Design Decisions for Architecting Digital Twins of Microservices-based Systems

Aurora Macías[1][0000-0002-1398-8826], Evangelos Ntentos[2][0000-0002-7997-905X], Uwe Zdun[2][0000-0002-6233-2591] and Elena Navarro[1][0000-0001-9496-6890]

[1] LoUISE Research Group, University of Castilla-La Mancha; Albacete, Spain
[2] Research Group Software Architecture, University of Vienna; Vienna, Austria
Aurora.Macias@alu.uclm.es, evangelos.ntentos, uwe.zdun@univie.ac.at and Elena.Navarro@uclm.es

Abstract. Architecting Digital Twins (DTs) that model and monitor microservice-based (MS-based) systems is challenging due to the lack of both design approaches and knowledge among practitioners. To address these shortcomings, we conducted a qualitative study of the patterns and practices applied by practitioners using a Straussian grounded theory-based methodology. After analyzing twenty-three sources of grey literature, we identified three Architectural Design Decisions (ADDs) with twenty-four decision options and their relationships, as well as seventeen key decision drivers based on quality (sub)characteristics from a standard model. We also evaluated the impact of the decision drivers on the ADDs. This resulted in a UML-based ADD model that provides practitioners with guidance for building distributed DT-based systems, including information often omitted or inconclusive in other scientific works on DTs and even on Application Programming Interfaces (API) design. Among the main findings are the intricacies and impact of decisions regarding the granularity of DTs, as well as the seamless integration both among DTs and between DTs and other systems. Additionally, we identified strong similarities between architecting DTs and MS-based systems.

Keywords: Architectural Design Decisions, Digital Twins, Microservices, Software Architecture, Grey Literature, Grounded Theory.

1 Introduction

Digital Twins [1] have emerged as a transformative approach to enhancing the design, operation, and maintenance of complex systems across multiple domains, such as manufacturing, smart cities, or healthcare among others. A Digital Twin (DT) is defined as "a virtual representation of real-world entities and processes, synchronized at a specified frequency and fidelity, that uses real-time and historical data to represent present and past states and simulate predicted future" [2]. DTs generally consist of three components: a physical counterpart (PT), a virtual counterpart, and the integration of both [3][4]. The PT can be either a physical entity, a software entity, or even a software process [3].

The properties of DTs make them ideal candidates for the real-time monitoring and continuous assessment of microservice-based (MS-based) systems [5]. MS-based systems inherently involve distributed components, dynamic interactions and contexts [5][6], and a high degree of complexity [7][8]. These characteristics can hinder their

ability both to provide reliable services [9] and to prevent service failures beyond acceptable limits [5].

There are three aspects to consider when architecting a DT, just as with any other system [4] [3] [10]:

- *Granularity* refers to the degree of abstraction or detail in the design [11], considering not only the functional requirements but also the non-functional ones [12]. The granularity level determines the boundaries, size, and complexity of the software components of the MS-based system to be twined, and consequently, of the DTs that will be developed based on control needs. Thus, granularity is key aspect in the development of DTs [25].
- An *Interface* defines an interaction point between two components or systems and specifies how they communicate with each other, including the methods, protocols, and data formats used. Interfaces are used when architecting DTs to facilitate their control, governance, and orchestration or for simple querying the DTs [3].
- *Integration* refers to the combination, coordination, and collaboration of different components to enable their seamless interaction. Different DTs can be integrated into a composite DT, so that both composite and individual DTs can be observed and controlled to meet specific requirements. DTs can also be integrated with other convenient services or systems for simulation, analysis, and other purposes, as needed to achieve specific business goals [3].

Implementing DT architectures is a challenging task [3][4], especially when the twinned element is a complex MS-based system [5]. Thus, architecting DTs of MS-based systems considering the three architectural aspects stated is both critical and non-trivial. Although various architectural patterns and practices for architecting DTs have been identified [3][4][7][8], they neither explicitly address these architectural aspects nor are they well documented or formalized. Moreover, there is limited research on their interrelations and the factors driving and influencing their selection. Practitioners often lack both a systematic understanding of how to apply these concepts and a comprehensive framework to guide their decision-making [3][13][14]. This knowledge gap challenges them in making informed architectural design decisions (ADDs) when designing robust and scalable DT architectures. This leads to suboptimal implementations that may not meet quality requirements such as compatibility, maintainability, or performance efficiency [15]. These issues also affect the design of DTs of MS-based architectures that should meet the properties expected from DT [23]. Considering this, this work aims to answer the following Research Questions (RQs):

- **RQ1**. Which ADDs and corresponding patterns and practices are used by practitioners while architecting DTs of MS-based systems with respect to the aspects of granularity, interfaces, and integration?
- **RQ2**. What are the relations between the identified patterns and practices?
- **RQ3**. Which decision drivers are relevant to the identified patterns and practices and in what degree?

To answer these RQs, we conducted a qualitative Grey Literature Study (GLS) [16][17] based on Straussian Grounded Theory (GT) [18][19][20][21], with the aim of formalizing the current understanding and architectural concepts of practitioners when architecting DTs of MS-based systems. The study focused on three key aspects: granularity, interfaces, and integration. Other aspects of DT design were excluded due to space constraints. We analysed 23 knowledge sources to identify established industrial patterns and practices. Based on our findings, we developed a formal ADD model comprising 3 decisions, 24 decision options, and 17 decision drivers.

This paper is organized as follows: Section II compares this work with related studies. Section III presents the research methods applied in this study and summarizes the knowledge sources. Section IV details the resulting reusable ADD model for architecting DTs of MS-based systems, focusing on the aspects of granularity, interfaces, and integration. Section V discusses the results. Subsequently, Section VI considers potential threats to the validity of this study, and Section VII summarizes the findings and outlines directions for future work.

2 Related Work

Few studies focusing on patterns and practices for architecting DTs can be found. Furthermore, existing research often lacks detail about which patterns or practices are used for architecting DTs with respect to granularity, interfaces, and integrations. For instance, Ferko et al. [15] presented a systematic mapping study on architecting DTs, capturing architectural solutions including reference architectures and architectural styles, as well as quality attributes based on those proposed by the ISO 25010 standard. They examined the correlations between those architectural styles and quality attributes. This work explores architecting DTs from a high-level perspective and does not consider more specific architectural patterns or practices.

Some studies [22][13][14] review academic works analysing key enabling technologies for DTs. They identify the use of communication protocols and APIs as interfaces for enabling the integration among DTs and other systems as well as their interactions. Besides defining a high-level architecture for DTs or their execution platform, they do not provide information about more specific architectural patterns or practices. Furthermore, none of these works provide precise information about common decision drivers. A similar work [3] posing similar issues, identifies the key properties that a DT should satisfy and thoroughly defines them. It also identifies different patterns and practices related to the interaction that mainly include interfaces defined based on well-formed APIs and structured data, meeting the key technical features and properties of DTs. This work does not consider standard quality attributes as decision drivers.

Although different studies have explored the definition of patterns for architecting DTs, they do not detail how these patterns support the DT properties or do not explore the correlation between the patterns and standard quality attributes. Some of those works, such as [23], document the use of patterns for DTs, such as digital shadow. Other works propose using well-known design patterns. For example, [24] documents the use of the façade pattern for integration. Some proposals, however, identify patterns that could meet the DT properties [3] and propose the definition of interfaces such as APIs for management and data querying. For instance, [10] proposes design patterns mostly popularized in the field of microservices, such as ambassador, sidecar, or adapter, to balance complexity and manage costs. This work assumes that DTs are monolithic services or decomposed in multiple components, which are deployed as a distributed network of containerized entities. Another work [4] proposed a domain-independent approach inspired by Domain-Driven Design (DDD), suitable for architecting DTs [25][S3]. It includes the use of various patterns, such as open host service (OHS), messaging, or integration services, and discusses how they support the DT properties.

Recent works address the design of DTs of complex systems with MS architectures that are cloud-native or deployed on Kubernetes-based platforms. For example, Borsatti et al. [7] define a message queue pattern for the interaction among DTs as well as between DTs and other systems, exploiting APIs as an alternative. This work does not study how these practices or patterns may satisfy any DT property, although the DT should present functional correctness. A message queue pattern is also used in [8], considering accuracy, precision, and performance as the main driving factors of the design. In both works, the impact of those driving factors is not formally specified.

Our study addresses the gap identified by systematically examining practitioners' methods and techniques, resulting in a formal model that includes ADDs, patterns and practices, decision drivers based on a quality standard, and their relationships. This model offers valuable insights to help practitioners effectively architect DTs of MS-based systems. To the best of our knowledge, this is the first study of its kind in this area.

3 Research Method

We conducted a systematic GLS of practitioners' sources such as blog posts, system documentation and practitioners' reports [17][16][26], to obtain information about practitioners' views on architecting DTs of MS-based systems in terms of the granularity of DTs and their components, their interfaces, and integrations among the DTs and of DTs with other convenient services or systems. GLS combined with Straussian GT was successfully applied in previous studies [27][28].

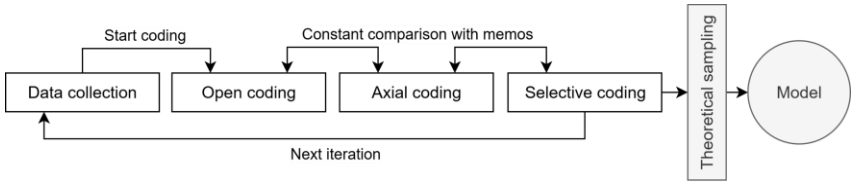

Fig. 1. Research method steps.

The research method consists of the steps shown in Fig. 1. As part of *data collection*, following Kitchenham's guidelines [29], the search terms and string were defined. The initial search string was <"digital twin*" AND microservice*>. We added alternative terms to the initial search string in subsequent searches, as done in <"digital twin*" AND microservice* AND granularity>, based on concepts included in Section I and other related concepts found during the analysis. The search for practitioner sources was conducted using search engines such as Google, Bing, Yahoo, DuckDuckGo, and StartPage, since they keep around 95% of the market share, as well as widely used technology topic portals such as DZone and InfoQ, among others. The practitioners' articles found were carefully read by the authors to determine their eligibility based on the criteria for evaluation and critical appraisal of grey literature defined by [30], namely authority, accuracy, coverage, objectivity, date, and significance. For instance, articles including patterns and practices for DTs of MS-based systems had to be at least recent (date), relevant to the topic (significance) and not marketing a business or product (objectivity) to be considered candidate sources. All authors reviewed and approved other author's selection of sources. Knowledge sources are summarised in **Table 1** (the link to a copy of each source in a web archive is included in the replication package, which is publicly accessible at https://zenodo.org/uploads/15514330). Then, GT coding practices and constant comparison were repeatedly and iteratively applied to identify concepts, categories, drivers, and relations. The identified entities were modelled using a diagramming tool. The types of coding activity proposed by GT [21] were conducted. *Open coding* was used to translate conceptual details into labels. *Axial coding* was applied to group recurring terms, synonymous, and related concepts into categories. Each source was examined line by line, with thought processes, interpretations, and argumentation documented in memos for traceability. Through *selective coding* concepts were refined, and the theory's core ideas were extracted. The final theory was presented as a formal UML-like model already used in previous studies [27][28]. The process was iterated until *theoretical saturation* [31] was achieved, that is, when five to seven additional sources no longer yielded any new insights.

Table 1. Knowledge sources included in the study.

ID	Title	Reference
S1	The Role of Digital Twins in Unlocking the Cloud's Potential	https://tinyurl.com/2r4epny8
S2	Webinar: Putting the 'I' in IoT – building digital twins with Akka microservices	https://tinyurl.com/3mneknnu
S3	Digital Twins: A digital counterpart for a new concept of Product	https://tinyurl.com/yc85jcfj
S4	Microservice Twins in Kubernetes - breaking new grounds	https://tinyurl.com/yeyj7837
S5	How Kubernetes supports digital twins	https://tinyurl.com/332u36jh
S6	Building Digital Twins of Software Systems with Generative AI: A Practical Approach	https://tinyurl.com/y35j89f7
S7	Kafka at the Core of any Digital Twins Architecture	https://tinyurl.com/55c9898b
S8	Digital twins to virtualise the world	https://tinyurl.com/4kfnxfu9
S9	Digital Twins Event Gateway	https://tinyurl.com/993ra83k
S10	Using Digital Twins	https://tinyurl.com/b44c8xsh
S11	Challenges of Creating Digital Twins in the Transition to Industry 4.0	https://tinyurl.com/5rj2ru8z
S12	IoT Architectures for Digital Twin with Apache Kafka	https://tinyurl.com/33ppzvvf
S13	Apache Kafka as Digital Twin in Industrial IoT	https://tinyurl.com/dvu8efuc
S14	Building the digital representation with digital twin using AWS stack	https://tinyurl.com/y2pr38mn
S15	Building the digital representation with Digital Twin using Microsoft stack	https://tinyurl.com/mr4837pr
S16	HiveMQ and Eclipse Ditto: Friends or Foes?	https://tinyurl.com/4xrh4f7c
S17	Understanding digital twin environments	https://tinyurl.com/mu7yf29z
S18	Hands-on Guide to Using MQTT and Eclipse Ditto for Digital Twins	https://tinyurl.com/uwekb2cy
S19	Digital Twin Platform Stack Architectural Framework	https://tinyurl.com/48zbdj8j
S20	Building Digital Twins for DFDS With Cross plane and Kubernetes	https://tinyurl.com/5bz52kuk
S21	How Digital Twins and IoT Work Together [With Example]	https://tinyurl.com/4d8f987r
S22	A Comprehensive Guide on Digital Twins	https://tinyurl.com/3z7bjfdn
S23	Building a Digital Twin with MQTT and Hasura Streaming Subscriptions	https://tinyurl.com/yh7wdch5

4 ADD Model for Architecting Digital Twins of MS-Based Systems

The ADD model derived includes practitioners' views, patterns, and practices extracted from GL related to architecting DT(s) of MS-based systems. The ADD model was specified as an instance of the metamodel shown in Fig. 2, defined in [28] and used in previous studies [27][28].

According to this metamodel, a Category groups Decisions that are closely related to a specific aspect. A Decision also has a Context: a domain object that denotes the system part or aspect for which the decision is applied. An Option can be a Pattern or a Practice as a possible Solution to the design problem expressed and modelled as a set of design Decisions. An Option has influencing factors driving the decisions called Forces, which can have a force impact. Forces identified in this work are expressed as quality (sub)characteristics of the software product quality model ISO/IEC 25010 [32]. An

Option can be related to another Option using a Relation. Possible Relations are ‘is-a’, ‘uses’, ‘can be combined with’, ‘realizes’, etc. All Options must be linked directly or via other Options to a Decision. Decisions and Options can have ‘next’ decision Relations as well, and they can be ‘mandatory’ or ‘optional’.

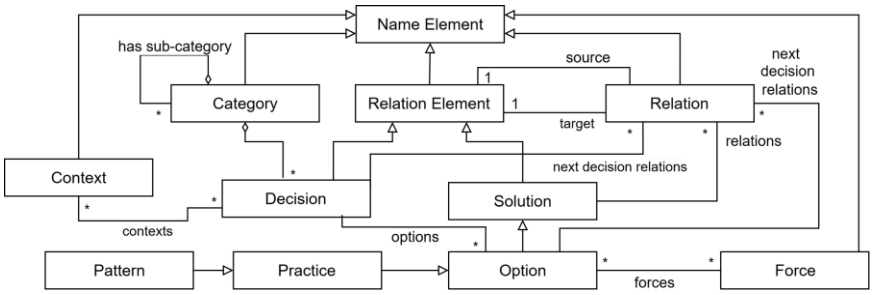

Fig. 2. Meta-model for ADD model.

As Fig. 3 depicts, the ADD model presented in this work consists of a single decision Category: *Architecting DTs of an MS-based system.* This has three top-level Decisions related to the architecting aspects of DTs considered in this study (see Section I). Some secondary decisions have been excluded due to space restrictions. Although all the decisions of the ADD model represent a crucial aspect, the last one has an optional character depending on whether the DT is composite and whether it interacts with other systems and services. For each mandatory Decision, one of the modelled Options should be chosen.

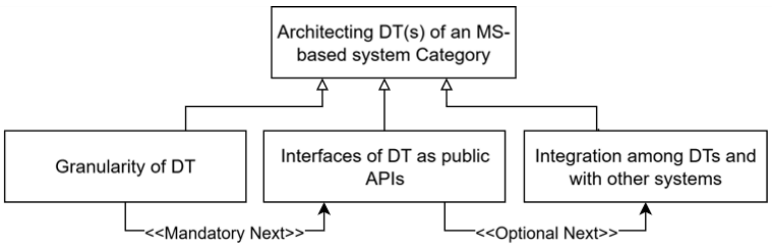

Fig. 3. ADD model on architecting DTs of MS-based systems: overview.

4.1 ADD: Granularity of DT

When designing DTs of MS-based systems, it is essential to determine the granularity level: the unit of operational control structure that the DT will represent. This decision plays a critical role in supporting stakeholders, such as engineers and operators [S1, S3, S9, S7, S10], in monitoring system performance and detecting issues or anomalies [33]. It is heavily influenced by the specific goals and requirements of these stakeholders. Granularity impacts on composability and interactions, as the number of DTs representing the MS-based system(s) and their integrations will vary accordingly [S2]. Four main options for the granularity of DTs have been identified (see Fig. 4).

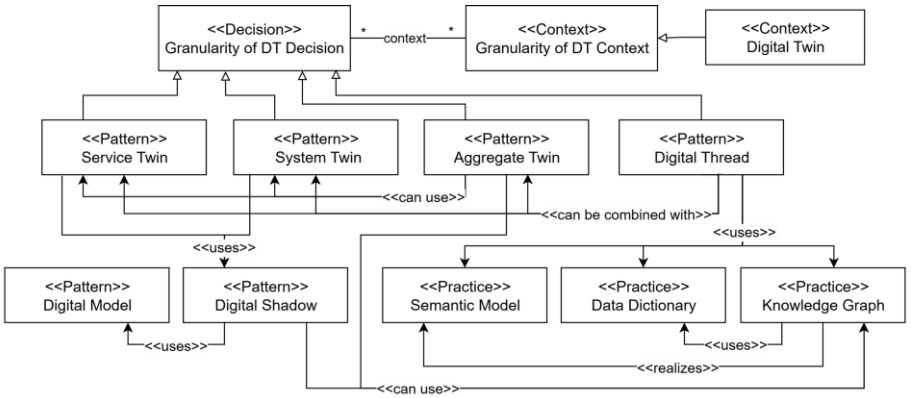

Fig. 4. Granularity of DT decision (the forces have been extracted to **Table 2**).

It is straightforward to use the *System Twin* pattern [S7, S3, S9, S10] to represent a (sub)system composed of interconnected components or services. This pattern can meet the operational requirements when monitoring and modelling a traditional monolithic or a simple MS-based system. Therefore, the degree of *adaptability* provided is **medium**. This pattern offers a **low** level of *modularity*, *reusability*, and *modifiability* due to the high level of combination of system components in the DT, which implies that changes to one component may affect the entire DT of a system. It also offers a **medium** degree of *resource utilization* and *time behaviour,* since the performance of the DT is efficient only at the (sub)system-level when it is simple, and a **low** degree of *scalability*, as handling variable workloads for the entire DT of a (sub)system can be complex.

Another option is the *Service Twin* pattern [S1, S4, S7, S3, S9, S10, S13] when a DT twins a component or service of a (sub)system. Since services are the primary units of MS-based systems, this pattern helps stakeholders to accomplish their tasks at the service level. It is suitable when interactions among different services do not need to be represented. This pattern requires designing multiple DTs for modelling an entire MS-based system, as it follows a one-to-one mapping between an MS and its corresponding DT. The Service Twin pattern provides a **medium** degree of *adaptability*, and, since DTs of individual services can be modified independently, a **very high** level of *modularity*, *reusability*, and *modifiability*. It also offers a **high** degree of *resource utilization*, *time behaviour*, and *scalability*.

When either the System Twin or the Service Twin are applied, then, the *Digital Shadow* (DS) pattern [S3, S10, S12] can be used. This pattern creates an up-to-date and accurate virtual replica, including properties and state, of an MS or MS-based system, depending on the main pattern that uses it. It involves one-way data flow from the twinned system to the DS to monitor its changes without commanding it. The DS can use multiple Digital Models (DMs), especially when used along with the System Twin, typically one per component or service of the system. Different types of DMs can be used to reflect different perspectives of the system (e.g., infrastructure, software, etc.) [S3, S7, S10], **positively** impacting the degree of *modularity* and *reusability*, and increasing the representation fidelity and the degree of *functional correctness*.

The third option is the *Aggregate Twin* pattern [S7, S3, S9]. It represents a composition or aggregation of DTs designed using the system or service twin patterns. It is used to interconnect DTs of components or subsystems of a complex system, including its IT infrastructure [S3]. It is also used to aggregate DTs of several independent (sub-)systems that are interconnected as part of a supply chain or broader organizational process. This pattern implies high complexity and offers a **high** degree of *modularity*, and a **medium** degree **of** *reusability*, *modifiability,* and *adaptability,* since modifications of a DT of the

aggregation may have widespread effects. It also offers a **medium** degree of *resource utilization* and *time behaviour* as aggregating DTs of multiple systems can lead to resource overhead and increase latency, and a **high** degree of *scalability*, as it has been designed for complex, interconnected systems, although scaling DTs of multiple systems simultaneously can be challenging.

The last option is the *Digital Thread* pattern [S12, S14, S15, S20][33], which offers the highest level of granularity and complexity. It represents and enables the monitoring of the lifecycle, including design, of a service or system modelled by one or more DTs based on any of the previous patterns. This pattern interconnects the DTs of all relevant components or services, systems, and functional processes as appropriate to enable efficient collaboration and decision-making across all the stages of a business process and to optimize traceability. It allows the seamless modification of the DTs network as the systems evolve and can connect to (almost) any other enterprise system. The degree of *adaptability* provided is therefore **very high,** as it supports the ability of the DT(s) to evolve alongside the system(s) represented across different stages. This pattern offers a **high** degree of *modularity* and *reusability*, since it spans across the entire lifecycle and multiple systems. Additionally, it provides a **high** degree of *modifiability,* as it allows for continuous updates across the entire lifecycle. It also offers a **low** degree of *resource utilization* and *time behaviour,* due to the high complexity and data volume, and a **very high** degree of *scalability*, as it supports scaling across different stages and systems.

Practices like Semantic model [S14, S15, S16], Data dictionary [S14, S15, S16] and Knowledge graph [S14, S15, S16] are used to contextualize data, mainly by the Digital Thread pattern. A *Knowledge Graph* [34] represents a collection of interlinked descriptions of objects, events or concepts, using semantic metadata. Knowledge Graphs can also be used by the DS and the Aggregate Twin patterns to link DMs and DTs, respectively [S15]. In general, the use of the patterns DS and DM and the practices Semantic Model, Data Dictionary and Knowledge Graph **positively** impact the degree of *modularity* and *reusability,* but it also increases the complexity and **negatively** impacts the level of *modifiability*. The degrees of *resource utilization* and *time behavior* also suffer a **negative** impact.

Table 2. Forces for the Architectural and representation granularity of DT decision (extracted from the ADD model).

	Option	Forces (Degree/Impact)
Main	Service Twin	ADAPT (M), MODUL (M), REUSA (M), MODIF (H), SCALA (M), RUTIL (H), TBEHA (H)
	System Twin	ADAPT (M), MODUL (L), REUSA (L), MODIF (H), SCALA (L), RUTIL (VH), TBEHA (VH)
	Aggregate Twin	ADAPT (M), MODUL (H), REUSA (H), MODIF (M), SCALA (H), RUTIL (M), TBEHA (M)
	Digital Thead	ADAPT (VH), MODUL (VH), REUSA (VH), MODIF (L), SCALA (VH), RUTIL (L), TBEHA (L)
Additional	Digital Shadow	MODUL (+), FCORR (+), REUSA (+), MODIF (-), TBEHA (-), RUTIL (-)
	Digital Model	MODUL (++), FCORR (++), REUSA (++), MODIF (- -), TBEHA (- -), RUTIL (-)
	Knowledge graph	MODUL (+), REUSA (+), MODIF (-), TBEHA (-), RUTIL (-)

Forces Codes: **FCORR**: Functional correctness, **FAPPR**: Functional appropriateness, **TBEHA**: Time behavior, **RUTIL**: Resources utilization, **OPERA**: Operability, **MODUL**: Modularity, **REUSA**: Reusability, **MODIF**: Modifiability, **ADAPT**: Adaptability, **SCALA**: Scalability.
Degree scale *(main options)*: **L**. Low, **M**. Medium, **H**. High, **VH**. Very High. **Impact scale** *(additional options)*: **+**: positive, **-**: negative.

Table 2 presents the forces identified for the different options of this ADD, extracted from the model for clarity. Each force's degree or impact value on a given option is shown in parenthesis. Values are based on authors' interpretation of the GL sources and of the well-known descriptions of patterns and practices. The legend explains the abbreviations and scales used. The force degree of the main options is aligned with the definitions provided by the quality standard and is rated from Low (negative) to Very High (positive) to facilitate decision-making, while the impact of forces of the additional options is represented by plus (+) or minus (-) signs depending on whether their impact on the main option(s) is positive or negative. An extended version of this table and the tables for the other decisions can be found in the replication package.

4.2 ADD: Integrations of DTs and between DTs and other Systems

The decision related to interfaces involves six options for exposing DT functions and/or data (Fig. 5), based on the endpoints design and influenced by common API design bests practices expressed as standard quality aspects.

The first option identified is the Action-oriented API pattern [S2, S4, S5, S6]. Endpoints are designed to expose and perform specific complex custom actions or operations, often encapsulated into single API calls made using verbs. The degree of *modularity* provided is **low**, since these APIs are often tightly coupled to specific actions. They offer a **low** degree of *adaptability*, as they are designed for specific tasks. They often have clear, descriptive endpoints, so they provide a **high** degree of *self-descriptiveness*. The degree of *installability* offered is **medium**, as they may require specific configurations. They provide a **high** degree of *operability* due to their straightforward nature. The degree of *interoperability* offered is **medium**, depending on how well they integrate with other systems.

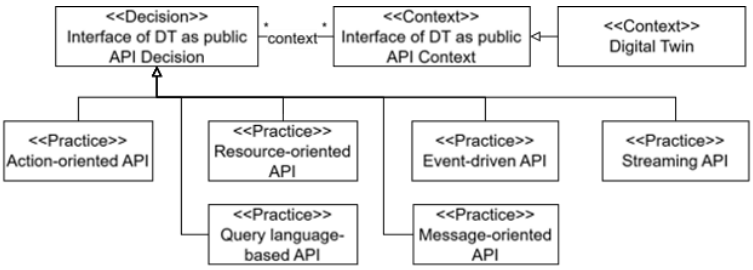

Fig. 5. Interfaces of DT as public APIs decision.

Another common option is the use of a Resource-oriented API pattern [S2, S4, S5, S6]. Endpoints are designed for exposing resources and support standard CRUD operations over resources mapped to standard HTTP methods to manipulate them. Endpoints are typically invoked using resource names. The degree of *modularity* provided is **high**, since these APIs are often loosely coupled to specific actions (they are resource-centric). They offer a **high** degree of *adaptability*, as they can be extended to new resources. They follow widely understood principles, so they provide a **high** degree of *self-descriptiveness*. The degree of *installability* offered is **high**, due to their standardized nature. They provide a **high** degree of *operability* as they are intuitive and easy to use. The degree of *interoperability* offered is **high**, due to adherence to API standards.

Another option is the *Query language-based API* [S2, S4, S5, S6] pattern. It involves endpoints designed to be invoked by sending queries written in a flexible query language that allows clients to specify exactly what data they need and how it should be formatted. The degree of *modularity* provided is **high**, since these APIs allow fine-grained data

access. They offer a **high** degree of *adaptability*, as queries can be modified as needed without changing the API. Understanding the query language is required, so they provide a **medium** degree of *self-descriptiveness*. The degree of *installability* offered is **medium**, as they may require additional setup for the query language. They provide a **high** degree of *operability* since they offer flexibility for data retrieval. The degree of *interoperability* provided is **high** when the query language is widely supported.

A variety of options that reduce the need for clients to repeatedly poll for updates, typically used in event-driven architectures, have been identified. For example, *Event-based API*s [S2, S4, S5, S6] are designed to handle events, triggering actions in response to specific occurrences. Notifications to subscribers are usually pushed from the server to the client, using callbacks or other push mechanisms. Another option is the use of the *Message-oriented API* [S2, S4, S5, S6] pattern, involving an API designed to exchange messages between producers and consumers to facilitate the communication between different systems or components. An additional option is the use of the *Streaming API* [S2, S4, S5, S6] pattern, in which the API is designed to provide a continuous data flow from a source to a subscriber client, that consumes and processes the data, often in real-time. These three types share the same driving forces degrees. The degree of *modularity* provided is **high**, since these APIs are decoupled from the main application logic. They offer a **high** degree of *adaptability*, as new events, types of messages or data streams can be added. Understanding event triggers and messaging and streaming protocols is necessary, so they provide a **medium** degree of *self-descriptiveness*. The degree of *installability* offered is **medium**, as they may require event handling, messaging, and streaming infrastructure like event bus, (message) broker, and streaming data pipeline, respectively. They provide a **high** degree of *operability* since they offer real-time responsiveness. The degree of *interoperability* provided is **high** when they follow standard event, messaging and streaming protocols, respectively.

4.3 ADD: Integrations of DTs and between DTs and other Systems

An important aspect of architecting DTs, or any other software system, is to identify and model the required interactions among components [4] [S16], bearing in mind that they do not always share the same interface [S12] [4] (see Section 4.B). Eight main decision options have been identified to support the integration of DTs and other systems for their seamless interaction (see Fig. 6).

The straightforward option is the *Direct communication* pattern [S12, S14, S17]: a DT acts as a client that can make requests directly to the public endpoint of other DTs, systems, or services, that act as servers (clients can act servers at the same time). Since it lacks built-in *fault tolerance* mechanisms, the degree of this quality attribute is **low**. This option involves a high degree of coupling and, therefore, a **low** degree of *modularity* and *modifiability*, which complicates the addition and handling of mandatory cross-cutting security concerns, resulting, for example, in a **low** degree of *authenticity* of the services involved, since there is no mechanism to verify the identity of the interacting entities. All the interacting DTs, systems, and services must be exposed to the "external world", which increases the attack surface compared to that offered when internal components are hidden. For that reason, the degrees of *confidentiality* and *integrity* are **low**. This pattern offers a **high** degree of *resources utilization* and *time behaviour*. However, the operation of complex composite DTs might require several calls to other DTs, services, or systems, which may result in multiple network round trips among them, adding significant latency. This option could be good enough for DTs twining small MS-based systems but not for large and complex MS-based systems. Therefore, the degree of *scalability* and *adaptability* is **low**.

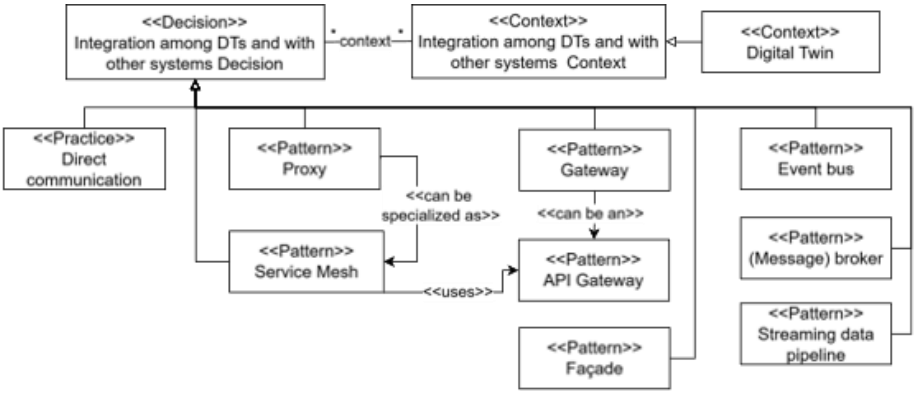

Fig. 6. Integration of DTs and between DTs and other systems.

A common option for integration is the *Proxy* pattern [S11, S12, S17]. It involves the use of an intermediary object between a client component of the DT and an invoked or server component that share the same protocol. It controls the access by executing additional functionality related to Load balancing, Catching, or Security before forwarding the request with small overhead. Similarly to Direct Communication pattern, the degree of *adaptability, modularity* and *modifiability* offered is **medium**, although the degree of *fault tolerance, authenticity*, *confidentiality*, and *integrity* is **high**. This pattern offers also a **medium** degree of *resources utilization*, *time behaviour*, and *scalability*.

Another option, based on proxies, is the *Service mesh* pattern [S1, S3, S4], commonly used as an infrastructure layer to support the interaction among service-based DTs and other services, and when numerous cross-cutting concerns such as traffic management, security, and observability, need to be (dynamically) implemented. The Service mesh pattern offers a **high** degree of *adaptability, modularity* and *modifiability*, *fault tolerance, authenticity*, *confidentiality*, *integrity*, *time behaviour*, and *scalability*, although the degree of *resource utilization* is **medium**.

A well-known option is the use of the *Gateway* pattern [S4, S5, S8, S16, S12, S13, S14], which constitutes a single-point-of-entry and encapsulates access to external services, systems or resources. A common type of Gateway identified is *API gateway* [S18, S13, S14] that provides application-level functionalities to manage API requests such as routing, rate limiting, and security. This pattern also offers a **high** degree of *adaptability, modularity*, *modifiability*, *fault tolerance, authenticity*, *confidentiality*, *integrity*, *time behaviour*, and *scalability*, but a **medium** degree of *resource utilization*. API gateway aspects are used by the Service Mesh pattern.

An additional option identified is the *Façade* pattern [S17, S13, S14] that provides a simplified interface to a complex subsystem although introducing some overhead. It is commonly used to provide a unified point of access to legacy systems [S16]. This pattern offers a **high** degree of *adaptability, modularity* and *modifiability*, *fault tolerance, time behaviour*, and *scalability*, but a **medium** degree of *resource utilization*. This pattern enforces security aspects, however, since it relies on the underlying subsystems for that, it offers a **medium** degree of *authenticity*, *confidentiality*, *integrity*.

A group of options used in even-driven architectures have been also identified. The *Event-bus* pattern [S3, S4, S8, S16, S13, S14, S15] facilitates asynchronous, decoupled communication between components when a change in a component state triggers an event that is consumed by other component(s). Another well-known option is the *Broker* pattern [S3, S4, S8, S16, S13, S14, S15] that handles the delivery of messages among components. It often enables queueing, topic-based routing, or persistence. The last

option is the *Streaming data pipeline* pattern [S3, S4, S8, S16, S13, S14, S15], used to continuously process incoming data from a DT in real time. It may enrich, transform, filter, and feed such data into analytics or storage systems. All these three patterns offer a **high** degree of the ten quality aspects previously identified.

Common combinations of some of the main options identified and with additional patterns are not documented in this work due to space constraints.

5 Discussion

The ADD model presented to support the design of DTs of (complex) MS-based systems is closely aligned with the selected sources and has been systematically developed. The point of theoretical saturation in GT [32] was reached after integrating information from 23 sources. While reading the first 10 sources, frequent adjustments to the identified codes and categories were necessary. However, in the subsequent 8 sources, such modifications became less frequent, and no further alterations were required for the remaining sources. In the following, the main results of each RQ are discussed.

RQ1: The 24 patterns and practices identified as decision options were modelled in association with 3 different ADDs, as we observed that they could be grouped based on the aspects defined in Section I. The resulting models and their descriptions help clarify and define, from a practical perspective, these three aspects paramount for the design of DTs [4]. It is worth noting that that the decision options related to the second ADD provide significantly more detail than descriptions found in existing works on DT design [3][4] and on API design [35], which primarily refer to public APIs with either an action-oriented or a data-oriented role. Our work documents a total of 6 options, focusing on the endpoint design and highlighting their invocation methods, thereby enabling more informed design decisions. Another remarkable finding is that some of the identified decision options associated with the third ADD correspond to well-known and documented patterns, such as the integration-related patterns API Gateway or Service Mesh. Some of those patterns are also commonly used in the design of MS architectures. The design of both DTs and MS-based architectures share some challenges, such as defining boundaries (first ADD) and communication across them (second and third ADD) [35]. These aspects are also addressed as part of the strategic design activity defined by the DDD-inspired approach for architecting DTs [4] (see Section II).

RQ2: The study reveals relations between ADDs and decision options that are generally clear. Although additional relations have not been specified in the diagrams, due to the modular structure of the ADD model, most of them have been mentioned in the text or can be derived from the descriptions provided. For instance, potential correlations may exist between certain API types outlined in the second ADD and the corresponding integration mechanisms proposed in the third ADD. Another noteworthy aspect is that the third ADD, although modelled as an "optional next" decision, plays a crucial role in the success of DTs of complex MS-based systems, particularly when DTs must interact among them and with other systems, and when business-oriented stakeholders are expected to benefit from DT adoption through the Digital Thread pattern.

RQ3: The research identified a total of 17 forces, including sub-characteristics of maintainability and flexibility, that are common to most of the ADDs and decision options. The forces identified correspond to standard quality characteristics and their impact factors are expressed using well-defined rating scales. This allows practitioners to easily select the most convenient options based on specific requirements. It is worth

noting that the first ADD, related to granularity, and the third ADD, related to integration, are the decisions influenced by the highest number of forces: 10. The ADDs are mainly driven by maintainability and flexibility aspects, and the rest of forces appear to play a complementary role. Furthermore, the third ADD is the only decision driven by forces related to security and reliability, which highlights its importance in DT design despite its optional nature.

6 Threats to Validity

In the following, the threats to validity identified by Wohlin et al [36] are discussed.

Construct validity ensures that the measurements accurately reflect what is intended. To mitigate potential biases, the research was not limited to specific sources. Instead, it followed well-defined inclusion and exclusion criteria, pulling from a wide range of relevant materials. Despite searches were exhaustively conducted, some sources may have been unintentionally overlooked.

Internal validity deals with factors that impact causality. To mitigate this threat, the study used independently created practitioner reports, ensuring objectivity but potentially missing insights from interviews. To address this, we reviewed more sources than necessary to reach theoretical saturation, reducing the risk of missing critical perspectives.

External validity concerns the applicability of findings beyond this work. While the inclusion of diverse sources supports generalization to DT architectures for MS-based systems, the exclusive use of grey literature, despite its practical value, may introduce bias due to the lack of peer review. Although strict selection criteria and data triangulation were applied, this limitation may affect the model's reliability. Additionally, focusing on three architectural aspects (granularity, interfaces, and integration) enabled depth but may restrict applicability to other concerns. Future work should explore broader architectural dimensions.

Conclusion validity ensures that the relationships identified between treatments and outcomes are sound. The sources were assessed based on objectivity, significance, and other relevant criteria, and were reviewed by all the authors prior to their consensual inclusion. Theoretical saturation was reached, and additional sources were even analysed beyond that threshold to be extra cautious. Since GT focuses on real-world phenomena, the main risk is misinterpreting concepts. Extensive sourcing minimizes that risk, but findings are intended the defined scope.

7 Conclusions and Future Work

This work presents a GL study using Straussian GT to develop an ADD model for patterns and practices adopted by practitioners when architecting DT(s) for (complex) MS-based systems. These DTs are designed to continuously and automatically monitor, model, and enhance its operation. The model encompasses 3 ADDs, 24 decision options, 18 relations, and 17 driving forces defined by an international standard on software quality. The impact of these forces is represented using well-defined rating scales, allowing practitioners to easily choose the most convenient options based on specific requirements. The work reveals that the main ADDs are defined around the architectural aspects of granularity of DTs and integration, including the use of interfaces of DTs as public APIs. It provides information on patterns and practices that support these aspects, often omitted or inconclusive in other scientific works on DTs and even on API design. These findings contribute to a clearer and more practical understanding of the foundational concepts that underpin DT architecture. They emphasize the unique complexity of

determining DT granularity (a decision specific to DTs) and highlight the critical importance of integration, both among DTs and between DTs and external systems. Although formally categorized as optional, integration decision has proven essential for the effective deployment of DTs of complex microservice-based systems. Although the findings are based on a specific group of practitioners, they align with widely accepted standards, suggesting potential applicability to other domains. However, further validation in diverse industrial settings is needed to confirm the model's broader relevance.

As future work, we plan to explore the use of DDD design principles in the design of DT architectures, considering the similarities found, as well as patterns mentioned in scientific papers that could be applied to DTs design despite not being identified in practitioners' sources.

Acknowledgments. This paper has been partially funded by the 2024 predoctoral mobility grant for research stays at international universities and research institutions for PhD students from the University of Castilla-La Mancha, and two Erasmus+ grants for short-term doctoral stays 2025 (reference numbers 2024-1-ES01-KA131-HED-000206822-MOB-0053 and 2023-1-ES01-KA131-HED-000113966-MOB-0733), all of them awarded to Aurora Macías. It is also part of the R+D+i project PID2022-140907OB-I00 funded by MICIU/AEI /10.13039/501100011033 and ERDF, EU. It has also been partially supported by Junta de Comunidades de Castilla-La Mancha/ERDF (SBPLY/21/180225/000020) and by the University of Castilla-La Mancha (2023-GRIN-34436). Evangelos Ntentos' and Uwe Zdun's work was supported by the FFG (Austrian Research Promotion Agency) project MODIS (no.\ FO999895431).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

[1] M. Grieves and J. Vickers, "Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems," in Transdisciplinary Perspectives on Complex Systems: New Findings and Approaches, S. and A. A. Kahlen Franz-Josef and Flumerfelt, Ed., Springer, 2017, pp. 85–113. doi: 10.1007/978-3-319-38756-7_4.

[2] John Turner, Richard Ferris, and Salla Eckhardt, "Infrastructure Digital Twin Maturity: A Model for Measuring Progress," 2021. Accessed: Dec. 16, 2023. [Online]. Available: https://www.digitaltwinconsortium.org/wp-content/uploads/sites/3/2022/06/Infrastructure-Digital-Twin-Maturity-Model.pdf

[3] R. Minerva, G. M. Lee, and N. Crespi, "Digital Twin in the IoT Context: A Survey on Technical Features, Scenarios, and Architectural Models," Proceedings of the IEEE, vol. 108, no. 10, pp. 1785–1824, 2020, doi: 10.1109/JPROC.2020.2998530.

[4] A. Macías, E. Navarro, C. E. Cuesta, and U. Zdun, "Architecting Digital Twins Using a Domain-Driven Design-Based Approach," in IEEE 20th International Conference on Software Architecture (ICSA), 2023, pp. 153–163. doi: 10.1109/ICSA56044.2023.00022.

[5] V. Grassi, R. Mirandola, and D. Perez-Palacin, "A conceptual and architectural characterization of antifragile systems," Journal of Systems and Software, vol. 213, p. 112051, 2024, doi: https://doi.org/10.1016/j.jss.2024.112051.

[6] M. and J. A. and R. B. and T. C. and van H. A. Avritzer Alberto and Camilli, "Continuous Dependability Assessment of Microservice Systems," in Software Architecture. ECSA 2022 Tracks and Workshops, T. and R. C. and M. H. Batista Thais and Bureš, Ed., Cham: Springer International Publishing, 2023, pp. 138–147.

[7] D. Borsatti et al., “KubeTwin: A Digital Twin Framework for Kubernetes Deployments at Scale,” IEEE Transactions on Network and Service Management, vol. 21, no. 4, pp. 3889–3903, 2024, doi: 10.1109/TNSM.2024.3405175.

[8] A. Bhardwaj and T. A. Benson, “KubeKlone: A Digital Twin for Simulating Edge and Cloud Microservices,” in Proceedings of the 6th Asia-Pacific Workshop on Networking, in APNet ’22. New York, NY, USA: Association for Computing Machinery, 2023, pp. 29–35. doi: 10.1145/3542637.3542642.

[9] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing,” IEEE Trans Dependable Secure Comput, vol. 1, no. 1, pp. 11–33, 2004, doi: 10.1109/TDSC.2004.2.

[10] P. Bellavista, N. Bicocchi, M. Fogli, C. Giannelli, M. Mamei, and M. Picone, “Requirements and design patterns for adaptive, autonomous, and context-aware digital twins in industry 4.0 digital factories,” Comput Ind, vol. 149, p. 103918, 2023, doi: https://doi.org/10.1016/j.compind.2023.103918.

[11] T. Spijkman, S. Molenaar, F. Dalpiaz, and S. Brinkkemper, “Alignment and granularity of requirements and architecture in agile development: A functional perspective,” Inf Softw Technol, vol. 133, p. 106535, 2021, doi: https://doi.org/10.1016/j.infsof.2021.106535.

[12] N. Medhat, “Granularity in Software Architecture,” nadermedhatthoughts.medium.com.

[13] A. Fuller, Z. Fan, C. Day, and C. Barlow, “Digital Twin: Enabling Technologies, Challenges and Open Research,” IEEE Access, vol. 8, pp. 108952–108971, 2020, doi: 10.1109/ACCESS.2020.2998358.

[14] Q. Qi et al., “Enabling technologies and tools for digital twin,” J Manuf Syst, vol. 58, pp. 3–21, 2021, doi: https://doi.org/10.1016/j.jmsy.2019.10.001.

[15] E. Ferko, A. Bucaioni, and M. Behnam, “Architecting Digital Twins,” IEEE Access, vol. 10, pp. 50335–50350, 2022, doi: 10.1109/ACCESS.2022.3172964.

[16] V. Garousi, M. Felderer, and M. V Mäntylä, “Guidelines for including grey literature and conducting multivocal literature reviews in software engineering,” Inf Softw Technol, vol. 106, pp. 101–121, 2019, doi: https://doi.org/10.1016/j.infsof.2018.09.006.

[17] M. and M. M. V. and R. A. Garousi Vahid and Felderer, “Benefitting from the Grey Literature in Software Engineering Research,” in Contemporary Empirical Methods in Software Engineering, G. H. Felderer Michael and Travassos, Ed., Springer, 2020, pp. 385–413. doi: 10.1007/978-3-030-32489-6_14.

[18] B. Glaser and A. Strauss, Discovery of Grounded Theory: Strategies for Qualitative Research, 1st ed. New York: Routledge, 1999.

[19] J. Corbin and A. Strauss, Basics of Qualitative Research (3rd ed.): Techniques and Procedures for Developing Grounded Theory. Thousand Oaks, California: SAGE Publications, Inc., 2008. doi: 10.4135/9781452230153.

[20] K. Charmaz, Constructing grounded theory: a practical guide through qualitative analysis. London; Thousand Oaks, Calif.: Sage Publications, 2006. [Online]. Available: http://www.amazon.com/Constructing-Grounded-Theory-Qualitative-Introducing/dp/0761973532

[21] J. M. Corbin and A. Strauss, “Grounded theory research: Procedures, canons, and evaluative criteria,” Qual Sociol, vol. 13, no. 1, pp. 3–21, 1990, doi: 10.1007/BF00988593.

[22] M. Liu, S. Fang, H. Dong, and C. Xu, “Review of digital twin about concepts, technologies, and industrial applications,” J Manuf Syst, vol. 58, pp. 346–361, 2021, doi: https://doi.org/10.1016/j.jmsy.2020.06.017.

[23] B. Tekinerdogan and C. Verdouw, “Systems Architecture Design Pattern Catalog for Developing Digital Twins,” Sensors, vol. 20, no. 18, 2020, doi: 10.3390/s20185103.

[24] W. Sulaiman Khail and P. Schnizer, “Patterns in Digital Twin Development,” in Proceedings of the 29th European Conference on Pattern Languages of Programs, People, and Practices, in EuroPLoP ’24. New York, NY, USA: Association for Computing Machinery, 2024. doi: 10.1145/3698322.3698325.

[25] S. Abrahão, M. Staron, E. Insfran, and H. Muccini, “Modeling and Architecting of Complex Software Systems,” IEEE Softw, vol. 41, no. 3, pp. 76–79, 2024, doi: 10.1109/MS.2024.3363541.

[26] A. Rainer and A. Williams, “Using blog-like documents to investigate software practice: Benefits, challenges, and research directions,” J. Softw. Evol. Process, vol. 31, no. 11, Nov. 2019, doi: 10.1002/smr.2197.

[27] M. and Z. O. and P. C. and L. D. Zdun Uwe and Stocker, “Guiding Architectural Decision Making on Quality Aspects in Microservice APIs,” in Service-Oriented Computing, M. and Y. J. and Y. Q. Pahl Claus and Vukovic, Ed., Springer, 2018, pp. 73–89.

[28] E. Ntentos, S. J. Warnett, and U. Zdun, “Supporting Architectural Decision Making on Training Strategies in Reinforcement Learning Architectures,” in IEEE 21st International Conference on Software Architecture (ICSA), 2024, pp. 90–100. doi: 10.1109/ICSA59870.2024.00017.

[29] B. Kitchenham and S. Charters, “Guidelines for performing Systematic Literature Reviews in Software Engineering,” 2007.

[30] J. Tyndall, “AACODS Checklist,” 2010.

[31] R. B. Johnson and L. Christensen, Educational research: Quantitative, qualitative, and mixed approaches. SAGE Publications, 2019.

[32] ISO/IEC 25010, “ISO/IEC 25010:2011, Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models,” 2011.

[33] C. R. China, “Digital twin vs. digital thread: Two complementary ways to digitally replicate assets,” www.ibm.com/think/topics. Accessed: Jan. 26, 2025. [Online]. Available: https://www.ibm.com/think/topics/digital-thread-vs-digital-twin#:~:text=A%20digital%20thread%20is%20a,all%20aspects%20of%20the%20lifecycle.

[34] C. Hunt, “Semantic Model vs Ontology vs Knowledge Graph: Untangling the latest data modeling terminology,” medium.com/@cassihunt. Accessed: Jan. 27, 2025. [Online]. Available: https://medium.com/@cassihunt/semantic-model-vs-ontology-vs-knowledge-graph-untangling-the-latest-data-modeling-terminology-12ce7506b455

[35] O. Zimmermann, M. Stocker, D. Lübke, U. Zdun, and C. Pautasso, Patterns for API Design: Simplifying Integration with Loosely Coupled Message Exchanges. 2022.

[36] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln, Experimentation in Software Engineering. Springer Publishing Company, Incorporated, 2012.