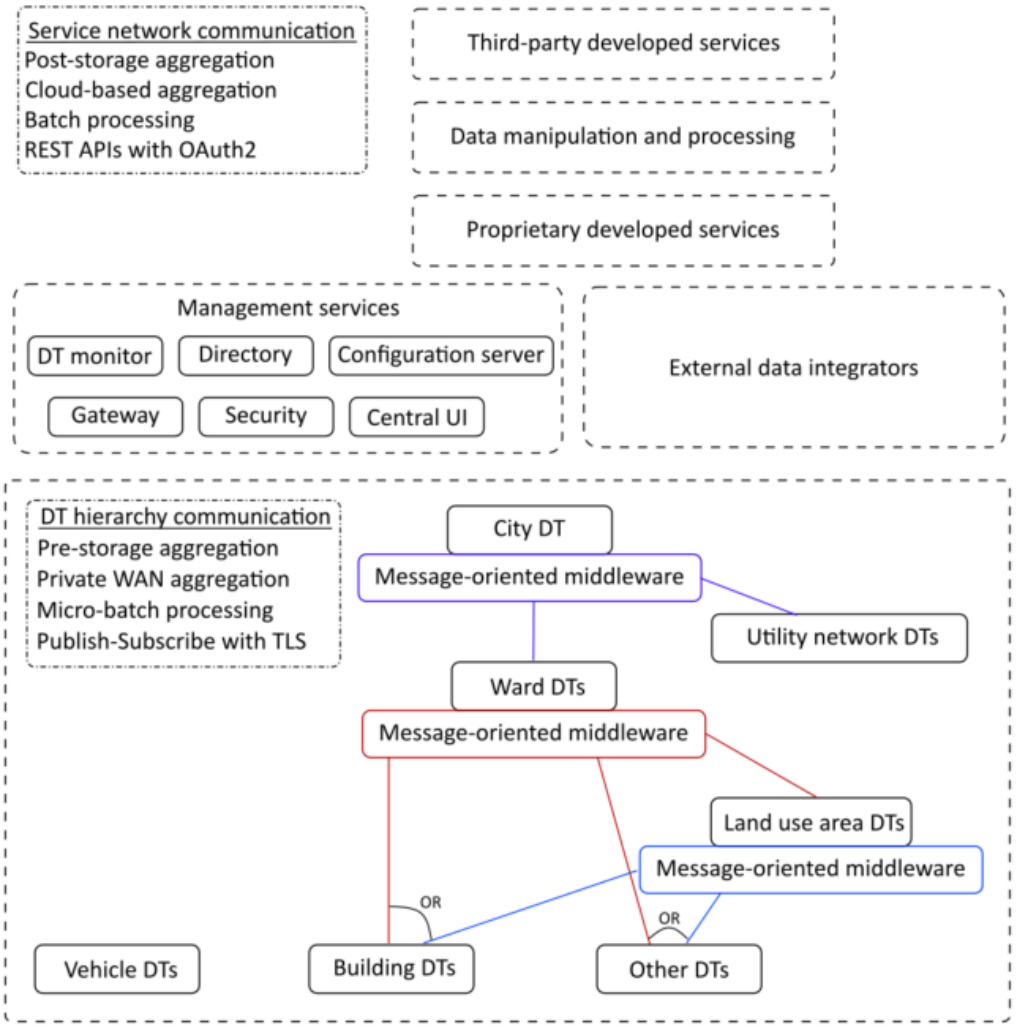



**Figure 11: Smart city aggregation and communication architecture.**

In terms of aggregation, the most likely scenario would be pre-storage aggregation within the DT hierarchy, where the DT has short-term storage to provide some redundancy. Pre-storage aggregation will allow for some pre-processing and pre-structuring of the data before it is stored in a long-term repository. This is intended to help structure and manage the data to help prevent data persistence issues, such as data swamps or data graveyards (discussed in N21). Furthermore, it is highly likely that the different stakeholders will make use of different cloud platforms and pre-storage aggregation is more suited to multi-cloud environments.

Local aggregation is suggested where possible, but the physical distribution of a smart city is likely to necessitate aggregation across networks. Therefore, the aggregation will resemble both local network aggregation and cloud-based

aggregation (refer to Section 9.3.3 for a discussion on this phenomenon). It is expected that many gateway devices (as in Smart Santander) and fog servers (such as in a hospital or traffic control centre) would be located within the city. Therefore, Wide Area Network (WAN) aggregation can be done to still allow for aggregation before the data enters a cloud platform. Furthermore, a private WAN can be used for better throughput, better security and better reliability.

Considering that much of the data acquisition in smart cities is done using WSNs, micro-batch processing and transmission is likely to be necessary. Micro-batch processing allows for data synchronisation and it is more suited to environments with intermittent network connectivity.

Some DTs in Figure 11, such as the building DTs, show multiple lines of aggregation. Some of these DTs may aggregate to a land use area, if it forms part of a land use area with a DT, and otherwise some buildings may aggregate directly to the ward. However, the same data should not be aggregated by both the land use area DT and the ward DT since this could cause data consistency issues as discussed in Section 5.1. The “Other DTs” refer to more specific DTs, such as a DT of an irrigation system in a park.

The vehicle DTs capture data about the vehicle, as well as the environment and they can contain other data, such as logs about important contents or data captured by vehicle manufacturers. The variety of stakeholders and the mobile nature of vehicles means that the vehicle DTs do not have a strong intensity of interaction with any specific DTAs. For example, a vehicle captures environment data and could be aggregated a ward DTA, but the vehicle will likely drive through multiple wards in a day. This means that the vehicle DT will constantly cause hierarchy reconfigurations and the ward DTAs will only receive data intermittently and possibly at unpredictable intervals. Therefore, even though a vehicle may be represented by a DT, its data is aggregated by the services network since it does not share a strong spatial relationship with any other DTs in the hierarchy.

The services network has also been expanded from the reference architecture presented in Chapter 6. The services in the expanded services network are classified into: 1) management services (as discussed in Section 10.2), 2) external data integrators (services that acquire data not captured within the DT hierarchy, such as citizen comments), 3) proprietary developed services (services designed by data owners that have full access to the applicable data), 4) data manipulation and processing services (service that manipulate the data to make it suitable for third-parties) and 5) third-party services (services that are designed by third-parties who do not have full access to the raw data).

For messaging within the DT hierarchy, multiple message-oriented middleware that support a binary protocol with a publish-subscribe messaging pattern is

suggested. Considering the variety of stakeholders and the size the system, having a single message-oriented middleware is not feasible. Instead, a message-orientated middleware for each group of aggregates may be more feasible and, in some cases, individual DTAs may require a dedicated message-oriented middleware. For example, some building DTAs, such as a DTA of a hospital, may require a dedicated middleware, whereas all the buildings in a land use area, such as a university, may use the same middleware.

Furthermore, reputable message-oriented middleware, such as RabbitMQ or the Mosquitto broker, have built-in security features, such as regulated data topics, and support for the TLS protocol. Making use of a message queue within a middleware, as well as making use of a publish-subscribe pattern, further accommodates the need to provide for intermittent network connectivity. Finally, protocols such as MQTT and AMQP are also binary protocols and thus allow for better interoperability. For messaging within the services network, REST APIs are commonly used along with a federated security approach, such as OAuth2.

For database queries, a proxy service should be used as opposed to allowing direct access. A proxy service for database queries can be beneficial for various reasons, including additional security, it allows for load balancing, it can make use of caching to improve response times to frequently made queries, it can be used to check queries, etc.

In terms of hosting, services that require low latencies, higher security standards and location awareness are better served on fog servers. Services that are open to the public and services that entail intensive periodic processing, such as training machine learning algorithms, are better served in the public cloud. Furthermore, different stakeholders may make use of different cloud platforms and thus multiple services networks may also be present. In such cases, the services networks of other stakeholders can be considered as external services to the services network being considered.

Furthermore, services that require stricter security measures or greater isolation can make use of VMs, but the most popular hosting option is likely to be containers. Containers' ability to dynamically allocate resources allows for more efficient resource utilisation and containers provide more support for performance and reliability monitoring. Furthermore, periodically invoked services will also benefit from containers' ability to deploy faster.

Finally, multifactor authentication can be employed for more secure services. This level of security may be beneficial for services that deal of safety and security issues. For example, a traffic congestion control service that allows for directed communications can benefit from multifactor authentication. Health related services are also services that may employ multifactor authentication.

### 12.2.6 Discussion

The smart city case study demonstrates the ability of the design framework to be applied to a larger and more abstract case. The design steps, principles and patterns can thus be applied to multiple levels of abstraction and it can be applied recursively to help further design sub-systems of the larger system. The security and compatibility design patterns were able to direct designers towards a feasible architecture and the few conflicts that arose between the design patterns were easy to resolve.

For the smart city case study, the physical system decomposition and data characterisation were a vital starting point. The smart city decomposition, both functionally and spatially, provided a top-down perspective, while the data characterisation provided a bottom-up perspective of the smart city. Taking both views into account was very useful for the needs and constraints analysis, as well as all subsequent steps (particularly the DT identification step).

Having a list of typical complexity needs (as presented in Chapter 3) helped to identify applicable needs within the smart city context. However, it is advisable to use domain related terms to rephrase the general complexity needs for the case. This helps to make the needs more specific to the case and more understandable when referred to later.

This case study also introduced mobile DTs, such vehicles (and potentially citizens), which present some noteworthy characteristics. Firstly, these mobile DTs are not spatially bound and thus do not fit well into a DT hierarchy. Therefore, mobile DTs are aggregated by the services network to prevent the need to continually reconfigure the DT hierarchy. Secondly, the mobility of the DTs means that they capture data that may be applicable to many other DTs. For example, in Smart Santander, data is captured about the vehicle, which may be applicable to the vehicle owner, as well as the manufacturer, while also capturing data about the environment throughout the city. In such cases, where a single DT captures data applicable to many other DTs and stakeholders, the need for good metadata becomes paramount. Many data management systems make use of “tags”, which are functional or object-oriented categories to help users identify what data is applicable to them.

In terms of communication, it is likely that multiple message-oriented middleware instances may be required to help manage the data load. This variety of middleware instances may cause some management issues and the effect of this still needs to be investigated. In terms of the services network, the inclusion of various service network groups may prove useful and this should also be further investigated.

# 13 Detailed case study

The purpose of this case study is to further demonstrate the systematic approach and generality of the design framework. This case study further validates the design frameworks approach by discussing how the architecture was implemented and tested. The implemented architecture's scalability and reconfigurability were tested to validate the ability of the architecture in key areas of concern.

Section 13.1 presents the architecture design, whereas Section 13.2 discusses the implemented architecture and the results of the experiments. Appendix A provides additional details about the case study and the related testing.

## 13.1 Heliostat field architecture design

### 13.1.1 Scenario

Concentrating solar power (CSP) plants are a method of utilising solar energy for power production. CSP plants generally consists of three major subsystems: a concentrating subsystem, a thermal storage subsystem and a power generation subsystem (Xiao, Xie & Deng, 2018). Heliostats are orientable high-reflectance mirrors that form part of the concentrating subsystem. Heliostats reflect solar energy onto a central receiving tower which contains a heating medium, typically molten salt, that is heated up and redirected to the thermal storage or steam generation subsystems (Cruz, Alvarez, Redondo, *et al.*, 2018).

To sufficiently heat the molten salt, a typical heliostat field can consist of tens of thousands of heliostats. For example, Malan (2014) designed a prototype heliostat field for operation in South Africa, where 10 000 heliostats were required for a 5 MW energy generation plant. Larger heliostat fields, such as the 392 MW Ivanpah heliostat field in California, can have over 300 000 software-controlled heliostats (BrightSource, 2013).

The scope of this case study is the design of a heliostat field system of DTs for a 5 MW heliostat field as presented by Malan (2014). Therefore, the design considers a heliostat field of about 10 000 heliostats. Malan, (2014) contributes to a larger research project lead by researchers from Stellenbosch University's Solar Thermal Energy Research Group (STERG). STERG uses a prototype heliostat field called Helio100 for research and development purposes. Therefore, technical information about the field is gathered from the Helio100 heliostat field.

The physical system's composition has an effect on the needs analysis and, therefore, the physical decomposition is given here:

The heliostat field can be decomposed into three tiers, namely the heliostats with their local controller units (LCUs), the cluster control units (CCUs) and the field control unit (FCU). Each heliostat consists of: a mirror, two stepper motors (that act as the actuators), an LCU, a battery, and some photovoltaic (PV) solar panels to charge the battery. The LCU is used to send control signals to the stepper motors to orientate the mirror and the LCU is the data source of the individual heliostats.

Heliostats are also arranged into groups of six according to their support structure. The six heliostats that are supported by the same structure are called a pod. The heliostats in the Helio100 field are named according to their pod number and heliostat number within that pod. For example, heliostat four in pod nine would have the ID number 9.4 (<*pod number*>.<*heliostat number*>).

The CCU is a small computer, such as a Raspberry Pi microcomputer, that calculates the motor positions for all the LCUs that are connected to it. Typically, a CCU has 24 LCUs (4 pods) or 30 LCUs (5 pods) connected to it via a short-range wireless communication protocol (radio frequency communication with serial bus for the Helio100 field). Furthermore, multiple CCUs are connected to the FCU using an ethernet connection that also provides power to the CCUs.

The FCU at the Helio100 field is a single desktop computer that is responsible for six CCUs. Furthermore, the FCU also collects data from the central receiving tower and the weather station. For a larger field, the desktop computer would likely be replaced with a dedicated server. Figure 12 depicts the heliostat field's physical decomposition, where a heliostat forms part of a pod, four or five pods communicate to a CCU and multiple CCUs are connected to the FCU.

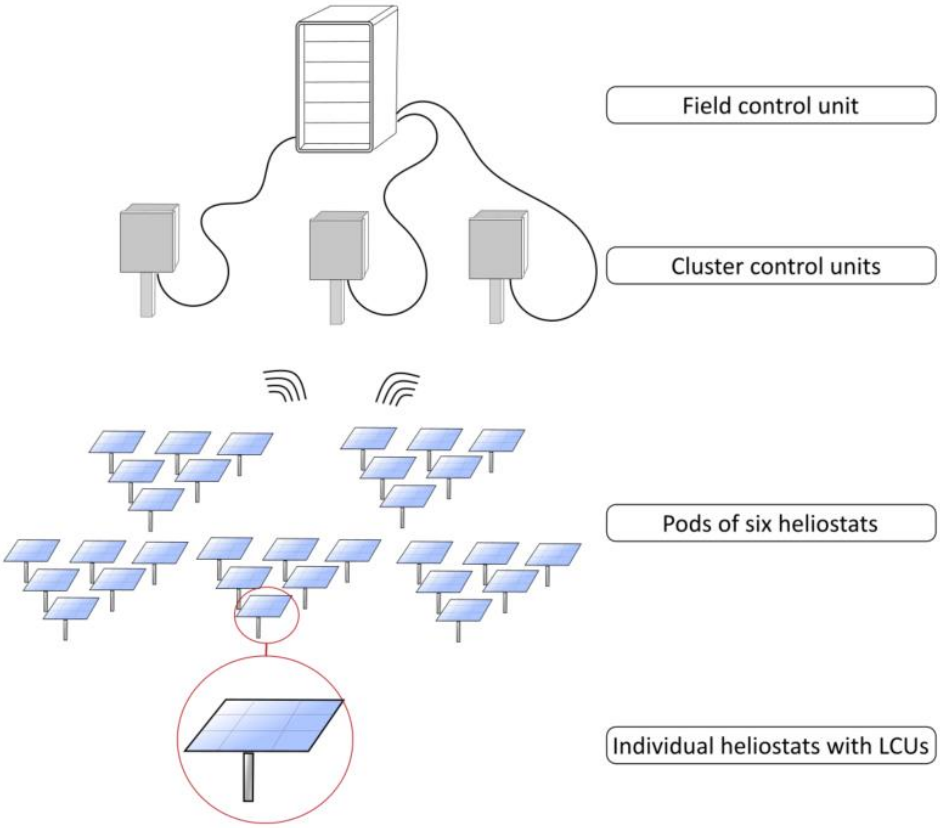



**Figure 12: Heliostat field hierarchy decomposition**

### 13.1.2 Needs and constraints analysis

STERG is the only known stakeholder of the heliostat field and they expressed the following needs:

- Heliostat fields tend to be in remote locations, such as desserts, and thus the field needs to be monitored remotely. Furthermore, the system of DTs must allow for supervisory control. For example, when the wind is too strong and may cause damage to the heliostats, the operator must be able to command the heliostats to move into their home position.
- Heliostat field performance is highly dependent on the weather and thus analytics are required to better understand how the weather effects the energy production.
- Manual inspection of the heliostats in a heliostat field is not feasible and thus physical fault detection and prognostics are needed.
- New heliostats are often added to the heliostat field during its operational lifetime, to either replace damaged heliostats or enable higher energy

production. Therefore, the system of DTs must be able to easily accommodate the loss or gain of heliostats, where newer model heliostats may be slightly different to the old models.

- One of the largest barriers to the wider adoption of heliostat fields is their high initial cost and thus cost effectiveness is a primary concern.

Based on the above, Table 26 lists the functional requirements for the heliostat field.

**Table 26: Functional requirements for the heliostat field.**

| **High-level functional requirements** | **Description** | **Group (Primary or secondary)** | **Priority** |
|---|---|---|---|
| Remote monitoring | The status of individual heliostats, as well as the status of subsections of the field need to be presented to a user. | Primary | Mandatory |
| Supervisory control | It must be possible to send commands to individual heliostats for calibration purposes and it must be possible to send the same command to subsections of the field for operational control. | Primary | Mandatory |
| Exploratory analytics | The relationship between the weather and the production capacity of the heliostat field is unknown and some insight into this relationship is required. | Primary | Highly desirable |
| Physical fault detection | Given the number of heliostats, it is not feasible to do manual inspection of each heliostat. Therefore, it must be possible to identify faulty hardware using the DT. | Primary | Mandatory |
| Prognostics | Scheduled maintenance of all the heliostats is not feasible and thus predictive maintenance is required to identify which heliostats need maintenance within a two-week timeframe (two weeks was given as the average time it takes to get new parts). | Primary | Highly desirable |

| High-level functional requirements | Description | Group (Primary or secondary) | Priority |
|---|---|---|---|
| Event logging | Considering that the heliostat field can be controlled automatically or by a user, it is considered good practice to log events related to the heliostat field control. The best practise is related to maintenance for debugging purposes and to security for accountability and non-repudiation. | Secondary | Mandatory |

The applicable NFRs, presented in Table 27, were identified using the needs tables in Chapter 3 with reference to the scenario and physical system decomposition in Section 13.1.1. For the sake of brevity, Table 27 only presents the first two needs in detail, while the subsequent needs are only listed with their implications. Table 30 in Appendix A.1 contains a more detailed description of each of the NFRs listed in Table 27.

**Table 27: Non-functional requirements for the heliostat field**

| **Need** | **Provide for large amounts of data. (Related to N24)** |
|---|---|
| **Related NFR** | Performance efficiency. |
| **Rationale for NFR** | Considering the size of the heliostat field, the amount of data generated by each heliostat and the potential resource constraints, there is a need to handle a large amount of data efficiently. Therefore, resource utilisation, scalability and high throughput are primary concerns and these are sub-characteristics of performance efficiency. |
| **NFR grouping** | Quality attribute. |
| **Implication of NFR** | Use performance efficiency design pattern. |
| **Need** | **Provide for resource constrained devices. (Related to N25)** |
| **Related NFR** | Performance efficiency, solution constraint. |
| **Rationale for NFR** | The required performance metrics must be reached with minimal resource usage to increase the longevity of the resource constrained devices. The LCUs of the individual heliostats are battery powered and the batteries are charged using photovoltaic (PV) panels. Therefore, the LCUs are energy constrained and as a result the heliostat control engineers have limited the computational responsibilities of the LCUs. The resource constraints of the CCUs and FCU are unknown. |
| **NFR grouping** | Quality attribute. |
| **Implication of NFR** | Use performance efficiency design pattern.<br>Additional code, other that the heliostat control program, may not be implemented on the LCUs. |

| | |
|---|---|
| **Need** | **Allow for system changes with minimal impact. (Related to N9)** |
| **Implication** | Use portability and maintainability design patterns. |
| **Need** | **Provide for intermittent network availability and limited network bandwidth. (Related to N26)** |
| **Implication** | The performance efficiency design pattern should be used with elements of the reliability design pattern to compensate for the intermittent network availability. |
| **Need** | **Avoid physical resource contention amongst software components. (Related to N27)** |
| **Implication** | Use the performance efficiency design pattern and incorporate elements of the compatibility design pattern related to co-existence. |
| **Need** | **Allow for retrofitting and integrate with existing information systems. (Related to N6 and N7).** |
| **Implication** | The compatibility and portability design patterns must be applied to ensure that different technologies and components can be replaced without disrupting the system. Some of the technologies related to the data acquisition part of the DT are predefined and must be integrated with, such as the use a PostgreSQL database for local storage. |
| **Need** | **Verify and validate the behaviour of DTs in response to system changes. (Related to N10)** |
| **Implication** | Use the maintainability design pattern to improve the testability of the digital twins. |
| **Need** | **Structure the data to prevent it becoming unusable. (Related to N21)** |
| **Implication** | Make use of the maintainability design pattern. |
| **Need** | **Provide a cost-effective solution. (Related to N28)** |
| **Implication** | The cost constraint will limit the amount of development time that can be spent on quality assurance and testing. |
| **Need** | **Allow for easy long-term maintenance and extension. (Related to N14)** |
| **Implication** | Use the maintainability and portability design patterns. The portability design pattern is particularly important in this case since STERG emphasised the need to adapt to changes in hardware. |

Based on Table 27 and in further consultation with the consultant at STERG it was determined that the performance efficiency is the primary quality attribute, particularly with respect to the limited resource usage and high throughput requirements. Furthermore, the portability was cited as the next most important quality attribute to allow for easy hardware changes as they need to take place.

Therefore, the architecture will primarily use the performance efficiency design pattern, supplemented with the portability design pattern. Furthermore, the

constraints will also influence some of the design choices, particularly where there are trade-offs involved.

### 13.1.3 Physical system decomposition

The physical system decomposition was done based on consultations with the engineers at the Helio100 field, a site visit to Helio100 and documentation provided by the engineers at Helio100. The decomposition is given in Section 13.1.1.

Table 28 provides the span of reality for a CCU, while the LCU and FCU are briefly discussed after the table. Table 31 in Appendix A.2 provides a detailed span of reality for each component of the heliostat field.

**Table 28: Span of reality of a heliostat field CCU.**

| Physical component | Cluster control unit (CCU) |
|---|---|
| **Physical system scope** | 24 or 30 heliostats (4 or 5 pods) |
| **Data characterisation (Data granularity) of data recorded/ generated by physical component** | Unique identifier for each heliostat (<*pod number*>.<*heliostat number*>).<br><br>Status value for each heliostat – [start-up, manual move, running, standby, home, calibration, e-stop, offline].<br><br>Translated operational commands – The commands are unknown.<br><br>Grena algorithm inputs – Fractional Universal Time (UT), date, time difference between UT and terrestrial time, longitude and latitude of heliostats.<br><br>Grena algorithm outputs – Global coordinates of the sun, local coordinates of the sun. |
| **Data characterisation (Data granularity) of data sent to physical component** | Operation commands (from FCU).<br><br>Grena algorithm inputs (from FCU) – Air pressure, ambient temperature.<br><br>Individual heliostat parameters (from every LCU in the CCU's scope) – Battery value, stepper motor positions, timestamp. |
| **Data format** | JSON formatted message. |
| **Communication** | ZeroMQ messaging over TCP/IP and WLAN/ethernet. |

| | |
|---|---|
| **Considerations and Constraints (Capacity for interaction)** | Available processing and storage capacity of CCUs is unknown.<br>Assuming each CCU controls 4 pods, 417 CCUs will be required. Assuming each CCU controls 5 pods, 334 CCUs will be required. |

The LCU collects data from the battery and the two stepper motors at one-minute intervals. The FCU collects operational data from the central receiving tower, such as the temperature of the molten salt and the operational data of the various pumps and valves. The weather station's data is also sent to the FCU, as well as the calibration images taken using an IP camera.

### 13.1.4 Services allocation

**Services identification and characterisation**

According to the service patterns described Section 8.1, the requirements can be captured by the following service patterns: mirror, life counter and root cause. Mirror captures the requirements for remote monitoring and supervisory control while the life counter relates to the requirement for prognostics and the physical fault detection requirement relates to the root cause service.

The exploratory data analysis does not neatly fit into any of the proposed service patterns in Table 15 and thus it will simply be referred to as exploratory data analytics. Therefore, there are four services to capture the five functional requirements. The mirror service is discussed in detail below, whereas the other services are only briefly discussed.

**Mirror service**

Description: The status of individual heliostats, as well as the status of subsections of the field need to be presented to a user. It must be possible to send commands to individual heliostats for calibration purposes and it must be possible to send the same command to subsections of the field for operational control. The actual control logic is performed by the CCUs and is outside the scope of the service, but the CCU must be sent the right commands based on user inputs.

Related primary functional requirements: Remote monitoring and supervisory control.

Related secondary functional requirements**:** Log files of events and log files of operator details and issued commands.

Required physical scope**:** Individual heliostats – Some control operations are applicable to single heliostats, e.g. during calibration, single heliostats may need to be adjusted.

Heliostat clusters – Heliostats related to a CCU may need to be configured differently since the heliostat technology may differ.

Field overview – An overview of the operational state of the heliostat field is required for general operational checking and full field commands, such as "move-to-home", are required when the wind becomes too strong and may cause damage.

Required data granularity:

Individual heliostat scale

- Data features: LCU level - Motor position values, battery values. CCU level – all Grena algorithm values, heliostat status values. FCU level - calibration images.
- Timescale: All data features, other than the calibration images, should be measured or calculated at one-minute intervals. The calibration images are only taken during heliostat calibration and 3 to 5 photos are taken per second.

Heliostat clusters scale

- Data features: CCU level - heliostat status values.
- Timescale: Status levels should be determined once per minute.

Full field scale

- Data features: FCU level - Cluster status summaries, weather data.
- Timescale: Cluster status summaries need to be updated every minute. Weather data is also measured once per minute.

Service characteristics:

- Required data update frequency: Real-time (at all levels).
- Degree of user interaction: Remote monitoring – periodic user interaction. Supervisory control – user driven.
- Intensity of interaction: Spatially focussed service.
- Persistence: Persistent data gathering and possibly model validation. Periodic user interaction.

Constraints and considerations: Remote monitoring requires access from an external network and thus the service must either be cloud hosted or it must allow for direct local network access, such as through a VPN or SSH connection. The local storage capacity is likely to become a constraint. The data throughput may become a bottleneck in a sufficiently large field.

The other services that are considered within the heliostat field case study are:

- Life counter service: This is a prognostics service that makes use of historical data from a heliostat to determine when maintenance is required. The consultants at STERG expressed the need to be notified at least two weeks before a heliostat failure. This service would most likely make use of trend analytics and possibly machine learning.
- Root cause service: This service is intended to identify physical system faults or failures. For example, the consultants at STERG mentioned that when PV panels or batteries get old, they cannot provide enough power anymore. In such cases, the heliostat's movement becomes sluggish or the heliostat becomes inactive. The root cause service must be able to identify such faults.
- Exploratory analytics service: This service is intended to allow the consultants at STERG to investigate trends in power production, such as the influence of the weather. Therefore, large amounts of cleaned historical data are required along with adequate processing ability.

**Digital twin identification and characterisation**

Based on the physical system decomposition, it is possible to build DTs of the CCUs and the FCU. The LCUs of individual heliostats are resource constrained and thus the control engineers have restricted the responsibilities of the LCUs to the heliostat control only. However, the CCUs do have all the LCU data of each of its associated heliostats. Therefore, DTIs can be built to represent CCUs and the FCU would be represented by a DTA.

**Service to digital twin allocation**

Table 29 provides a list of DTs that have the span of reality required by the various services.

**Table 29: Potential service allocation, for a heliostat field, based on span of reality**

| Service | CCU DTI | FCU DTA | Service Network |
|---|---|---|---|
| Mirror | | X | X |
| Root cause | X | X | X |
| Life counter | X | X | X |
| Exploratory analytics | | X | X |

The initial service allocation is according to the lowest level DT that has the correct span of reality. Based on this, the root cause and life counter should be allocated to the individual DTIs, while the mirror service should be allocated to the DTA and the exploratory analytics should be allocated to the service network. The exploratory analytics service has been delegated to the services network because

it is a more generally defined, functionally focussed service that is periodically invoked.

In terms of service separation, the root cause and life counter are separated. The life counter service is hosted in the DTI's cloud platform and the root cause is hosted in the local portion of the DTI. This is because the root cause service has a high reliability requirement since its primary function, in this case, is to identify system faults (in near real-time) and notify the appropriate stakeholders accordingly. The life counter service, on the other hand, is more of an analytics service and thus computational performance and adaptability is likely to be more important.

### 13.1.5 Design pattern selection and application

The primary quality attributes related to the system are performance efficiency and portability as identified in Section 13.1.2. Therefore, the performance efficiency design pattern will be applied to the mirror service, while the portability design pattern will be applied to the exploratory analytics and life counter services. However, the root cause service will be designed according to the reliability design pattern.

The internal architectural design of the DTIs, DTAs and the service network is given in Figure 13. The internal architectural design of the DTs is done according to SLADTA as discussed in Section 2.3. The internal design also shows the services allocation. The life counter service is hosted in the cloud portion of the DTI, to allow access to large amounts of stored data, as well as computational resources. The root cause service is hosted in the DTI's local portion for better reliability. The services network includes the exploratory analytics services, as well as a directory to improve the adaptability of the design. To further improve their portability, the services should be hosted in containers.

It should be noted that the mirror service should ideally be hosted within the local portion of the DTA, where the local network's benefits (higher throughput and reliable connection) can be exploited. However, the requirement for remote monitoring limits this. Therefore, the mirror service has been separated into the mirror service dashboard (hosted in the cloud portion of the DTA) and the mirror service computation (hosted in the local network portion of the DTA). This allows for aggregation within the local network, while only sending the necessary aggregated data to the mirror service dashboard.

Regarding the DT's internal design, PostgreSQL databases are specified for each DTs' short-term local data repository, because the heliostat field engineers are familiar with PostgreSQL. Layer 4 consists of a publish-subscribe client, which relates to the aggregation and communication architecture discussed later in this section. The long-term data storage on Layer 5 is intended for data analytics and

for each DT a NoSQL database is recommended, as per the performance efficiency and portability design patterns. NoSQL databases tend to have better scalability and latency performance, while also allowing for more adaptability because they do not enforce a strict schema.

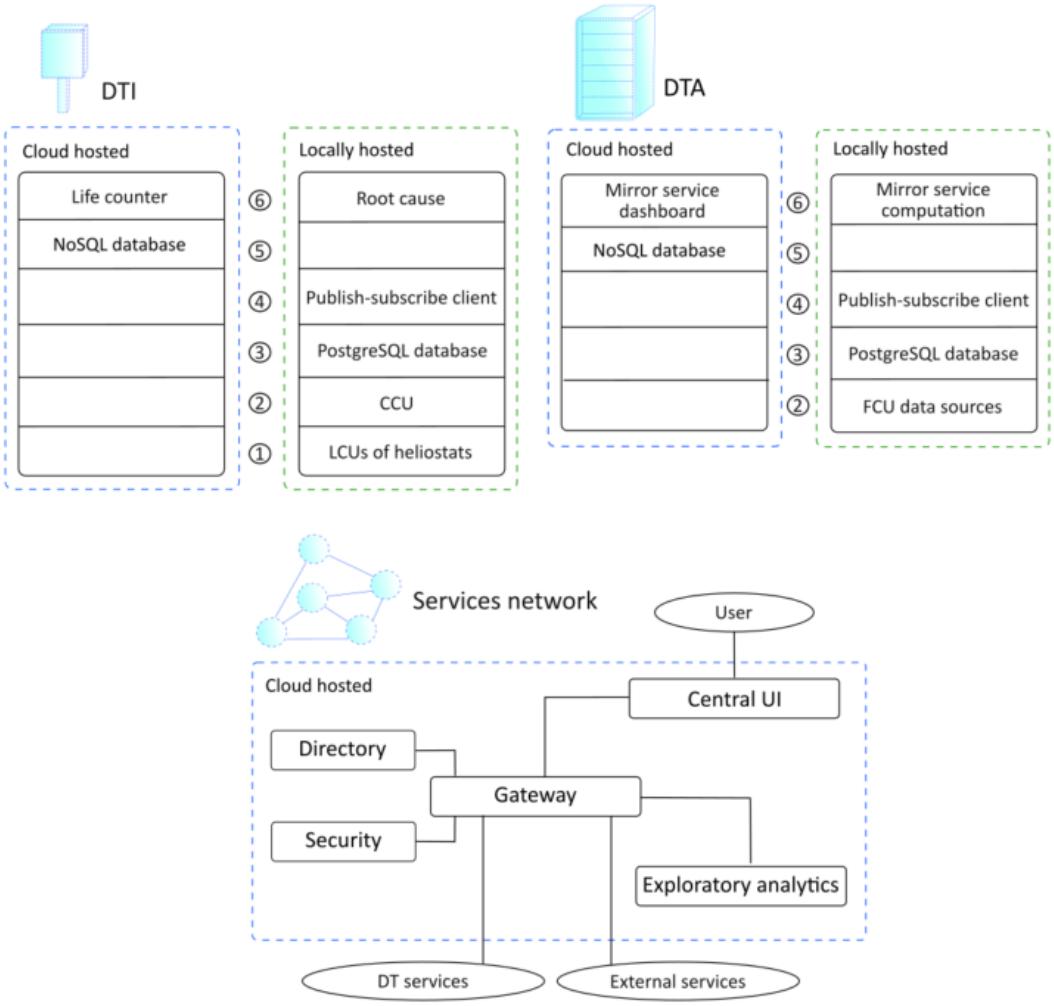



**Figure 13: Internal design of the DTI (top left), DTA (top right) and service network (bottom)**

The aggregation hierarchy design is given in Figure 14. The broker, which is a locally hosted, lightweight broker, facilitates publish-subscribe communication, as recommended by the performance efficiency and portability design patterns. Publish-subscribe communication is scalable and it is well-suited to many-to-one communication as is the case for the heliostat field. Publish-subscribe also decouples the DTs, which is beneficial for reliability and replaceability.

The aggregation within the DT hierarchy, i.e. between the CCU DTIs and the FCU DTA, is pre-storage, local aggregation performed through stream processing. These are all recommendations made by the performance efficiency design pattern. Pre-storage aggregation allows for reduced data storage and lower latencies, while local aggregation further allows for lower latencies and higher throughput. Stream processing further lowers latencies.

Communication and aggregation through the gateway are intended for event-driven, batch data or service requests. The exploratory analytics service can use the gateway to request data from DTIs or the DTA and combine it with external data where necessary.

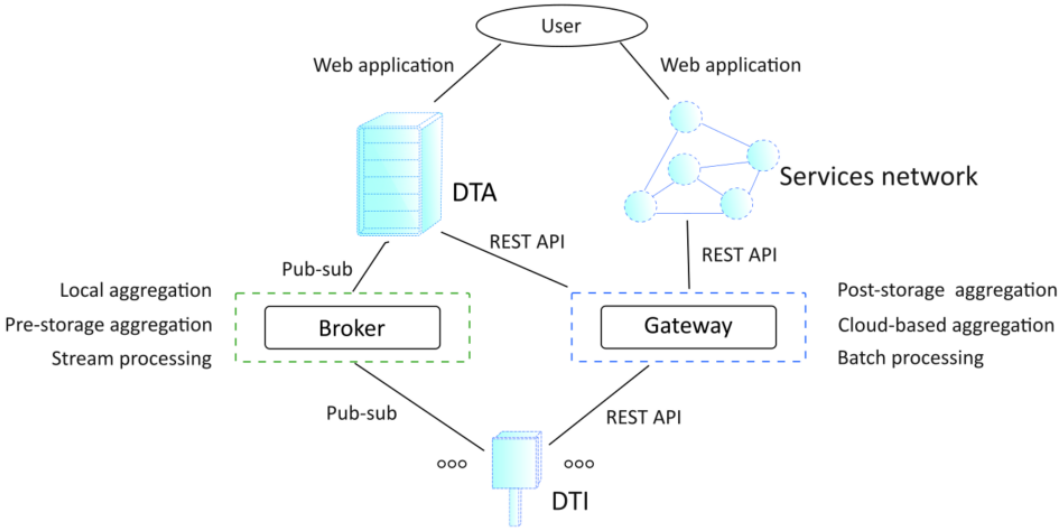



**Figure 14: Aggregation hierarchy design**

## 13.2 Heliostat field implementation

### 13.2.1 Implementation scope

The implementation of the case study is intended to contribute to validating the design framework and design patterns by testing the architecture that was developed. However, a full implementation of the architecture is not feasible in the scope of this dissertation and instead a proof-of-concept implementation is presented here. The proof of concept is aimed at assessing the scalability and portability of the architecture to contribute to validating the performance efficiency and portability design patterns, respectively. Furthermore, the architecture will be validated against the complexity needs and NFRs in Table 27.

For the proof of concept, the implementation of the heliostat field system of DTs only considers the DT hierarchy and not a services network. Furthermore, the scope of the DT hierarchy is limited to a rudimentary mirror service which simply displays the relevant data. The mirror service is a suitable subject for the assessment since it requires good scalability. To assess the portability of the architecture, some reconfiguration scenarios are investigated.

Furthermore, the proof of concept focusses on assessing the scalability and portability enabled by the aggregation hierarchy. For this case, pre-storage aggregation is performed and thus the experiments focus on assessing the scalability and portability of the DTs hierarchy's internal communication, which relates to Layer 4 of SLADTA.

### 13.2.2 Physical architecture

The internal physical architectures for the DTIs and DTAs are presented in Figure 15. Appendix A.3 presents a more detailed description of the physical architecture and the aggregation communication.

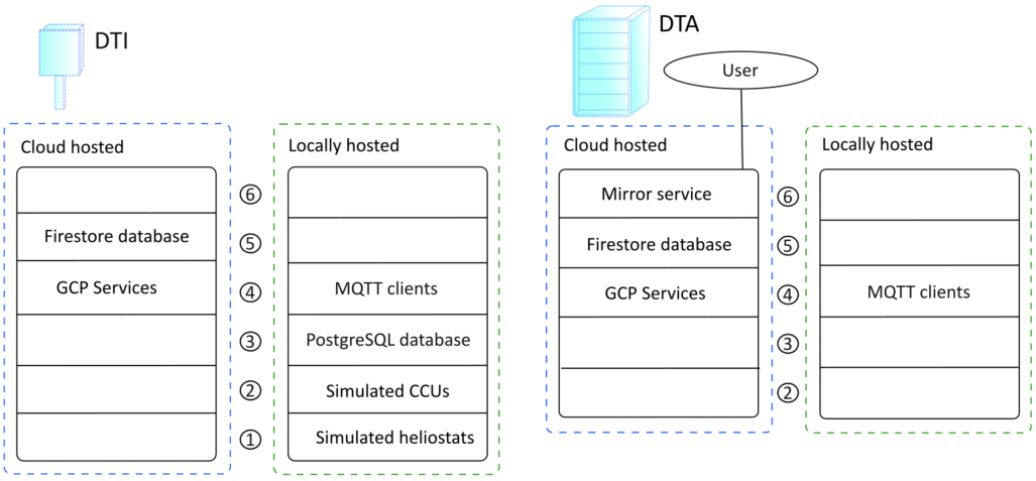



**Figure 15: Internal physical architectures for heliostat field DTIs and DTAs**

The heliostats and CCUs (Layer 1 and 2, respectively) have been simulated to allow for more flexibility to change experimental parameters, while also allowing for a scale of data that is not feasible in a laboratory environment. The short-term, local PostgreSQL database is used because the heliostat field control engineers are familiar with PostgreSQL. The long-term, cloud-based Firestore database (which is a NoSQL database) is a design choice according to the performance efficiency design pattern.

The DTIs' and DTAs' Layer 4 are custom developed Python programs, as well as some Google Cloud Platform (GCP) services. This Layer contains two MQTT clients per DT: one to send data to the cloud and another for aggregation. In the cloud, the GCP IoT Core, Pub/Sub and Cloud Functions services were used to receive data from the local Layer 4 and write the data to the appropriate data repository.

Furthermore, three aggregation scenarios are considered in this case study. For the three cases, the message-oriented middleware that facilitates aggregation is: 1) a locally hosted Mosquitto broker, 2) a cloud-based Mosquitto broker and 3) a GCP broker. The local Mosquitto broker performs pre-storage, local aggregation using MQTT, which is a lightweight publish-subscribe protocol. The cloud-based Mosquitto broker (hosted in a cloud-based GCP VM) performs pre-storage, WAN aggregation using MQTT. The cloud-hosted Mosquitto broker represents WAN aggregation because it only makes use of a VM hosted within the cloud and does not make use of any cloud specific services. Therefore, the data and the data pipeline components remain fully in the control of the developer.

The GCP broker refers to GCP's Pub/Sub service, which makes use of HTTP and a message queue to perform publish-subscribe communication. The Pub/Sub service performs pre-storage, cloud-based aggregation. The Pub/Sub service represents a typical cloud offered solution that could be used as a broker for aggregation. Furthermore, all the brokers facilitate stream processing.

### 13.2.3 Scalability experiments

The purpose of the scalability experiments is to contribute to validating the design framework by testing the scalability of the architecture discussed in the previous section. The scalability of the architecture is a key concern in this case study and thus these experiments aim to determine whether this concern has been satisfied.

The scalability of the architecture was tested by performing various experiments using the various brokers described in the previous section. In particular, the scalability of each broker scenario was measured for varying message frequencies (discussed in Appendix A.4), varying message sizes (discussed in Appendix A.5), as well as through partitioning of the aggregate and/or broker (discussed in Appendix A.6). The scalability experiments were designed to stress the system of DTs until failure or until significant performance degradation occurs.

#### 13.2.3.1 Procedure and measurement method

To quantify the scalability (as defined in Section 4.3), the broker being tested, as well as a DTA, were started before periodically adding DTIs (one DTI was added per minute) to gradually increase the data throughput that needed to be sustained. The round-trip latency was then monitored to determine when the system became unstable. The round-trip latency is the collective time it takes a

message to 1) travel from DTI to broker, 2) travel from broker to DTA, 3) be processed by the DTA, 4) travel from DTA to broker and 5) travel from broker to original DTI. Round-trip latency is a standard method of measuring latency because the start and end timestamps are generated by the same machine to ensure accurate measurements.

To better understand the experimental method and the variables, please refer to Figure 16, which is an example of a typical scalability experiment output graph. In this graph, the coloured vertical lines indicate where a DTI is initialised and the jagged graph thereafter (in the same colour) is the time a message was received (x value) plotted against the round-trip latency for that message (y value).

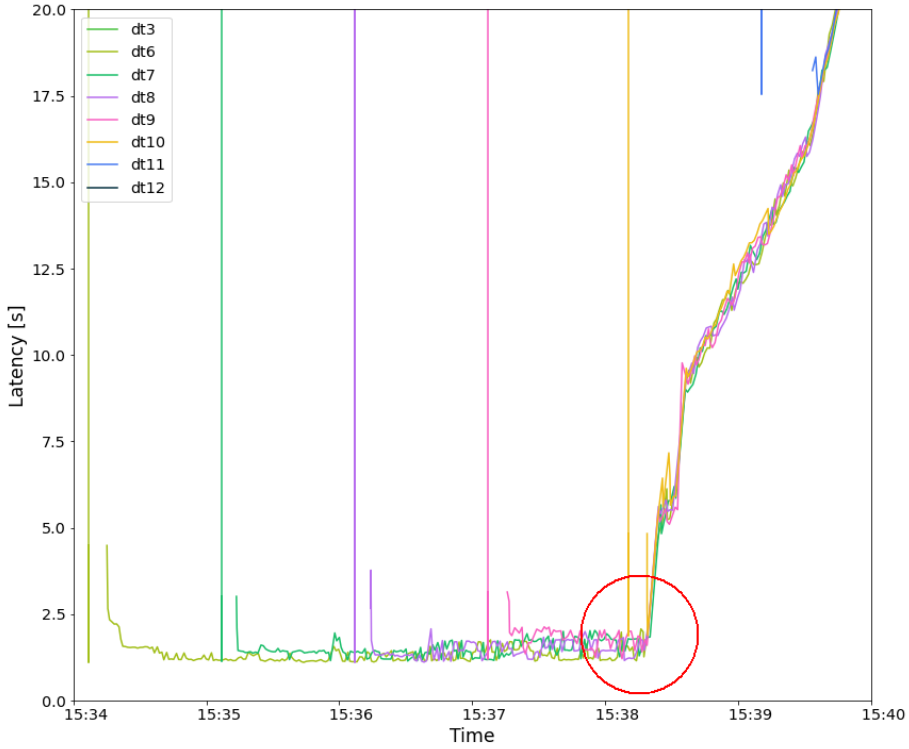



**Figure 16: Limit point of the DT aggregation.**

Some terms that will be used to explain the behaviours of the data pipeline are:

- The *limit point* is defined as the point where the latency starts increasing rapidly and probably unbounded (circled in red in Figure 16). This was a common occurrence for all the scalability experiments.
- The *threshold period* is defined as the minute before the limit point is reached and it is one minute long because there is a one-minute interval between the start of successive DTIs. This threshold period is the period during which the

system of DTs was able to sustain the largest throughput, and thus indicates the data pipeline's best performance while stable. The one-minute delay period between DTI starts was a choice that allowed the data pipeline to reach steady-state before increasing the load further.

#### 13.2.3.2 Discussion of results

Figure 17 presents the results of the message frequency experiments for each of the broker scenarios (discussed in Section 13.2.2), given a single broker and a single DTA configuration. The vertical bars in the figures indicate the range of the values observed in repeated experiments, while the lines pass through the mean values. The latency and percentage message loss results, as well as all the results for the varying message size experiments are available in Appendices A.4 and A.5, respectively.

Figure 17 presents the *number of DTIs* that could be sustained at different message frequencies, as well as the *collective message threshold* at different message frequencies. The number of DTIs refers to the maximum number of DTIs that could be sustained (measured within the threshold period) and the collective message threshold refers to the number of messages that were processed by the DTA per second (within the threshold period). The message frequency refers to the frequency at which DTIs send their messages and it was controlled by adjusting the *sleep time* of the DTIs. The sleep time is the time in between logic execution cycles of a DTI, where a logic execution cycle refers to the DTI reading data, processing data and sending the data as a message (Appendix A.3 describes the modules that relate to these steps).

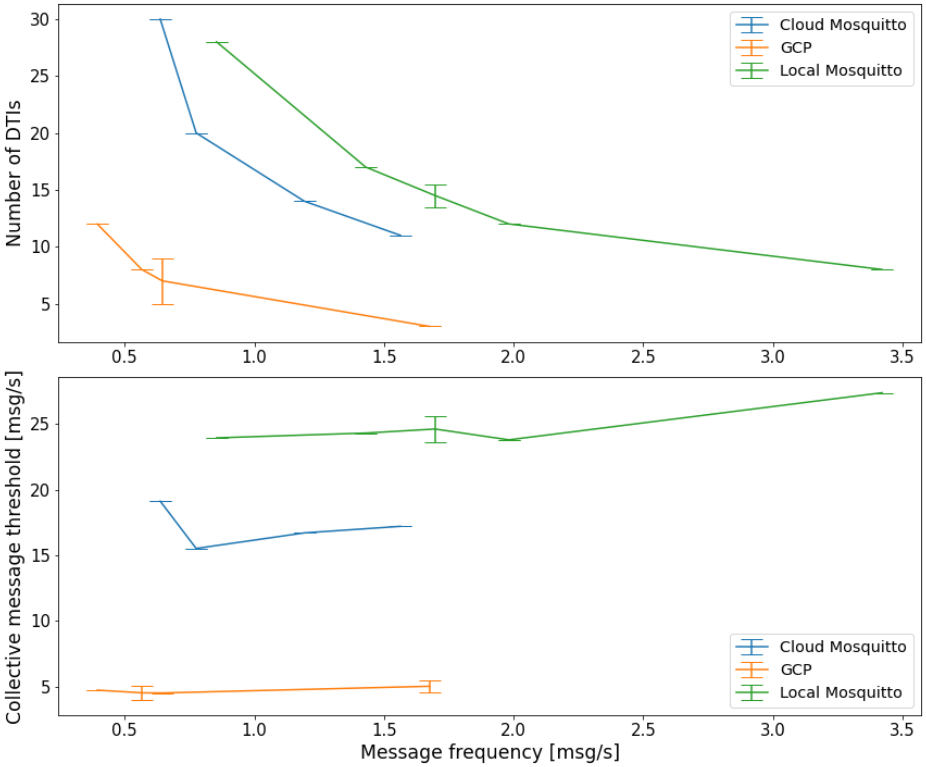



**Figure 17: Number of DTIs and the collective message threshold for different message frequencies.**

The results of these experiments indicate that message frequency of the DTIs have a significant effect on the number of DTIs that can connect to the broker and DTA, while the collective message threshold remains relatively constant. This is significant because it indicates that throughput (represented by collective message frequency) is relatively constant and thus to accommodate more DTIs, the message frequency should be reduced. It also provides a basis to extrapolate how many DTI could connect to a DTA and thus how many heliostats could be represented.

The results also indicate that message size is not as significant as message frequency when considering the scalability. However, at sufficiently large message sizes (20 kB or more for this case study) the message size does become increasingly more important to consider. The message size experiments also emphasised the need to make provision for poor network connectivity and MQTT was ideal for this. With a QoS level of 1 (QoS = 1 is supposed to guarantee that each message is received at least once), the message loss was very low, even near maximum DTA capacity. Therefore, in this architecture, MQTT and the Mosquitto broker (which

was the message-oriented middleware) allowed the architecture to satisfy N26 in Table 27 (the need to provide for intermittent network connectivity).

Extrapolating the results (as presented in Appendix A.7), it was determined that the locally hosted Mosquitto broker could sustain 460 DTIs (13800 heliostat), the cloud-hosted Mosquitto broker could sustain 310 DTIs (9300 heliostats) and the Pub/Sub service could sustain 80 DTIs (2400 heliostats). The limiting factor was the DTA's processing ability when using the Mosquitto brokers or cloud throttling when using the Pub/Sub service. The network bandwidth also became a limiting factor when large messages were being communicated. Based on the extrapolated results, the locally hosted Mosquitto broker and single DTA configuration could sustain the data load for the proposed 5 MW heliostat field. Therefore, for this case study, the architecture satisfies the need to sustain a large amount of data (N24 in Table 27).

To further demonstrate the scalability of the architecture, the DTA and/or broker were/was partitioned. In particular, four cases were investigated: 1) a local and a cloud-based Mosquitto broker with a single DTA, 2) a local and a cloud-based Mosquitto broker with a DTA dedicated to each broker, 3) a single cloud-based Mosquitto broker with two DTAs and 4) the Pub/Sub service with two DTAs. The results are presented in Table 32 in Appendix A.6.

The results demonstrate the scalability of the architecture, where an additional DTA resulted in a 50% increase in message throughput in comparison to the best single broker, single DTA equivalent (36 messages per second in comparison to 24 messages per second). In comparison to the single cloud-hosted Mosquitto broker with a single DTA, the partitioning resulted in a 125% increase in message throughput. Similarly, the Pub/Sub service achieved a 78% increase in throughput with an additional DTA. It should be noted that these results were achieved through static partitioning, where half the field as assigned to one DTA and the other half to the other DTA. If dynamic partitioning were used or if the load was distributed according to each host machine's ability, the result may be better.

Based on the results discussed above, the architecture was able to scale to the demand required for the heliostat field. Therefore, in terms of scalability, the design framework and performance efficiency design pattern were able to guide the design of a feasible architecture.

### 13.2.4 Reconfigurability experiments

The purpose of the reconfigurability experiments is to contribute to validating the design framework by testing the reconfigurability of the architecture discussed in Section 13.2.2. Therefore, this section discusses the effort required to perform certain reconfigurations on the system of DTs and thus it serves as a qualitative evaluation of the portability and maintainability of the architecture. Three

reconfiguration scenarios were investigated: adding or removing a DTI, adding or removing a DTA and adding or removing a broker. The reconfigurations required for each scenario are discussed in Appendix A.8.

The modularity of the Layer 4 design (as discussed in Appendix A.3) means that many of the software modules (such as the communication modules) of the DTIs and DTAs are the same (differences in communication are accommodated through the configuration file). Therefore, the DTIs' and DTAs' reconfigurations are also similar (as discussed in Appendix A.8). Adding a new DT (DTI or DTA) into the system of DTs requires changes to three components: the DTI's Layer 4, the DTA's Layer 4 and the cloud platform. All these changes are configurations file changes and thus no source code changes have to be made. The most time and effort were spent on configuring and generating the security credentials, such as assigning new security certificates and encryption keys to the DTs.

In terms of failures, if a DTI fails, it causes a notification within the cloud platform, but it does not influence the performance of any of the other DTIs and the DTA will simply stop updating the data profile of the DTI that failed. If a DTA fails, it causes all aggregation related to that DTA to cease, but the DTIs still communicate their data to their respective long-term repositories within the cloud. As soon as the DTA recovers and reconnects to the broker, the aggregation automatically continues.

Furthermore, a Mosquitto broker can also be added to the system of DTs by specifying the appropriate parameters in the configuration file. Similarly, the Pub/Sub service also only requires a configuration change to accommodate another DTA. If either broker fails, the aggregation ceases, but the broker can be automatically restarted using batch script. Broker failure also required the DTs to be restarted before they could reconnect to the broker. However, the DTs could implement an exponential backoff reconnection approach to remedy this problem, but this would have to be verified.

The results of the reconfigurability experiments contributed to demonstrating the portability of the architecture, while also demonstrating how horizontal and z-axis scalability are achieved within the architecture. Furthermore, these experiments demonstrated how reoccurring patterns and the concept of self-similarity within aggregation hierarchies (as discussed in Section 5.2.4) can be exploited to create modular and reusable software.

The observations made about the architecture through the reconfigurability experiments demonstrate the architecture's ability to allow for system changes with minimal impact (N9 in Table 27). The portability of the architecture also allows for easier long-term system maintenance (N14 in Table 27), while making use of the cloud to host proprietary technology (such as the cloud-based

Mosquitto broker) can further ease maintenance. These qualities also contribute to the need to allow for differing levels of technological maturity N6 in Table 27. Therefore, in terms of portability, the design framework and portability design patterns were able guide the design of a feasible architecture.

### 13.2.5 Heliostat field architecture evaluation

The architecture that was implemented for the experiments was able to address scalability needs, N24 and N26, from Table 27 (as discussed in Section 13.2.3), as well as the portability needs, N6, N9 and N14, from Table 27 (as discussed in Section 13.2.4). Furthermore, by comparing the broker scenarios, it can be determined that the locally-hosted Mosquitto broker induces the least amount of strain of the DTs and it induces the least variation is strain. Therefore, it is the best option to satisfy the resource constraints (N25 in Table 27), as well as the resource contention (N27 in Table 27). Making use of the cloud platform, where it is appropriate to do so, also contributes to alleviating the resource constraints of the heliostat field.

Furthermore, the needs listed in Table 27 that were not directly addressed using the architecture are: the need to verify and validate DT behaviour (N10), the need to structure data (N21) and the need for a cost-effective design (N28). For this case study, the DT behaviour was validated using system and program logs, as well as through the services provided by the cloud to monitor program execution. DT behaviour could be further validated through unit testing.

The data was structured according to heliostat, where each heliostat had its own subcollection in the Firestore database. This data structuring was made easier and more scalable using the GCP Cloud Functions. The cloud functions also allow the local portion of the DT to delegate processing responsibilities to the cloud where appropriate. Finally, some provisions were made to reduce costs, such as using the cloud to reduce the upfront cost of computational hardware.

Based on the results from the experiments, the design framework, along with the performance efficiency and portability design patterns, were able to guide the design of a feasible architecture for the system of DTs to represent the 5 MW heliostat field. The architecture managed to address most of the needs that were identified and the needs that were not addressed directly were still provided for to some degree.

# 14 Design framework evaluation

The general evaluation, presented in the next subsection, discusses overall aspects of the design framework, such as the systematic approach, generality, traceability and facilitation of communication. The design step evaluation, presented in Section 14.2, considers the individual design steps.

## 14.1 General evaluation

The design framework provides a systematic design approach. The sequence of design steps, proposed by the design framework, follow well on each other and, for all the case studies, very little iteration was required between consecutive design steps. The most iteration occurred within the needs and constraints analysis, where needs and constraints were updated and added as the design process continued. In particular, the quality attributes and constraints were closely linked to the physical system decomposition, as expected. However, for the WDS case study, the uncertainty about the services allocation may cause some iteration when implementing the architecture.

Furthermore, the systematic approach helps developers identify system derived needs, such as the complexity needs listed in Chapter 3. These derived needs can then be related to NFRs to determine their influence on the design choices. In particular, the applicable quality attributes associated with a derived need can be used to choose and apply design patterns that help a developer design software with satisfactory quality. The heliostat field case study contributed to demonstrating the feasibility of this quality focussed approach by applying the performance efficiency and portability design patterns and testing the resulting architecture. This systematic and quality focussed approach to software design should enable better software design in a shorter period of time.

The variety of case studies also demonstrated the generality of the design framework and the accompanying design patterns. The heliostat field presented a system with many similar devices, where scalability and portability were important. The WDS presented a continuous network system that required high reliability and maintainability, whereas the smart city presented a large scale heterogenous system where interoperability and security were important. These cases also contain different types of physical entities, for example discrete heliostats, a continuous pipeline and mobile vehicles in, respectively, the heliostat field, WDS and smart city. The level of abstraction and detail also differed amongst the case studies, demonstrating the recursive nature of the design framework, where the framework can be applied to a system, as well as its individual subsystems.

In terms of traceability, having short descriptive paragraphs, such as the rationale, and implication (see Table 20, Table 24 and Table 27) in the needs and constraints analysis proved highly useful. Similarly, the constraints and considerations section (see Table 21, Table 25 and Table 28) for the services characterisation was also very useful. However, the traceability can be obscured when constraints prevent certain architectural choices and it is suggested that a method for assessing need conflicts during the needs and constraints analysis be further investigated.

Furthermore, the span of reality definitions for both the services and the physical components, with their DTs, were helpful to map the services to DTs. Based on the span of reality, the potential service allocation tables (Table 22 for the WDS and Table 29 for the heliostat field) could be setup, which makes initial service allocation easy and it provides a reference for alternative hosting positions should the need arise.

During meetings and interviews with other researchers in the research group, the design framework allowed for targeted discussion. For example, the complexity needs in Chapter 3 provided a reference to establish which needs were applicable to the WDS, when consulting a fellow researcher on the topic. The physical system decomposition step aided in developing a system-focussed top-down perspective of the WDS, as well as bottom-up data-focussed perspective. The service patterns (provided in Table 15) also helped to stimulate discussion about the possibilities of the system of DTs in the WDS. Therefore, with regards to needs related communication, such as an interview with a client, the design framework provides examples and considerations for stakeholders to discuss. With regards to design related communication, the common requirements taxonomy, physical system decomposition and span of reality characterisation, provide a common problem space to consider.

In terms of research about the complexities and architecture implementation, the service allocation provides each DT with a clear purpose to focus on when designing the architecture and when choosing implementation technologies. The needs and constraints analysis helps focus research on key areas of concern and complexity, while the design patterns guide research in key areas of interest, such as an appropriate communications technology.

## 14.2 Design step evaluation

### 14.2.1 Needs and constraints analysis

The needs and constraints analysis tables are useful to refer to while designing the architecture and while making implementation choices. The rationale and implication sections are particularly so. The rationale helps understand the reason for the requirement which is useful to know when making trade-off decisions. For

example, knowing whether the performance efficiency design pattern is being applied to improve scalability, latency or to reduce resource usage has an effect on where compromises can be made when conflicts arise with other design patterns. The implication section is useful when determining if all the needs and constraints have been met.

The rationale that relates needs to a requirement can also be used to group similar needs. This is useful when communicating the needs. Furthermore, using the general complexity needs listed in Chapter 3 can help identify relevant needs, but it is useful to use domain related language to express the need.

### 14.2.2 Physical system decomposition

The physical system decomposition helps the designer(s) understand the context and composition of the system, as well as the external dependencies. The span of reality characterisation helps identify data dependencies and thus component dependencies within the system. Having a span of reality table (such as Table 21, Table 25 and Table 28) to refer to for information about the data characteristics, communication details and data formats, is helpful during service identification, as well as during implementation.

Together, the physical system decomposition and span of reality characterisation help a designer make more informed decisions because the decomposition provides a top-down perspective of the system while the span of reality characterisation provides a bottom-up perspective. The smart city case study was a good example of this where the city was decomposed both functionally and spatially, while the existing IoT infrastructure was used to identify what data is captured, how it is captured and how it can be used. The top-down and bottom-up perspectives also help identify many of the system derived complexity needs that form part of the needs and constraints analysis.

It should be noted that the physical system decomposition is not always simple. In the WDS case study the division of the continuous network presented a challenge. The network was divided into arbitrary sections (as is the common practice in the industry) but choosing where to make these divisions can be confusing. Some principles do make the divisions easier, such as dividing where there are minimal interactions between sections and dividing where boundary nodes are known, but such information may not always be available. It was found that proposing many decompositions and then narrowing down the options during DT selection makes the decomposition and DT selection easier (because alternatives are known and because data and services should also be known by then).

Furthermore, from the heliostat field case study, it was found that the span of reality characterisation can be long and very detailed, making it harder to use for quick references. Therefore, it may be appropriate to have a secondary document

that contains the full characterisation while the tables in the span of reality characterisation section of the design framework only contain summaries. The data characterisation can also be cumbersome to setup, but it is very useful to refer to when designing the services and architecture and during implementation.

### 14.2.3 Services allocation

When identifying applicable services, it is very useful to have a list of potential service, such as the service patterns in Table 15, to refer to. This helps to simulate stakeholders' ideas about the possibilities of the system of DTs and it helps developers focus their design efforts toward a common goal. Furthermore, the characterisation of the services was particularly useful in the WDS case study, where a service hierarchy was identified. Some services were dependent on inputs form other services (such as the benchmarking service receiving data from the virtual sensor service) and this influenced the allocation of the services. Services at a higher level in the service hierarchy were subsequently located at a higher level in the DT hierarchy.

Identifying DTs is relatively easy when the physical system decomposition has already been done. However, it is important to carefully consider the scope of the DTIs because they affect the data available to the DTAs and they perform much of the homogenisation of the data and protocols. If the DTI scope is too small, the DTI may be too simple to host any services, whereas if the DTI scope is too large, the DTI may experience poor performance or it may become difficult to maintain. Deciding when a DTI scope it too large can be difficult and the problem is exacerbated for continuous network cases, such as the WDS. The interconnectedness of the continuous network also raises the concern of how boundary values should be shared amongst the DTs without causing data inconsistencies. Furthermore, deciding on the scope of a DTI is also harder when the data characteristics and services are not yet known or if they are very vague.

The smart city case study also introduced mobile DTs, such vehicles (and potentially citizens), which present some noteworthy characteristics. Firstly, these mobile DTs are not spatially bound thus do not fit well into a DT hierarchy. Therefore, the mobile DTs were aggregated by the services network to prevent the need to continually reconfigure the DT hierarchy. Secondly, the mobility of the DTs means that they capture data that may be applicable to many other DTs and stakeholders. For example, vehicle DTs capture data about the vehicle that may be applicable to the vehicle owner, as well as the manufacturer, while also capturing data about the environment throughout the city. In such cases where a single DT captures data applicable to many other DTs and stakeholders, the need for good metadata becomes paramount. Many data management systems make use of "tags" which are functional or object-oriented categories to help users identify what data is applicable to them.

For the service allocation, the potential service allocation tables (Table 22 and Table 29) were particularly useful, not only to make initial service allocations but also to be aware of alternative hosting positions. The potential services allocation table was informed by the span of reality of the services and DTs and the span of reality seemed to work well for this purpose. However, the span of reality requirements of some services is hard to determine if the details about the services, such as data requirements, are not known.

### 14.2.4 Design pattern selection and application

The design patterns help narrow down architectural and implementation choices to allow for more focussed research and more rapid development. The services allocation is useful when identifying which design pattern(s) to apply because they give the DTs a specific purpose. Different design patterns can also be applied to different subsystems so that the most appropriate quality attribute(s) are emphasised for individual DTs and services. For each of the case studies, the applicable design patterns were able to produce a feasible architecture. The heliostat field case study helped demonstrate the feasibility of the architecture and it helped to validate the design choices recommended by the performance efficiency and portability design patterns.

More than one design pattern can also be applied to a single DT or service, but this presents some difficulty. The design patterns can be in conflict about some design choices and then it is up to the designer to decide on a suitable compromise. These compromises are largely informed by the rationale and implication sections of the needs and constraints analysis, as well as the constraints and considerations section of the services characterisation. However, without metrics to refer to, it can be hard to decide what would be a good compromise.

Despite the uncertainty inherent to some of the previous steps, the design patterns could still be applied to guide the design of a feasible architecture. One issue that arose while designing the communication for the architecture was services navigation. Navigating between the widespread services could be cumbersome and thus it could be beneficial to have a centralised dashboard within a DTA that serves all the DTs below it. This will also help limit the complexity of designing the central UI, where the central UI would redirect to a DTA dashboard instead of serving as the single dashboard to all the services. This may also be beneficial for security, for reliability when considering intermittent networks and for the separation of concerns.

### 14.2.5 Verification and validation

The verification and validation step was not discussed in detail, but some observations can still be made based on the heliostat field case study. The implications paragraph in the needs and constraints analysis tables is useful when

identifying which needs and constraints still need to be satisfied. It may be beneficial to include a secondary table that only summarises the implications.

The experiments performed, as part of the heliostat field case study, also demonstrated how the performance metrics can be applied to validate the performance of a DT. Using host machine logs, as well as DT program logs are useful to identify faults and inconsistencies in the system.

### 14.2.6 Suggestions for future work

This section makes suggestions for future work.

- The distinction between system-level requirements and service-level requirements is not always clear. Initially, the FRs are imposed on the system but some of the FRs are delegated to services in different DTs as the design progresses. This transition from system-level to service-level is not always clear and it can cause confusion, particularly when more than one service addresses an FR and when more than one FR is addressed within a service. Therefore, it could be useful to investigate a taxonomy or mechanism that can help discern what requirements are applied to a given level of abstraction.
- In table format it can be hard to identify which needs are overlapping or conflicting. Therefore, an additional diagram could be added to help clarify need and requirement interactions. Furthermore, nested and chained needs can be difficult to deal with and the framework does not make provision for handling this.
- More research is required to determine the intricacies of developing a DT hierarchy for a continuous network. The continuous nature of the network creates a higher degree of dependency between DTs at the same level of aggregation and thus more horizontal data sharing is required. Network elements such as boundary nodes will have data applicable to both DTs that represent the sections being divided by the boundary. How such data should be shared is uncertain. For example, should DTs communicate directly through a message-oriented middleware or with a request-response protocol, should DTs communicate through a DTA or should DTs share their data via API request as is typically done with a service in the services network.
- Further work is required to determine the efficacy of the services allocation and the joining of the DT hierarchy and the service network. The subdivision of the services network, as presented in the smart city case study, may also prove useful and should be further investigated.
- The framework has not considered having a central DT dashboard separate from the services network. It may be beneficial to have a secondary dashboard that can serve as the central dashboard for the local DTs. This may

also be particularly relevant when there are numerous locally hosted services in DTs that may need to function despite external network failures.

- Some services can be implemented with multiple spans of reality. For example, the mirror service in the heliostat field case study can be used to monitor and control single heliostats or the entire field. Such services present a trade-off decision where the service can either be completely centralised in a high-level DT for maintainability or it can be distributed for better performance efficiency and reliability. However, it is uncertain whether this service should be separated without metrics to determine the effects.
- More thorough evaluation is still required for the reliability, maintainability, compatibility and security design patterns. These design patterns have only been assessed at a high level.
- Testing parameters, methods and procedures should still be added to the verification and validation section.
- Some aspects related to data quality require more attention. Suitable methods for handling aspects such as the data consistency, veracity and persistence (see Section 9.2.1), should be investigated in more detail. Furthermore, additional data quality aspects, such as data provenance, can also be considered. Data provenance is concerned with the documentation of the origin of a piece of data, as well as documenting how and why it got to its present position (Gupta, 2009).

# 15 Conclusion

A system of DTs is able to represent complex physical systems, while maintaining the separation of concerns. This is enabled through hierarchical aggregation of DTs, while also making use of a services network.

The main objective of this dissertation was to develop a design framework to guide the design of a DT aggregation architecture, or a system of DTs, to reflect complex systems. The design framework had to enable the systematic, traceable design of a system of DTs, while remaining domain independent.

To achieve this objective, the dissertation investigated the needs related to managing physical system complexity. Considering these needs, the dissertation presents a design framework to help design a system of DTs, according to hierarchical aggregation principles, to meet the needs identified in a particular case. The design framework is arranged in six steps: 1) needs and constraints analysis, 2) physical system decomposition, 3) services allocation, 4) performance and quality considerations, 5) implementation considerations and 6) verification and validation. These design steps were then moulded into six design patterns, which simplify the design process by focussing of key quality attributes. The quality attributes considered for the design patterns are performance efficiency, reliability, maintainability, compatibility, portability and security.

The use of the design framework was then demonstrated and validated through three case studies, i.e. two high-level case studies and one detailed case study. The high-level case studies were a water distribution system and a smart city. Each of these high-level case studies presented unique characteristics, such as the continuous nature of the water distribution network and the mobility of elements within the smart city. For each of these case studies, the design framework, along with the applicable design patterns, was able to guide the design of a feasible architectures for a system of DTs.

The detailed case study, which considered a heliostat field, allowed for a more in-depth demonstration and validation of the design framework. As with the high-level case studies, the design framework was able to guide the design of a feasible architecture for a system of DTs to represent a heliostat field. In particular, the performance efficiency and portability design patterns were applied to produce the resulting architecture. The architecture was then validated through an implementation case study, where the scalability and portability of the architecture was tested.

The scalability of the architecture was tested through the scalability experiments, which considered three types of broker and aggregation scenarios: a local Mosquitto broker - which performed local aggregation, a cloud-based Mosquitto

broker – which performed WAN aggregation and GCP broker – which performed cloud-based aggregation. The scalability experiments demonstrated the scalability of the architecture for different message frequencies and different message sizes, as well as through DTA partitioning. The experiments validated the design choices made according to the performance efficiency design pattern.

The portability of the architecture was tested through the reconfigurability experiments, which considered three reconfiguration scenarios: the addition and removal of a DTI, the addition and removal of a DTA and the addition and removal of a broker. Each of these reconfigurations could be made with just a configuration file change. However, there are cases of more extensive reconfiguration that require source code changes. These experiments helped to validate the design choices made according to the portability design pattern.

In general, the design framework achieved its objectives. It was successfully applied to three different case studies, that each presented different challenges.

# References

Abbott, M. 2020. *The Scale Cude*. [Online], Available: https://akfpartners.com/growth-blog/scale-cube [2021, September 20].

Aderaldo, C.M., Mendonça, N.C., Pahl, C. & Jamshidi, P. 2017. *Benchmark Requirements for Microservices Architecture Research*. in Proceedings - 2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering, ECASE 2017. 8–13.

Adolphs, D.P., Bedenbender, D.H., Dirzus, D.D., Ehlich, M., Epple, P.U., Hankel, M., Heidel, R., Hoffmeister, D.M., et al. 2015. *Status Report - RAMI4.0.* [Online], Available: https://www.zvei.org/fileadmin/user_upload/Presse_und_Medien/Publikation en/2016/januar/GMA_Status_Report__Reference_Archtitecture_Model_Industrie _4.0__RAMI_4.0_/GMA-Status-Report-RAMI-40-July-2015.pdf.

Akbulut, A. & Perros, H.G. 2019. *Performance Analysis of Microservice Design Patterns*. in IEEE Internet Computing, vol. 23, no. 6, pp. 19-27.

Archer, E., Landman, W., Malherbe, J., Tadross, M. & Pretorius, S. 2019. *South Africa's winter rainfall region drought: A region in transition?* Climate Risk Management. 25(April):100188.

Bachmann, F., Bass, L. & Nord, R. 2007. *Modifiability Tactics.* [Online], Available: www.sei.cmu.edu/publications/pubweb.html.

Bajpai, V. & Gorthi, R.P. 2012. *On Non-Functional Requirements : A Survey*. 2012 IEEE Students' Conference on Electrical, Electronics and Computer Science On. 8–11.

Balalaie, A., Heydarnoori, A., Jamshidi, P., Tamburri, D.A. & Lynn, T. 2018. *Microservices migration patterns*. Software - Practice and Experience. 48(11):2019–2042.

Bao, J., Guo, D., Li, J. & Zhang, J. 2018. *The modelling and operations for the digital twin in the context of manufacturing*. Enterprise Information Systems. 13(4):534–556.

Bekker, A. 2018. *Exploring the blue skies potential of digital twin technology for a polar supply and research vessel*. in Marine Design XIII Vol. 1. 135–146.

Berrone, P., Ricart, J.E., Carrasco, C. & Duch, A. 2018. *IESE cities in motion index 2018*. University of Navarra.

Bertoli, A., Cervo, A., Rosati, C.A. & Fantuzzi, C. 2021. *Smart node networks orchestration: A new e2e approach for analysis and design for agile 4.0 implementation*. Sensors. 21(5):1–25.

Bonér, J., Farley, D., Kuhn, R. & Thompson, M. 2014. *The Reactive Manifesto.* [Online], Available: https://www.reactivemanifesto.org/ [2021, September 01].

Bonnet, L., Laurent, A., Sala, M., Laurent, B. & Sicard, N. 2011. *Reduce, you say: What NoSQL can do for data aggregation and BI in large repositories.* in Proceedings - International Workshop on Database and Expert Systems Applications, DEXA. 483–488.

Bonomi, F., Milito, R., Zhu, J. & Addepalli, S. 2012. *Fog computing and its role in the internet of things*. MCC'12 - Proceedings of the 1st ACM Mobile Cloud Computing Workshop. 13–15.

Borangiu, T., Oltean, E., Raileanu, S., Anton, F., Anton, S. & Iacob, I. 2019. *Embedded digital twin for ARTI-type control of semi-continuous production processes.* in Service Orientated, Holonic and Multi-agent Manufacturing Systems for Industry of the Future - Proceedings of SOHOMA 2019 T. Borangiu, P. Leitão, V. Botti, D. Trentesaux, & A.G. Boggino (eds.). Springer Nature Switzerland AG T. 113–133.

Bourque, P. & Fairley, R.E. 2014. *Guide to the software engineering body of knowledge (swebok)* V3.0. [Online], Available: http://www.swebok.org

Brandenbourger, B. & Durand, F. 2018. *Design Pattern for Decomposition or Aggregation of Automation Systems into Hierarchy Levels*. in IEEE International Conference on Emerging Technologies and Factory Automation, ETFA Vols 2018-September. IEEE. 895–901.

BrightSource. 2013. *Ivanpah Project Facts*. [Online], Available: www.brightsourceenergy.com.

Brooks, F. 1995. *The Mythical Man-Month.* Anniversary ed. Boston, MA,USA: Addison Wesley Longman Inc.

Brown, R.R., Keath, N. & Wong, T.H.F. 2009. *Urban water management in cities: historical , current and future regimes*. Water Science and Technology. 59(5):847–855.

BSI, ISO & IEC. 2011. *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*. BSI Standard Publication. First Edition.

Buenabad-Chavez, J., Kecskemeti, G., Tountopoulos, V., Kavakli, E. & Sakellariou, R. 2018. *Towards a Methodology for RAMI4.0 Service Design*. in 2018 Sixth International Conference on Enterprise Systems (ES). 188–195.

Butler, D., Farmani, R., Fu, G., Ward, S. & Diao, K. 2014. A New Approach to Urban Water Management : Safe and Sure. Procedia Engineering. 89:347-354.

CDBB. 2018. *The Gemini Principles*. Centre of Digital Built Britain: University of Cambridge, UK.

Chung, L. & Do Prado Leite, J.C.S. 2009. *On non-functional requirements in software engineering*. Borgida, V.K. Chaudhri, P. Giorgini, & E.S. Yu (eds.) Conceptual Modelling: Foundations and Applications. Lecture Notes in Computer Science Vol. 5600 LNCS. A.T. Springer, Berlin, Heidelberg. 363–379.

Ciavotta, M., Alge, M., Menato, S., Rovere, D. & Pedrazzoli, P. 2017. *A Microservice-based Middleware for the Digital Factory*. Procedia Manufacturing. 11(June):931–938.

Ciavotta, M., Bettoni, A. & Izzo, G. 2018. *Interoperable meta model for simulation-in-the-loop.* in Proceedings - 2018 IEEE Industrial Cyber-Physical Systems, ICPS 2018 IEEE. 702–707.

Ciavotta, M., Maso, G.D., Rovere, D., Tsvetanov, R. & Menato, S. 2020. *Towards the Digital Factory: A Microservices-Based Middleware for Real-to-Digital Synchronization*. In: Bucchiarone, A. et al. (eds) Microservices. Springer, Cham. 273–297.

Cisco. 2015. *Cisco Fog Computing Solutions: Unleash the Power of the Internet of Things*. Cisco. [Online], Available: https://docplayer.net/20003565-Cisco-fog-computing-solutions-unleash-the-power-of-the-internet-of-things.html.

Clark, A.G., Walkinshaw, N. & Hierons, R.M. 2021. *Test case generation for agent-based models: A systematic literature review*. Information and Software Technology. 135.

Cruz, N.C., Alvarez, J.D., Redondo, J.L., Fernández-Reche, J., Berenguel, M., Monterreal, R. & Ortigosa, P.M. 2018*. A new methodology for building-up a robust model for heliostat field flux characterization.* Solar Energy. 173:578–589.

Dierks, T. & Rescorla, E. 2008. *The Transport Layer Security (TLS) Protocol Version 1.2.* [Online], Available: https://www.semanticscholar.org/paper/The-Transport-Layer-Security-(TLS)-Protocol-Version-Dierks-Rescorla/6a74a8573cb1bd15c5f4fa4e047613d2340e61b9

Duffie, N.A., Chitturi, R. & Mou, J.I. 1988. *Fault-tolerant heterarchical control of heterogeneous manufacturing system entities*. Journal of Manufacturing Systems. 7(4):315–328.

Durão, L.F.C.S., Haag, S., Anderl, R., Schützer, K. & Zancul, E. 2018. *Digital twin requirements in the context of industry 4.0*. IFIP Advances in Information and Communication Technology. 540:204–214.

Engel, T., Langermeier, M., Bauer, B. & Hofmann, A. 2018. *Evaluation of microservice architectures: A metric and tool-based approach.* In: Mendling J., Mouratidis H. (eds) Information Systems in the Big Data Era. CAiSE 2018. Lecture Notes in Business Information Processing, vol 317. Springer, Cham.

Erikstad, S.O. & Bekker, A. 2021. *Design Patterns for Intelligent Services Based on Digital Twins*. In Bertram V. (ed.) 20th International Conference on Computer and IT Applications in the Maritime Industries. 235–245.

Fadlalla, A. 2005. *An experimental investigation of the impact of aggregation on the performance of data mining with logistic regression*. Information and Management. 42(5):695–707.

Ferrández-Pastor, F.J., García-Chamizo, J.M., Nieto-Hidalgo, M. & Mora-Martínez, J. 2018. *Precision agriculture design method using a distributed computing architecture on internet of things context*. Sensors. 18(6).

Gadge, S. & Kotwani, V. 2017. *Microservice Architecture : API Gateway Considerations.* San Jose. [Online], Available: https://www.globallogic.com/wp-content/uploads/2017/08/Microservice-Architecture-API-Gateway-Considerations.pdf.

Galster, M. & Bucherer, E. 2008. *A taxonomy for identifying and specifying non-functional requirements in service-oriented development*. Proceedings - 2008 IEEE Congress on Services, SERVICES 2008. PART 1:345–352.

van Geest, M., Tekinerdogan, B. & Catal, C. 2021. *Design of a reference architecture for developing smart warehouses in industry 4.0*. Computers in Industry. 124:103343.

Givehchi, O., Imtiaz, J., Trsek, H. & Jasperneite, J. 2014. *Control-as-a-service from the cloud: A case study for using virtualized PLCs*. in IEEE International Workshop on Factory Communication Systems - Proceedings, WFCS IEEE. 1–4.

Glaessgen, E. & Stargel, D. 2012. *The Digital Twin Paradigm for Future NASA and U.S. Air Force Vehicles*. in Structures, Structural Dynamics, and Materials Conference. 1–14.

Greater London Authority. 2021. *London Datastore*. [Online], Available: https://data.london.gov.uk/ [2021, October 24].

Grieves, M. & Vickers, J. 2016. *Digital twin: Mitigating unpredictable, undesirable emergent behavior in complex systems*. in Kahlen F.J., Flumerfelt S., & Alves A. (eds.) Transdisciplinary Perspectives on Complex Systems: New Findings and Approaches. Cham: Springer International. 85–113.

Griffiths, H. 2018. *Smart city demonstrators: a global review of challenges and lessons learned*. Future Cities Catapult. [Online] Available: https://cp.catapult.org.uk/wp-content/uploads/2021/01/SMART-CITY-DEMONSTRATORS-A-global-review-of-challenges-and-lessons-learned.pdf

Gupta A. 2009. *Data Provenance*. In: LIU L., ÖZSU M.T. (eds) Encyclopedia of Database Systems. Springer, Boston, MA. https://doi.org/10.1007/978-0-387-39940-9_1305

Harper, E., Malakuti, S. & Ganz, C. 2019. *Digital Twin Architecture and Standards*. IIC Journal of Innovation. (November):1–12.

Hoagland, S. n.d. *System id: KY 12*. [Online], Available: https://uknowledge.uky.edu/wdst/index.2.html

Huang, M., Liu, A., Xiong, N.N., Wang, T. & Vasilakos, A. V. 2020. *An effective service-oriented networking management architecture for 5G-enabled internet of things*. Computer Networks. 173(September 2019):107208.

Human, C., Kruger, K. & Basson, A.H. 2021. *Digital Twin Data Pipeline using MQTT in SLADTA*. in Service Orientated, Holonic and Multi-agent Manufacturing Systems for Industry of the Future - Proceedings of SOHOMA 2020 Borangiu T., Leitão P., Botti V., Trentesaux D., & Boggino A.G. (eds.). Springer

Internet Engineering Task Force (IETF). 2012. *The OAuth 2.0 Authorization Framework*. [Online], Available: https://datatracker.ietf.org/doc/html/rfc6749 [2021, September 28].

Ismail, A., Truong, H.L. & Kastner, W. 2019. *Manufacturing process data analysis pipelines: a requirements analysis and survey*. Journal of Big Data. 6(1):1–26.

ISO & IEC. 2021. *ISO/IEC 25010*. [Online], Available: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010 [2021, September 01].

ISO, IEC & IEEE International Standard. 2010. *Systems and software engineering – vocabulary*. in ISO/IEC/IEEE 24765:2010(E).

Jolly, M.D., Lothes, A.D., Sebastian Bryson, L. & Ormsbee, L. 2014. *Research Database of Water Distribution System Models.* Journal of Water Resources Planning and Management. 140(4):410–416.

Kajati, E., Papcun, P., Liu, C., Zhong, R.Y., Koziorek, J. & Zolotova, I. 2019. *Advanced Engineering Informatics Cloud based cyber-physical systems : Network evaluation study*. Advanced Engineering Informatics. 42(August):100988.

Karabey Aksakalli, I., Çelik, T., Can, A.B. & Tekinerdoğan, B. 2021. *Deployment and communication patterns in microservice architectures: A systematic literature review.* Journal of Systems and Software. 180.

Karanjkar, N., Joglekar, A., Mohanty, S., Prabhu, V., Raghunath, D. & Sundaresan, R. 2018. *Digital twin for energy optimization in an SMT-PCB assembly line*. in Proceedings - 2018 IEEE International Conference on Internet of Things and Intelligence System, IOTAIS 2018 IEEE. 85–89.

Kondepudi, S.N. Ramanarayanan, V. Jain, A., Singh, G.N., Nitin Agarwal, N.K. Kumar, R., Singh, R., Bergmark, P., Hashitani, T. & Gemma, P. 2014. *Smart sustainable cities analysis of definitions.* The ITU-T focus group for smart sustainable cities.

Koren, Y. & Shpitalni, M. 2010. *Design of reconfigurable manufacturing systems*. Journal of Manufacturing Systems. 29(4):130–141.

Kritzinger, W., Traar, G., Henjes, J., Sihn, W. & Karner, M. 2018. *Digital Twin in manufacturing: A categorical literature review and classification*. IFAC-PapersOnLine. 51(11):1016–1022.

Kruger, K. & Basson, A.H. 2019. *Evaluation criteria for holonic control implementations in manufacturing systems.* International Journal of Computer Integrated Manufacturing. 32(2):148–158.

Kuhn, T., Schnicke, F. & Oliveira Antonino, P. 2020. *Service-Based Architectures in Production Systems: Challenges, Solutions Experiences*. 2020 ITU Kaleidoscope: Industry-Driven Digital Transformation, ITU K 2020.

Lamb, K. 2019. *Principle-based digital twins: a scoping review*. Centre for Digital Built Britain: University of Cambridge, UK.

Lindsay, D., Gill, S.S., Smirnova, D. & Garraghan, P. 2021. *The evolution of distributed computing systems: from fundamental to new frontiers*. Computing 103:1859-1878.

Liu, Q., Leng, J., Yan, D., Zhang, D., Wei, L., Yu, A., Zhao, R., Zhang, H., et al. 2020. *Digital twin-based designing of the configuration, motion, control, and optimization model of a flow-type smart manufacturing system*. Journal of Manufacturing Systems. 58(Part B):52-64.

Longo, F., Nicoletti, L. & Padovano, A. 2019. *Ubiquitous knowledge empowers the Smart Factory: The impacts of a Service-oriented Digital Twin on enterprises' performance.* Annual Reviews in Control. 47:221–236.

Lutters, E. 2018. *Pilot production environments driven by digital twins.* South African Journal of Industrial Engineering. 29(3 Special Edition):40–53.

Lutters, E. & Damgrave, R. 2019. *The development of Pilot Production Environments based on digital twins and virtual dashboards*. in Procedia CIRP Vol. 84. Elsevier B.V. 94–99.

Lutze, R. 2019. *Digital Twins in eHealth: Prospects and Challenges Focussing on Information Management*. in Proceedings - 2019 IEEE International Conference on Engineering, Technology and Innovation, ICE/ITMC 2019.

Maier, M.W. 1999. *Architecting principles for systems-of-systems*. Systems Engineering. 1(4):267–284.

Mairiza, D., Zowghi, D. & Nurmuliani, N. 2010*. An investigation into the notion of non-functional requirements*. in Proceedings of the ACM Symposium on Applied Computing. 311–317.

Malan, K.J. 2014. *A Heliostat Field Control System*. Master's Thesis. Stellenbosch University. [Online], Available: http://scholar.sun.ac.za/handle/10019.1/86674

Márquez, G., Villegas, M.M. & Astudillo, H. 2018. *An Empirical Study of Scalability Frameworks in Open Source Microservices-based Systems*. in Proceedings - International Conference of the Chilean Computer Science Society, SCCC Vol. November.

Minerva, R., Lee, G.M. & Crespi, N. 2020. *Digital Twin in the IoT Context: A Survey on Technical Features, Scenarios, and Architectural Models*. Proceedings of the IEEE. 108(10):1785–1824.

Mitchell, M. 2009. *Complexity: A guided tour*. Oxford University Press, Inc.

Mohanty, S.P., Choppali, U. & Kougianos, E. 2016. *Everything you wanted to know about smart cities*. IEEE Consumer Electronics Magazine. 5(3):60–70.

Moyne, J., Qamsane, Y., Balta, E.C., Kovalenko, I., Faris, J., Barton, K. & Tilbury, D.M. 2020. *A Requirements Driven Digital Twin Framework: Specification and Opportunities.* IEEE Access. 8:107781–107801.

O'Brien, L., Merson, P. & Bass, L. 2007. *Quality attributes for service-oriented architectures*. in Proceedings - ICSE 2007 Workshops: International Workshop on Systems Development in SOA Environments, SDSOA'07. 3–9.

O'Donovan, P., Leahy, K., Bruton, K. & O'Sullivan, D.T.J. 2015. *An industrial big data pipeline for data-driven analytics maintenance applications in large-scale smart manufacturing facilities*. Journal of Big Data. 2:25(1):1–26.

Odun-Ayo, I., Ananya, M., Agono, F. & Goddy-Worlu, R. 2018*. Cloud Computing Architecture: A Critical Analysis*. Proceedings of the 2018 18th International Conference on Computational Science and Its Applications, ICCSA 2018.

Owen, D.A.L. 2018. *The Technologies and Techniques Driving Smart Water.* in Smart Water Technologies and Techniques: Data Capture and Analysis for Sustainable Water Management Vol. 1. 57–78.

Padovano, A., Longo, F., Nicoletti, L. & Mirabelli, G. 2018. *A Digital Twin based Service Oriented Application for a 4.0 Knowledge Navigation in the Smart Factory.* in IFAC-PapersOnLine Vol. 51. Elsevier B.V. 631–636.

Page, S.E. 2009. *Understanding Complexity*. Chantilly, VA, USA: The Teaching Company.

Pan, Z., Shi, J. & Jiang, L. 2020. *A Novel HDF-Based Data Compression and Integration Approach to Support BIM-GIS Practical Applications*. Advances in Civil Engineering. 2020.

Pargmann, H., Euhausen, D. & Faber, R. 2018. *Intelligent big data processing for wind farm monitoring and analysis based on cloud-Technologies and digital twins: A quantitative approach*. in 2018 3rd IEEE International Conference on Cloud Computing and Big Data Analysis. 233–237.

Pathak, J., Jiang, Y., Honavar, V. & McCalley, J. 2006. *Condition data aggregation with application to failure rate calculation of power transformers*. in Proceedings of the Annual Hawaii International Conference on System Sciences. 1–10.

Pernici, B., Plebani, P., Mecella, M., Leotta, F., Mandreoli, F., Martoglia, R. & Cabri, G. 2020. *Agilechains: Agile supply chains through smart digital twins*. in Baraldi P., Di Maio F., & Zio E. (eds.). Proceedings of the 30th European Safety and Reliability Conference and the 15th Probabilistic Safety Assessment and Management Conference. Singapore: Research Publishing. 2678–2684.

Petrova-Antonova, D., Manova, D. & Ilieva, S. 2020. *Testing web service compositions: Approaches, methodology and automation.* Advances in Science, Technology and Engineering Systems. 5(1):159–168.

Poort, E.R. & De With, P.H.N. 2004. *Resolving requirement conflicts through non-functional decomposition*. in Proceedings - Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004). 145–154.

Pourghebleh, B. & Navimipour, N.J. 2017. *Data aggregation mechanisms in the Internet of things: A systematic review of the literature and recommendations for future research*. Journal of Network and Computer Applications. 97(April):23–34.

Redelinghuys, A.J.H. 2019. An Architecture for the Digital Twin of a Manufacturing Cell. PhD Dissertation. Stellenbosch University. [Online], Available: http://scholar.sun.ac.za/handle/10019.1/108283.

Redelinghuys, A.J.H., Basson, A.H. & Kruger, K. 2020. *A six - layer architecture for the digital twin: a manufacturing case study implementation*. Journal of Intelligent Manufacturing. 31:1383–1402.

Redelinghuys, A.J.H., Kruger, K. & Basson, A.H. 2020. *A six-layer architecture for digital twins with aggregation*. Studies in Computational Intelligence. 853:171–182.

Rossman, L.A. 2000. EPANET 2 User's Manual EPA/600/R-00/57. [Online], Available: https://www.epa.gov/water-research/epanet.

Sage, A.P. & Cuppan, C.D. 2001. *On the systems engineering and management of systems of systems and federation of systems.* Information knowledge systems management journal. 2(4):325–345.

SAICE.2017. *SAICE 2017 Infrastructure Report Card for South Africa*. South African Institution of Civil Engineering. [Online], Available: https://saice.org.za/wp-content/uploads/2017/09/SAICE-IRC-2017.pdf

Salah, T., Zemerly, M.J., Yeun, C.Y., Al-Qutayri, M. & Al-Hammadi, Y. 2016. *The evolution of distributed systems towards microservices architecture*. in 2016 11th International Conference for Internet Technology and Secured Transactions, ICITST 2016. Infonomics Society. 318–325.

Santana, C., Andrade, L., Delicato, F.C. & Prazeres, C. 2021. *Increasing the availability of IoT applications with reactive microservices*. Service Oriented Computing and Applications. 15(2):109–126.

SEBoK Editorial Board. 2021. *The Guide to the Systems Engineering Body of Knowledge (SEBoK)*, v. 2.4, R.J. Cloutier (Editor in Chief). Hoboken, NJ: The Trustees of the Stevens Institute of Technology.

Shangguan, D., Chen, L. & Ding, J. 2019. *A hierarchical digital twin model framework for dynamic cyber-physical system design*. in ACM International Conference Proceeding Series Vol. Part F1476. 123–129.

Sharma, S.K. & Vairavamoorthy, K. 2009. *Urban water demand management: prospects and challenges for the developing countries*. Water and Environment Journal.23(3):210–218.

Silva, B.N., Khan, M. & Han, K. 2018. *Towards sustainable smart cities: A review of trends, architectures, components, and open challenges in smart cities.* Sustainable Cities and Society. 38(February):697–713.

Simon, H.A. 1996. *The Sciences of the Artificial.* Cambridge, Massachusetts: MIT Press.

Smart Santander. n.d. Santander facility. [Online], Available: https://www.smartsantander.eu/index.php/testbeds/item/132-santander-summary [2021, October 24].

Sparrow, D.E., Kruger, K. & Basson, A.H. 2021. *An architecture to facilitate the integration of human workers in Industry 4.0 environments*. International Journal of Production Research, DOI: 10.1080/00207543.2021.1937747.

Suba, C. 2018. *Data Warehousing Methods and its Applications*. International Journal of Engineering Science Invention (IJESI). 12–19. [Online], Available: www.ijesi.org.

Taibi, D., Lenarduzzi, V. & Pahl, C. 2018. *Architectural patterns for microservices: A systematic mapping study*. in CLOSER 2018 - Proceedings of the 8th International Conference on Cloud Computing and Services Science. 221–232.

Taylor, N., Human, C., Kruger, K., Bekker, A. & Basson, A.H. 2020. *Comparison of Digital Twin Development in Manufacturing and Martime Domains*. In Borangiu T., Leitão P., V. Botti V., Trentesaux D., & Boggino A.G. (eds.) Service Orientated, Holonic and Multi-agent Manufacturing Systems for Industry of the Future - Proceedings of SOHOMA 2019 Volume 853 ed. Springer Nature Switzerland AG. 158–170.

Therrien, J.D., Nicolaï, N. & Vanrolleghem, P.A. 2020. *A critical review of the data pipeline: How wastewater system operation flows from data to intelligence.* Water Science and Technology. 82(12):2613–2634.

Tovarnitchi, V.M. 2017. *Cloud-Based Architectures for Environment Monitoring.* in Proceedings - 2017 21st International Conference on Control Systems and Computer, CSCS 2017. 708–714.

Tovarnitchi, V.M. 2019. *Designing distributed, scalable and extensible system using reactive architectures*. in Proceedings - 2019 22nd International Conference on Control Systems and Computer Science, CSCS 2019. 484–488.

Ullah, A., Azeem, M., Ashraf, H., Alaboudi, A.A., Humayun, M. & Jhanjhi, N.Z. 2021. *Secure Healthcare Data Aggregation and Transmission in IoT - A Survey*. IEEE Access. 9:16849–16865.

University of Kentucky. n.d. *Kentucky Dataset*. [Online], Available: https://uknowledge.uky.edu/wdst/index.2.html [2021, October 18].

VanDerHorn, E. & Mahadevan, S. 2021. *Digital Twin: Generalization, characterization and implementation*. Decision Support Systems. 145(February):113524.

Villalobos, K., Ramírez-Durán, V.J., Diez, B., Blanco, J.M., Goñi, A. & Illarramendi, A. 2020. *A three-level hierarchical architecture for an efficient storage of industry 4.0 data*. Computers in Industry. 121.

Villalonga, A., Negri, E., Fumagalli, L., Macchi, M., Castaño, F. & Haber, R. 2020. *Local Decision Making based on Distributed Digital Twin Framework*. in IFAC World Congress 2020, July 11-17, 2020, Vol. 53. 10568–10573.

Villalonga, A., Negri, E., Biscardo, G., Castano, F., Haber, R.E., Fumagalli, L. & Macchi, M. 2021. *A decision-making framework for dynamic scheduling of cyber-physical production systems based on digital twins.* Annual Reviews in Control. (December 2020).

Xiao, Y., Xie, Q. & Deng, Z. 2018. *A review on heliostat field layout and control strategy of solar tower thermal power plants*. in Proceedings of the 2018 Chinese Automation Congress (CAC) IEEE. 1909–1912.

Zimmermann, O. 2017. *Microservices tenets: Agile approach to service development and deployment*. Computer Science - Research and Development. 32(3–4):301–310.

van Zyl, J.E. 2014. *Introduction to Operation and Maintenance of Water Distribution Systems.* First edit ed. Pretoria, South Africa: Water Research Commission.

# Appendix A: Heliostat field case study details

## A.1 Extended needs and constraints analysis

Table 30 provides a more complete description for each need and NFR related to the heliostat field case study.

**Table 30: Extended list of NFRs for the heliostat field case study**

| **Need** | **Provide for large amounts of data. (Related to N24)** |
|---|---|
| **Related NFR** | Performance efficiency. |
| **Rationale for NFR** | Considering the size of the heliostat field, the amount of data generated by each heliostat and the potential resource constraints, there is a need to handle a large amount of data efficiently. Therefore, resource utilisation, scalability and high throughput are primary concerns and these are sub-characteristics of performance efficiency. |
| **NFR grouping** | Quality attribute. |
| **Implication of NFR** | Use performance efficiency design pattern. |
| **Need** | **Allow for system changes with minimal impact. (Related to N9)** |
| **Related NFRs** | Portability, maintainability. |
| **Rationale for NFRs** | New heliostats, that may utilise newer technology, can be added to the heliostat field during the heliostat field's operational lifetime.<br>Portability – Considering the large number of heliostats, the system requires automatic reconfigurability to be feasible.<br>Maintainability – Modularity will help to minimise dependencies between systems and reusability will help to reduce the number of modules required to represent the heliostat field. |
| **NFR grouping** | Quality attributes. |
| **Implication of NFRs** | Use portability and maintainability design patterns. |
| **Need** | **Provide for resource constrained devices. (Related to N25)** |
| **Related NFR** | Performance efficiency, solution constraint. |
| **Rationale for NFR** | The required performance metrics must be reached with minimal resource usage to increase the longevity of the resource constrained devices. The LCUs of the individual heliostats are battery powered and the batteries are charged using photovoltaic (PV) panels. Therefore, the LCUs are energy constrained and as a result the heliostat control engineers have limited the computational responsibilities of the LCUs. The resource constraints of the CCUs and FCU are unknown. |

| | |
|---|---|
| **NFR grouping** | Quality attribute. |
| **Implication of NFR** | Use performance efficiency design pattern.<br>Additional code, other that the heliostat control program, may not be implemented on the LCUs. |
| **Need** | **Provide for intermittent network availability and limited network bandwidth. (Related to N26)** |
| **Related NFR** | Performance efficiency, reliability. |
| **Rationale for NFR** | The Helio100 field has no wired of fibre connections and only makes use of radio frequency communication between LCUs and CCUs and of WLAN/ethernet between CCUs and FCU(s). Therefore, the local network has good capacity, but because of the location of the heliostat field, the external network connection is likely to be wireless, which is comparatively poor.<br>Performance efficiency – The data throughput must be optimised to allow for the required data acquisition despite the limited network bandwidth.<br>Reliability – The intermittent network availability will require some reliability measures to ensure that data does not go lost. |
| **NFR grouping** | Quality attributes. |
| **Implication of NFR** | The performance efficiency design pattern should be used with elements of the reliability pattern to compensate for the intermittent network availability. |
| **Need** | **Avoid physical resource contention amongst software components. (Related to N27)** |
| **Related NFR** | Performance efficiency, compatibility. |
| **Rationale for NFR** | To optimise hardware usage and thus also save costs, software components will have to be hosted together on a single host machine. Therefore, resource contention may become an issue.<br>Performance efficiency – The digital twins must be resource efficient to allow for multiple DTs per host.<br>Compatibility – Co-existence is a sub-division of compatibility that relates to efficient resource usage of software components to minimise the impact on other components on the same host. |
| **NFR grouping** | Quality attributes. |
| **Implication of NFR** | Use the performance efficiency design pattern and incorporate elements of the compatibility design pattern related to co-existence. |
| **Need** | **Allow for retrofitting, differing levels of technological maturity and integration with existing information systems. (Related to N6 and N7)** |
| **Related NFR** | Compatibility, portability, solution and implementation constraints. |

| | |
|---|---|
| **Rationale for NFR** | The consultants at STERG are responsible for designing the heliostat field and its accompanying control systems. Therefore, the digital twins must be able to integrate with the heliostat field as if it is being fitted onto an existing system.<br>Solution constraint – the design must allow for retrofitting onto a heliostat field similar to the Helio100 field without interfering with the existing control network.<br>Implementation constraints - The Helio100 field makes use of a local PostgreSQL database that serves as the current primary data source of all historical data. Therefore, there is a preference for PostgreSQL because the current engineers are familiar with it.<br>Compatibility – The differing technologies must be interoperable.<br>Portability – System components must be replaceable. |
| **NFR grouping** | Quality attributes, development constraints. |
| **Implication of NFR** | The compatibility and portability design patterns must be applied to ensure that different technologies and components can be replaced without disrupting the system.<br>Some of the technologies related to the data acquisition part of the digital twin are predefined and must be integrated with, such as the use a PostgreSQL database for local storage. |
| **Need** | **Verify and validate the behaviour of DTs in response to system changes. (Related to N10)** |
| **Related NFR** | Maintainability. |
| **Rationale for NFR** | The proper functioning and performance of the heliostat field must be ensured after physical parts are replaced and after software components are updated. Considering that there is a high opportunity for software reuse, the impact of software changes can also be widespread.<br>Maintainability – Testability is a sub-characteristic of maintainability that directly links to the verification and validation of system components. |
| **NFR grouping** | Quality attribute. |
| **Implication of NFR** | Use the maintainability design pattern to improve the testability of the digital twins. |
| **Need** | **Structure the data to prevent it becoming unusable. (Related to N21)** |
| **Related NFR** | Maintainability. |
| **Rationale for NFR** | Reusability of data should be ensured despite the increase in volume. |
| **NFR grouping** | Quality attribute. |
| **Implication of NFR** | Make use of the maintainability design pattern. |
| **Need** | **Provide a cost-effective solution. (Related to N28)** |
| **Related NFR** | Cost constraint. |
| **Rationale for NFR** | The high initial costs of heliostat fields are a deterrent to their adoption and thus the cost must be minimised. |

| | |
|---|---|
| **NFR grouping** | Development NFR. |
| **Implication of NFR** | The cost constraint will limit the amount of development time that can be spent on quality assurance and testing. |
| **Need** | **Allow for easy long-term maintenance and extension. (Related to N14)** |
| **Related NFR** | Maintainability, portability. |
| **Rationale for NFR** | Maintainability – A heliostat field has a long operational lifetime and software maintenance must be provided for the duration of the physical system's lifecycle.<br>Portability – The long operational lifetime of the heliostat field means that new technologies will be developed during the operational lifetime of the heliostat field. Therefore, provision must be made to allow for changes in software and hardware technology. |
| **NFR grouping** | Quality attributes. |
| **Implication of NFR** | Use maintainability and portability design patterns. The portability design pattern is particularly important in this case since STERG emphasised the need to adapt to changes in hardware. |

## A.2 Extended span of reality for the heliostat field

Table 31 presents an extended span of reality for the components of the heliostat field.

**Table 31: Span of reality of the heliostat field components.**

| **Physical component** | **Heliostat with local control unit (LCU)** |
|---|---|
| **Physical system scope** | Individual heliostat |
| **Data characterisation (Data granularity) of data recorded/ generated by physical component** | Stepper motor positions – two, int, between 0 and 200 000, (step counts), generated every minute<br>Battery value – float, between 5.5 and 8.2, (Volts), generated every minute<br>Timestamp – datetime, N/A, (N/A), generated every minute |
| **Data characterisation (Data granularity) of data sent to physical component** | Local coordinates of the sun – See Cluster control unit below.<br>Translated operator control commands – details unknown |
| **Data format** | JSON formatted message |
| **Communication** | Radio Frequency (RF) communication using a serial bus. |
| **Considerations and Constraints (Capacity for interaction)** | LCUs are power constrained and thus the activity of the LCUs need to be minimised.<br>The LCUs only support RF communication. |

| | |
|---|---|
| | The design requires 10 002 individual heliostats and they may differ slightly in composition (e.g. newer heliostats make use of newer components and future heliostats may have more sensors). |
| **Physical component** | **Pod of heliostats** |
| **Physical system scope** | 6 Heliostats |
| **Data characterisation (Data granularity)** | No data added at this level |
| **Data format** | None |
| **Communication** | No communication to a pod |
| **Considerations and Constraints (Capacity for interaction)** | There is no hardware or software implemented at pod level.<br>The design will have 1667 pods. |
| **Physical component** | **Cluster control unit (CCU)** |
| **Physical system scope** | 24 or 30 heliostats (4 or 5 pods) |
| **Data characterisation (Data granularity) of data recorded/ generated by physical component** | Unique identifier for each heliostat – String, N/A, (N/A) static value<br>Status value for each heliostat – String, [start-up, manual move, running, standby, home, calibration, e-stop, offline], (N/A), generated every minute or on request (the CCU generates a status value per minute but an operator can force a state aswell)<br>Translated operational commands – The commands are unknown but the operational commands come from the FCU, are translated by the CCU and sent to the LCU in a format familiar to the LCU.<br><u>Grena algorithm inputs:</u><br>Fractional Universal Time (UT) – Float, between -12.00 and +12.00, (N/A), every minute. Fractional UT is the time in hours and fractions of hours from the Greenwich midnight.<br>Date – three (day – *d,* month – *m,* year - *y*), int, *d* – 1 to 31; *m* – 1 to 12; *y* – 2003 to 2023, (N/A), every minute. The Grena algorithm is valid for 20 years, after which the algorithm parameters must be adjusted.<br>Time difference between UT and terrestrial time – float, unknown, (seconds), every minute.<br>Longitude - Float, unknown, (radians), every minute.<br>Latitude - Float, unknown, (radians), every minute.<br><u>Grena algoritm outputs</u>:<br>global coordinates of the sun – two, Float, unknown, (radians), every minute. The global coordinates are right ascension and declination<br>local coordinates of the sun – three, Float, unknown, (radians), every minute. The local coordinate angles are the hour angle, zenith angle and azimuth angle. |
| **Data characterisation (Data granularity) of** | <u>Grena algorithm inputs:</u><br>Air pressure - See FCU for details |

| | |
|---|---|
| **data sent to physical component** | Ambient temperature – See FCU for details<br>Operation commands – See FCU for details |
| **Data format** | JSON formatted message |
| **Communication** | ZeroMQ messaging over TCP/IP and WLAN/ethernet |
| **Considerations and Constraints (Capacity for interaction)** | Available processing and storage capacity of CCUs is unknown<br>Assuming each CCU controls 4 pods, 417 CCUs will be required. Assuming each CCU controls 5 pods, 334 CCUs will be required. |
| **Physical component** | **Field control unit (FCU)** |
| **Physical system scope** | Six CCUs (Which is the whole field) |
| **Data characterisation (Data granularity) of data recorded/ generated by physical component** | Operational commands (from a user) - String, unknown, (N/A), user driven. |
| **Data characterisation (Data granularity) of data sent to physical component** | Calibration images – images (format unknown), N/A, N/A, 32 images are taken at a rate of 3-5 images a second for every heliostat calibration sequence. A calibration sequence is triggered by a user.<br>Weather data:<br>Direct normal irradiance (DNI) – float, unknown, (W/m^2), every minute.<br>Wind speed – float, unknown, (m/s), every minute<br>Air pressure - Float, unknown, (atm), every minute<br>Ambient temperature – Float, unknown, (°C), every minute. |
| **Data format** | JSON formatted message |
| **Communication** | With CCUs - ZeroMQ messaging over TCP/IP and WLAN/ethernet<br>With weather station – HTTP<br>IP camera (for calibration) – Unknown |
| **Considerations and Constraints (Capacity for interaction)** | Available processing and storage capacity of the FCU is unknown and thus the number of CCUs that can be supported by a single FCU is unknown.<br>The current FCU design stores all the captured data locally in a PostgreSQL database with no reduction being applied. |

## A.3 Extended physical architecture description

The physical architecture in Figure 15 presents the internal structure of DTs implemented as part of the heliostat field system of DTs. Layers 1 and 2 of the DTI were simulated to allow for the flexibility to add and remove heliostats and CCUs as the experiments required. The simulated heliostats and CCUs also allow for a scale of data that would not be feasible in a laboratory environment. The

simulated heliostats and CCUs generated data that mimicked the data of a heliostat field as described in Table 28. This amounted to a message size of 300 bytes which contained all the relevant data for the mirror service concerning a heliostat. However, the frequency that data was generated at was adjusted as required by the various experimental scenarios.

Furthermore, the simulated CCUs collected the data from multiple simulated heliostats before writing the data to a PostgreSQL database in Layer 3. The simulated heliostat's battery values and motor values were programmed to fluctuate and can trigger a simulation status change (if the values deviated from the expected ranges) or the heliostat simulation could generate a "fault" (an artificial fault could be triggered). These value fluctuations and artificial faults were introduced so that typical pre-processing and fault handling could also be mimicked within the DTs' Layer 4.

The DTIs' and DTAs' Layer 4 are custom developed Python programs, as well as some Google Cloud Platform (GCP) services. The Layer 4 composition of both DTIs and DTAs are presented in Figure 18. The DTIs' local Layer 4 consists of five modules: data ingestion, processing, communication, orchestration and configuration. The data ingestion module reads data from the PostgreSQL database. The processing module performs a data format conversion (from csv to JSON), as well as some rule-based checks and fault handling on the data (such as out-of-bound value checks). The communication module contains an MQTT client with the connection and callback logic, as well as SSL/TLS security logic to secure all communications. Two communication modules were used per DT: one to send data to the cloud and another for aggregation. The orchestration module is the DT program's entry point and coordinates the other modules, while the configuration file contains information unique to each DT (such as an ID, SSL/TLS security certificates, connection information, etc.)

The cloud-based portion of Layer 4 is fulfilled using GCP's IoT Core, Pub/Sub and Cloud Functions services. The IoT Core contains an MQTT broker which receives data from one of the local Layer 4's MQTT clients. The Pub/Sub service essentially acts as an MQTT client within the cloud and makes the data in the IoT Core available to the rest of the cloud platform. Cloud Functions are standalone, stateless and temporary functions that are triggered by events within the cloud. In this case study, these functions were triggered by the Pub/Sub service when new data became available within the cloud. Once triggered, the functions write the data to a Firestore database, which is a NoSQL document store.

It should be noted that it is possible to write directly to the Firestore database from the local Layer 4, using the Firestore APIs. However, the method above is generally preferred when many devices are interacting with the database, because

the IoT Core and Pub/Sub services provide support for features such as device monitoring, load balancing and temporary data storage for reliability.

The DTA modules are similar to the DTI modules, except that the DTA does not have a data ingestion module and the processing module aggregates the data instead of doing format conversion. The aggregation is done through a broker and since it is aggregation through Layer 4, it is pre-storage aggregation.

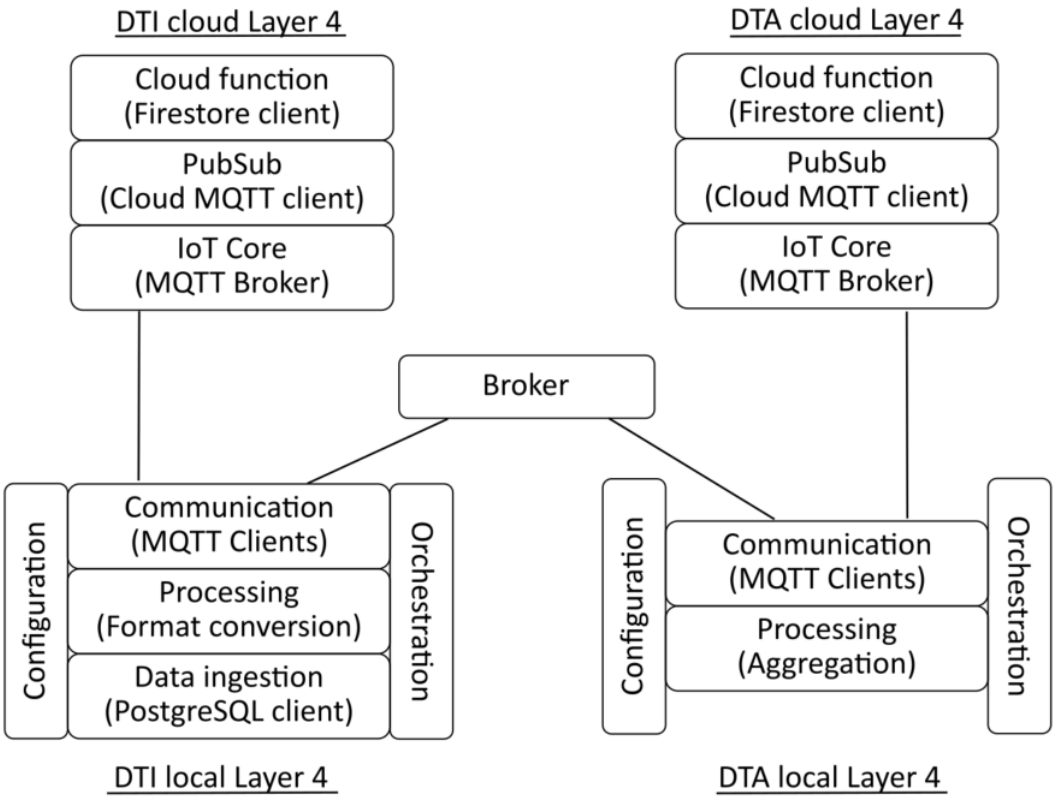



**Figure 18: Layer 4 breakdown of the DTI and the DTA**

The long-term database on Layer 5 is a Firestore database. Firestore is a document store and it was chosen because of its scalability, reconfigurability and compatibility with the JSON data format. Finally, Layer 6 makes use of GCP's App Engine service which is a service that helps deploy web applications within GCP. In this case, App Engine was used to deploy a basic mirror service that simply displayed the data available in the Firestore database. Therefore, the mirror service just served as validation that the data, sent by the simulated heliostats, were captured in long-term storage.

## A.4 Varying message frequency experiments

The purpose of the varying message frequency experiments is to determine the sensitivity of the architecture to changing message frequency (which is a component of throughput as discussed in Section 9.1.2). Scalability is essentially the ratio between throughput and resource usage (as discussed in Section 4.3). The host machines used for the experiments were kept constant and thus at full load the resource usage in each case was also constant. Therefore, the difference in throughput is a result of the different aggregation methods and technologies.

Throughput can be adjusted by changing the message frequency, changing the message size and by increasing the number of messaging channels, i.e. the number of DTIs (as discussed in Section 9.1.2). Therefore, for the experiments that tested the effect of varying message frequency, the message size was kept constant and the number of DTIs was periodically increased as discussed in Section 13.2.3.1. The results of these experiments are presented in Figure 19 and Figure 20. The vertical bars in the figures indicate the range of the values observed in repeated experiments, while the lines pass through the mean values.

Figure 17 presents the *number of DTIs* that could be sustained at different message frequencies, as well as the *collective message threshold* at different message frequencies. The number of DTIs refers to the maximum number of DTIs that could be sustained (measured within the threshold period) and the collective message threshold refers to the number of messages that were processed by the DTA. The message frequency refers to the frequency at which DTIs send their messages and it was controlled by adjusting the *sleep time* of the DTIs. The sleep time is the time in between logic execution cycles of a DTI, where a logic execution cycle refers to the DTI reading data, processing data and sending the data as a message (Section 13.2.2 describes the modules that relate to these steps).

Figure 17 shows that the local Mosquitto broker can sustain the most DTIs at a given message frequency, followed by the cloud-based Mosquitto broker and then the Pub/Sub service. For example, at a frequency of one message per second, the local Mosquitto broker can sustain 25 DTIs, the cloud-based Mosquitto broker can sustain 17 DTIs and the Pub/Sub service can sustain 6 DTIs. Similarly, at a fixed number of DTIs, the local Mosquitto broker can sustain the highest message frequency, followed by the cloud-based Mosquitto broker and then the Pub/Sub service. Furthermore, the collective message threshold fluctuates slightly, but remains relatively consistent across multiple message frequencies. Therefore, the collective message threshold is likely the maximum throughput that can be sustained by the system of DT for the given experiment configuration.

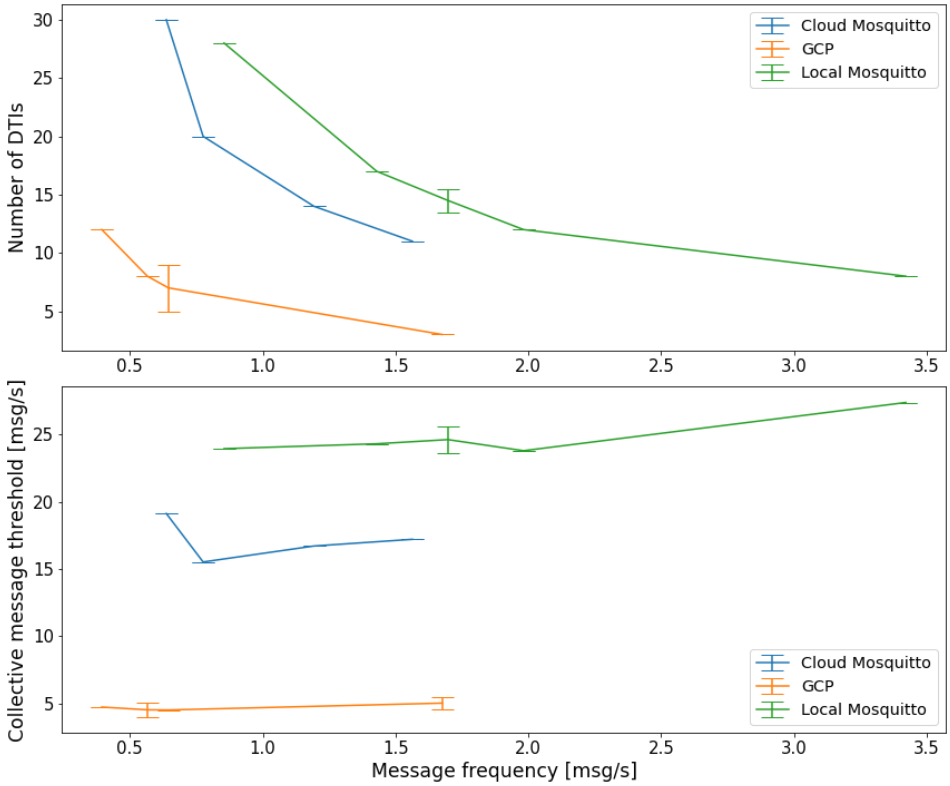



**Figure 19: Number of DTIs and the collective message threshold for different message frequencies.**

The collective message threshold of the cloud-based Mosquitto broker does, however, fluctuate more than the other brokers' collective message threshold. The reason for this deviation from the expected result is uncertain but is likely because of network effects since it does not seem to correlate with the number of DTIs, message frequency, percentage message loss or mean latency.

Figure 20 presents the mean latency and percentage message loss at various message frequencies. Figure 20 shows that the mean latency of the local Mosquitto broker is the lowest and it displays the least variation in latency. The Pub/Sub service displayed the highest latencies, as well as the highest variation in latencies.

Furthermore, the cloud-based Mosquitto broker displays the lowest percentage message loss, followed by the local Mosquitto broker and then the Pub/Sub service. The cloud-based Mosquitto broker also displays the lowest variation in percentage message loss, followed by the local Mosquitto broker and then the Pub/Sub service. However, it is important to note that the DTIs were using a quality of service (QoS) of 0 (QoS = 0 does not guarantee that a message is

received) when aggregating via the local Mosquitto broker. When aggregating via the cloud-based Mosquitto broker, the DTIs were using a QoS of 1 (which is supposed to guarantee that each message is received at least once). Originally the DTIs used a QoS of 0 for both Mosquitto brokers, to minimise latencies and prevent possible data duplication, but the cloud-based Mosquitto broker was losing more that 20% of the messages when QoS was set to 0. Therefore, the DTIs' QoS levels were adjusted to 1 for the cloud-based Mosquitto broker. The DTAs used a QoS of 1 to send data back to the DTIs, regardless of which broker was being used.

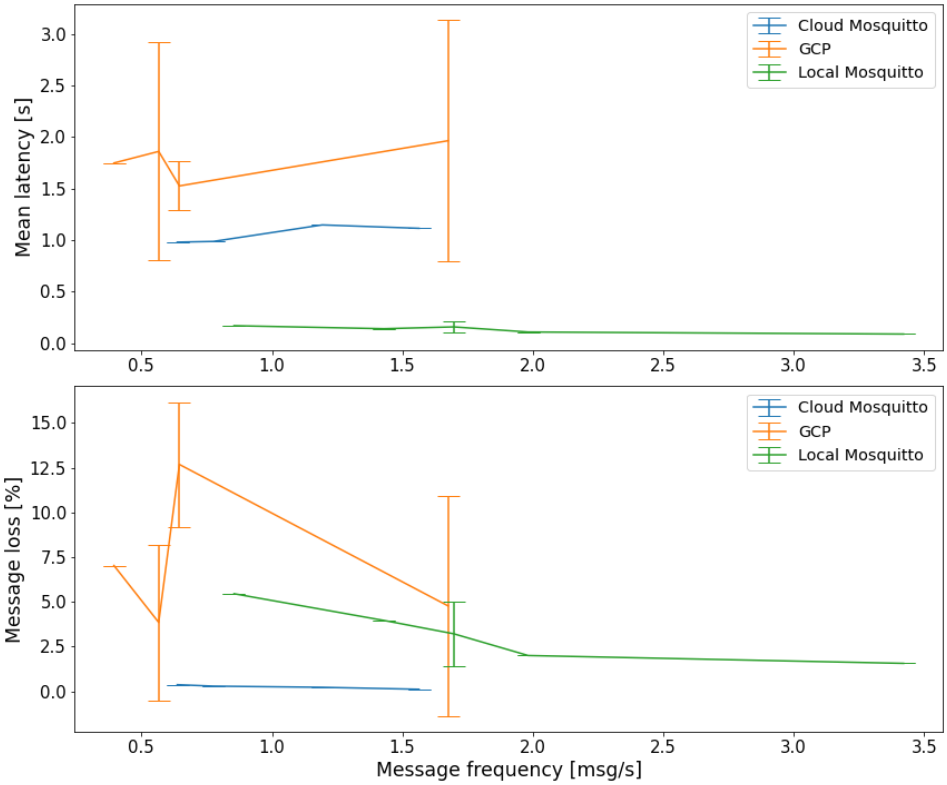



**Figure 20: Mean latency and percentage message loss for different message frequencies.**

The experiment results presented in this section indicate that message frequency of the DTIs have a significant effect on the number of DTIs that can connect to the broker and DTA, but the collective message threshold sustained by the DTA remains relatively constant. This is significant because it indicates that throughput is relatively constant and thus to accommodate more DTIs, the message frequency should be reduced.

Furthermore, these experiments contributed to validating the performance efficiency design pattern by demonstrating how local network aggregation is better for lower latencies and higher throughput than cloud-based aggregation. The reliability of the local network aggregation is likely to be better, with a lower percentage messages loss, less variation in percentage message loss and less variation in latency.

## A.5 Varying message size experiments

The purpose of the varying message size experiments is to determine the sensitivity of the architecture to changes in message size. As mentioned in Appendix A.4, throughput can be adjusted by adjusting the message frequency, message size or number of messaging channels. For the experiments discussed in this section, the message frequency was fixed while the message size was changed. The results of the experiments are presented in Figure 21 and Figure 22.

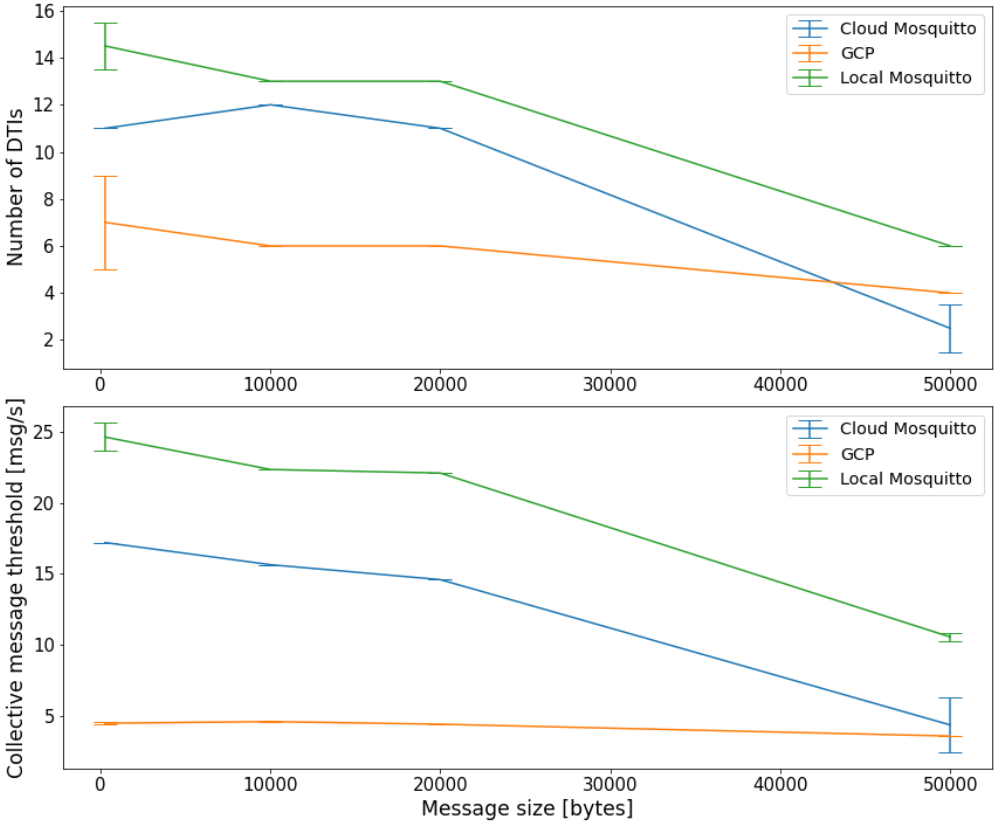



**Figure 21: Number of DTIs and collective message threshold for different message sizes.**

Figure 21 shows that the local Mosquitto broker can sustain the greatest number of DTIs and the greatest collective message threshold, followed by the cloud-based Mosquitto broker and then the Pub/Sub service. Therefore, the local Mosquitto broker again displays the best throughput, followed by the cloud-based Mosquitto broker and then the Pub/Sub service.

The results from Figure 21 further indicate that the effect of message size is not significant before the 20kB message size but thereafter a new bottleneck is reached. Before the 20kB point, the number of DTIs for each broker remains within expected range and the collective message threshold drops off slightly by one message per second for each Mosquitto broker. However, after the 20 kB message size, a significant downward trend can be observed for both Mosquitto brokers. The Pub/Sub service on the other hand remains relatively stable for all the message sizes, where only a slight downward trend is observed for the number of DTIs.

For the Mosquitto brokers, the bottleneck is most likely the network bandwidth. The reasoning behind this is that as the message size increases after the 20kB point, the number of DTIs decreases. Considering that message frequency was kept constant, the throughput is also likely keeping constant at its maximum threshold.

For the Pub/Sub service, the bottleneck is likely a cloud platform throttling limit. The reasoning behind this is that for the varying message frequency and varying message size experiments, the collective message threshold remains relatively constant, around 5 messages per second, regardless of any other changes made to the system. This observation is further supported by Kajati *et al.* (2019), where a similar observation was made while using Microsoft Azure.

Furthermore, Figure 22 shows that the local Mosquitto broker also has the lowest latency, followed by the cloud-based Mosquitto broker and then the Pub/Sub service. The bandwidth bottleneck at the 20 kB point also marked a significant increase in latency for both Mosquitto brokers. The reason for the increase in latency for the Pub/Sub service, however, is uncertain but is likely related to the decrease in percentage message loss.

For the Mosquitto brokers, the DTIs used a QoS level of 0 for aggregation through the local Mosquitto broker, whereas the DTIs used a QoS level of 1 for aggregation through the cloud-based Mosquitto broker (the reason for this is discussed in the previous section). For the smaller message sizes, the cloud-based Mosquitto broker displayed a lower percentage message loss than the local Mosquitto broker. However, as message size increased, the difference in percentage message loss between the brokers became smaller. This is likely because fewer messages were sent at the larger message sizes.

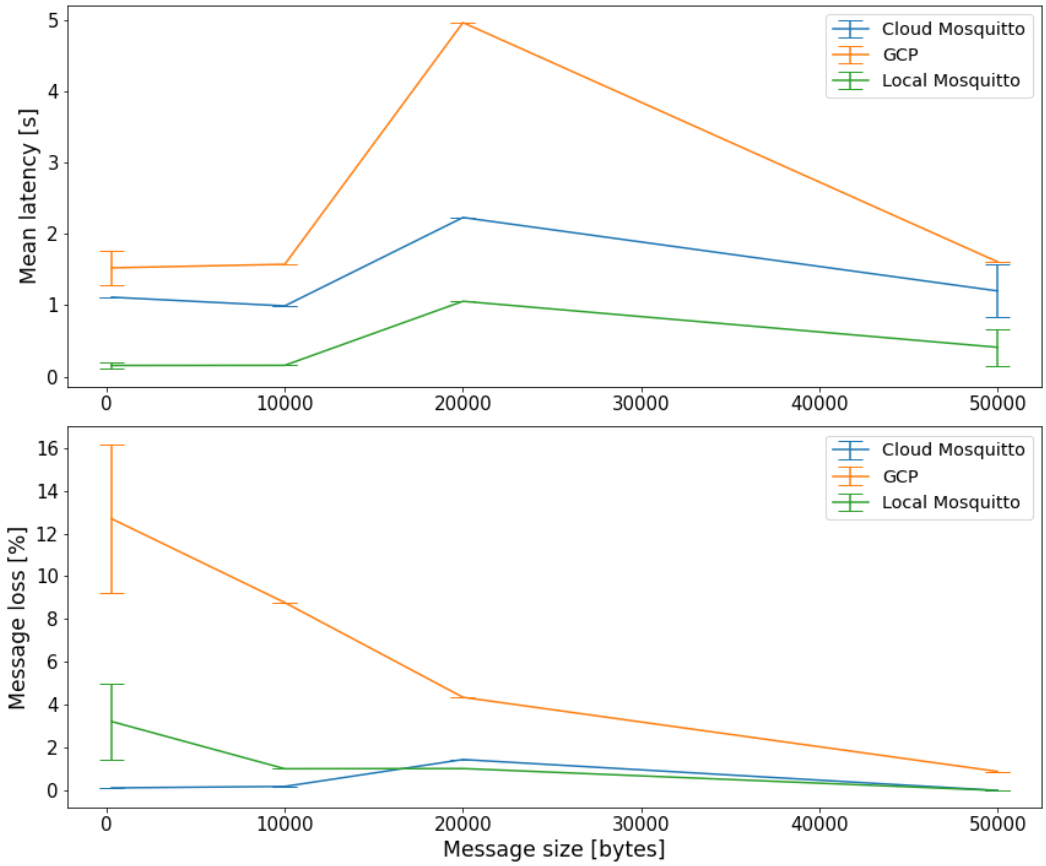



**Figure 22: Mean latency and percentage message loss for different message sizes.**

The experiment results presented in this section indicate that message size is not as significant as message frequency when considering the scalability. However, at sufficiently large message sizes (20 kB or more in this case) the message size does become increasingly more important to consider. The message size experiments also emphasised the need to make provision for poor network connectivity and MQTT was ideal for this. With a QoS level of 1, the message loss was very low, even near maximum DTA capacity. Therefore, in this architecture, MQTT and the Mosquitto broker (which was the message-oriented middleware) allowed the architecture to satisfy N26 in Table 27 (the need to provide for intermittent network connectivity). However, in performance efficiency scenarios, good network hardware, such as ethernet connections and network switches, would make a significant impact.

Furthermore, the results discussed in this section further support the conclusions that local network aggregation has lower latencies and better throughput than cloud-based aggregation.

## A.6 Multiple aggregate, multiple broker experiments

The purpose of the multiple aggregate and multiple broker cases is to demonstrate the scalability of the architecture through partitioning. In particular, four cases were investigated: 1) a local and a cloud-based Mosquitto broker with a single DTA, 2) a local and a cloud-based Mosquitto broker with a DTA dedicated to each broker, 3) a single cloud-based Mosquitto broker with two DTAs and 4) the Pub/Sub service with two DTAs. Each of these cases have a fixed message frequency and fixed message size, while DTIs are periodically added to the system of DTs to gradually increase the messaging load.

The results of the various cases are summarised in Table 32, where each experimental case was conducted twice to consider the repeatability of the results. The respective results of an experimental case are marked as a and b, while the cases have been separated with alternating light and dark rows.

**Table 32: Scalability experiment results for multiple brokers and multiple aggregates**

| Case number | Brokers | DTAs | Threshold number of DTIs | Collective message threshold | Total message loss |
|---|---|---|---|---|---|
| 1a | L&C[i] | 1[ii] | 6 | 10.60 | 0.40% |
| 1b | L&C | 1 | 8 | 10.97 | 0.30% |
| 2a | L&C | 1&4[iii] | 21+[iv] | 36.58+ | 0.56% |
| 2b | L&C | 1&4 | 20 | 34 | 1.95% |
| 3a | C[v] | 1&4 | 20 | 32.67 | 3.74% |
| 3b | C | 1&4 | 21+ | 35.05+ | 4.91% |
| 4a | Pub/Sub | 1&4 | 8 | 7.82 | 0.00% |
| 4b | Pub/Sub | 1&4 | 9 | 8.23 | 2.47% |

[i] - L&C means that both the Local Mosquitto and the Cloud Mosquitto brokers were active.
[ii] - 1 here refers to Host 1. Therefore, the DTA being used is the one on Host 1.
[iii] - 1&4 means that the DTA on Host 1 was active as well as the DTA on Host 4.
[iv] - The "+" operator indicates that the system never reached a limit point.
[v] - C refers to the Cloud Mosquitto broker.

The results of Case 1 (two Mosquitto brokers with one DTA) indicate that adding brokers without adding DTAs decreases the throughput of the system of DTs. The likely reason for this is that the DTA must sustain a third MQTT client (one client for the IoT Core and one client for each Mosquitto broker) and this requires the DTA to sustain an additional thread (each MQTT client requires their own thread because MQTT clients enter a blocking loop to allow for the receipt of messages). Therefore, forcing the DTA to switch between more threads negatively impacts

the throughput of the DTA. This can be observed in the data captured during the experiments which show that the DTA's mean processing time per message goes up from 0,01 – 0,02 seconds per message to 0,03 – 0,05 seconds per message. That is roughly double the processing time and as a result roughly half the throughput when compared to the single DTA and single broker case with the same experimental parameters.

The results of Case 2 (two brokers with two DTAs) indicated that the partitioning of the DTIs between two brokers and two DTAs produced the best throughput. In this case, the DTIs were divided into two groups, where one group was aggregated through one broker to one DTA, while the other group was aggregated through the other broker to the other DTA. This allowed for a collective message threshold of up to 36.58 messages per second, 12 messages per second more (50% increase) than a single local Mosquitto broker and 20 messages per second more (125% increase) than the single cloud-based Mosquitto broker with otherwise the same experimental parameters.

The results of Case 3 (a single cloud-based Mosquitto broker with two DTAs) proved to be very insightful. The single cloud-based Mosquitto broker with two DTAs was able to sustain a collective message threshold twice as high as the cloud-based Mosquitto broker with a single DTA (all other parameters being the same). Therefore, this experimental case indicates that the DTA was the limiting factor in the scalability experiments, involving the Mosquitto brokers, that investigated varying message frequencies presented in Appendix A.4.

It is suspected that the DTA was the bottleneck because it was unable to process and return all the messages fast enough. This means that the collective message threshold is the maximum number of messages that the DTA could receive, process and send back before becoming unstable. The difference in throughput for the local Mosquitto broker versus the cloud-based Mosquitto broker is likely because of a combination of three factors: 1) the difference in messaging latency, 2) the difference in time it takes the DTA to acknowledge and/or resend messages to the respective brokers (the DTA uses a QoS of 1 regardless of the broker as mention in Appendix A.4) and 3) the DTA must likely resend more messages to the cloud-based Mosquitto broker.

Finally, the results of Case 4 (the Pub/Sub service with two DTAs) show that the Pub/Sub service with two DTAs shows about a 78% increase in throughput when compared to a single DTA. This means that the cloud platform throttling discussed in Appendix A.5 is likely being applied to the number of messages being sent and received by each DTA or on the respective host machine that the DTAs are being hosted on.

Case 1 demonstrates the effect of increased processing time within the DTA, where the number of DTIs that could be supported were halved when DTA processing time was doubled. Therefore, this supports the performance efficiency design pattern's suggestion to decentralise processing logic from the DTA to the DTIs where possible to improve performance and to prevent the DTA from being a bottleneck. The results of cases 2, 3 and 4 demonstrates the scalability of the architecture by partitioning the DTA to improve the throughput, where each of these cases displayed significant increases in throughput after partitioning. This also validates the performance efficiency design pattern's recommendation to replicate or partition to improve performance.

Furthermore, during the multiple brokers and DTA experiments it was determined that the DTA is the likely bottleneck for these experiments. To alleviate this bottleneck, the DTA was partitioned (as shown by case numbers 2 and 3 in Table 32) and the throughput was significantly increased.

## A.7 Real world scenario

The previous experiments investigated the system scalability in various configurations**,** as well as the potential limitations to the scalability of the system of DTs. The real-world scenario, presented here, aims to determine how many heliostats can potentially be monitored using one broker and one DTA with typical heliostat field data parameters. The results of the experiments for each broker are summarised in Table 33. The discussion after the table extrapolates the results to determine the potential number of CCUs and heliostats that could be monitored.

**Table 33: Broker results comparison**

| Broker | Threshold number of DTIs | Collective message threshold [msg/s] | Total messages loss | Mean latency [s] |
|---|---|---|---|---|
| GCP Pub/Sub | 28+ | 2.68+ | 2.19% | 1.126 ±0.231 |
| Local Mosquitto | 31+ | 2.92+ | 0.29% | 0.049 ±0.019 |
| Cloud Mosquitto | 32+ | 3.05+ | 0.00% | 0.725 ±0.116 |

In general, the threshold number of DTIs and the collective message threshold for each broker case are similar, where none of the cases reached a limit point. Instead, the maximum number of DTIs that the host machines could sustain, given the broker configuration, was reached. Therefore, the experimental hardware was the limiting factor during these experiments. Despite this limitation, the results are extrapolated to provide an indication of how many heliostats could potentially be sustained.

The experiment shows that the Pub/Sub service can at least sustain 28 DTIs. The messages are also being sent back and forth (because round-trip latency is being measured) and thus it is possible that the data pipeline can sustain a higher number of DTIs when fewer messages are sent back to the DTI. This is particularly likely given the conclusion that the DTA is the bottleneck because it must resend messages to the DTIs (as discussed for Case 3 in the previous section).

Given that the number of threshold messages per second remained relatively constant during the experiments discussed in Appendix A.4, this could be a good indicator of the data pipeline's potential scale. If the collective message threshold for the Pub/Sub service was capped at 4 messages per second, and no other problems are incurred (such as bandwidth limitations, or throttling), the system of DTs could theoretically sustain 41 DTIs. If the DTIs are also only required to send data frequently and not to receive data frequently, the number of threshold DTIs might even be 80. If each DTI had 5 pods connected to it, the number of heliostats represented by system of DTs would be 2400 heliostats.

For the local Mosquitto broker all 31 of the available DTIs were able to connect. Extrapolating according to the same logic as for the Pub/Sub service and based on a collective message threshold of 22 messages per second, the broker should be able to support about 233 DTIs. Then assuming a relatively high frequency of messages sent per second and a low frequency of messages received, the potential number of threshold DTIs could be as high as 460, equating to 13800 heliostats. Similarly, the cloud-based Mosquitto broker could theoretically be able to support 310 DTIs, equating to 9300 heliostats (assuming a message threshold of 15 messages per second).

Based on the extrapolated results, the local Mosquitto broker would be the only broker configuration that could sustain the data capturing requirements of the 5 MW heliostat field. However, further consideration should also be given to the partitioning of the heliostat field amongst two or three DTAs and brokers for the sake of reliability. If this advice were to be followed, the cloud-based Mosquitto broker would also be capable of sustaining the data capturing.

## A.8 Reconfigurability experiments

The purpose of the reconfigurability experiments is to contribute to validating the portability of the aggregation hierarchy, as well as the design choices made according to the portability design pattern. Therefore, this section discusses the effort required to perform certain reconfigurations on the system of DTs and thus it serves as a qualitative evaluation of the portability and maintainability of the architecture. Three reconfiguration scenarios are discussed: adding or removing a DTI, adding or removing a DTA and adding or removing a broker.

**Adding or removing a DTI:**

Adding a new DTI into the system of DTs requires reconfiguration on three components: the DTI's Layer 4, the DTA's Layer 4 and the cloud platform. The reconfigurations required on each component are discussed below.

In Layer 4 of the DTI:

- For communication to Layer 5 of the DTI, the authentication credentials must be generated and the MQTT client must be configured to connect to the right IoT Core device (the IoT Core device is a cloud-based avatar for the actual device which is the DTI's Layer 4 in this case). The authentication credentials consist of the CA certificate for the TLS protocol and a private key to sign the JWT.
- For aggregation and communication through the Mosquitto brokers, the CA certificate, the private key, and the private key certificate for the TLS protocol must be configured. The MQTT client must also be configured to connect to the right broker, connect to the right DTA, and connect with a unique ID. When using the Pub/Sub service, the DTI only requires a service account for authentication and the correct Pub/Sub topic (that links to a certain DTA) must be specified.

In Layer 4 of the DTA:

- For a new DTI to connect to the DTA, only the pod numbers associated with that DTI need to be specified in the DTA's configuration file. The DTA also needs to know the DTI's unique device id, but that is part of the metadata of the JSON message sent by the DTI.

In the cloud platform:

- For communication to a DTI's Layer 5, a new device, with an accompanying public key, must be created within an existing registry of GCP IoT Core. Care must be taken to link the right public-private key pairs. The most effective way to do this is to use the Google SDK and batch script to create all the required IoT Core devices with their accompanying public keys. No changes have to be made in Cloud Pub/Sub, Cloud Functions or in Firestore.
- If aggregation is done using the Pub/Sub service, Cloud IAM must be used to create a service account for the DTI, or an existing service account can also be used.

These are the minimum reconfigurations that need to be done to create a new DTI. None of these reconfigurations require source code changes. All the changes can be done by changing the specifications in the configuration file of the DTI and DTA, respectively. Changes made to the cloud platform components are also configuration changes that are done through a GUI provided by the cloud

platform. The most time and effort were spent on configuring and generating the security credentials.

Furthermore, if a DTI suddenly stops working, the rest of the DT system can function normally. The DTA will simply stop updating the DTI's related profiles (a profile is a class within the DTA responsible for aggregating and keeping track of data for each heliostat). The IoT Core will raise an error to indicate that the DTI has stopped sending messages, but this has no effect on the performance of the other DTIs.

**Adding or removing a DTA:**

As with the DTI, when a DTA is added to the system of DTs, changes need to be made to the DTI Layer 4, DTA Layer 4 and the cloud platform.

In Layer 4 of the DTI:

- If a DTI must send data to the new DTA, the same reconfiguration must be followed as described for aggregation (point two) in a DTI's Layer 4.
- If a DTA fails, the DTI connected to the DTA will continue to operate normally, and still send the full set of data to the IoT Core.

In Layer 4 of the DTA:

- For communication to a DTA's Layer 5, the same reconfigurations can be applied as described for the DTI's Layer 4.
- For aggregation and communication through the Mosquitto brokers, the CA certificate, private key, and private key certificate must be provided along with the hostname of the broker. The pod numbers of all the heliostats that need to connect must be specified and the desired profile data must be specified (only if the DTA aggregates different data than previous DTAs). The logic to process different data must also be added if applicable.

In the cloud platform:

- A new IoT Core device must be added and provided with the right public key. If applicable, a new registry must be created in the IoT Core and it must be linked to a new or existing Cloud Pub/Sub topic.
- If a DTA fails, IoT Core will log the error and continue to serve other digital twins.

As with the DTI, all the changes mentioned above are configuration file changes. The one exception would be if the new DTA applied different aggregation or processing logic, in which case a source code change would be required.

**Adding or removing a broker:**

To create a new Mosquitto broker, the TLS credentials (the CA certificate, the public key and private key certificates) must be specified in the configuration file. The private key certificate must also contain the IP address and/or domain name of the new Mosquitto broker. In addition, the password file, i.e. the file with allowed usernames and passwords, must be provided in the broker's configuration file.

If the Mosquitto broker fails, it can be configured to automatically start-up again using a batch script. While the broker is down, however, no aggregation will take place. Once the broker is available again, all the clients should reconnect, provided their MQTT clients are configured to do so.

For Pub/Sub, creating a new broker would be to create a new Pub/Sub topic and subscription. This can be done using two lines of Google SDK batch script. The new topic name and subscription name would have to be supplied to the DTIs and DTAs that need to use it. In the case that a Pub/Sub client stops sending to or receiving from a topic, it is unsure whether the messaging will continue automatically. In the experiments where a client did stop sending or receiving messages, the messaging only continued when the DTI or DTA was restarted.