

# Automating build, deployment, and monitoring of model-based digital twins

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

**Diplom-Ingenieur**

in

**Software Engineering/Internet Computing**

by

**Ing. David Markusfeld, Bsc**

Registration Number 01125239

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Mag.rer.soc.oec. Dr.rer.soc.oec. Manuel Wimmer

Assistance: Dipl.-Ing. Daniel Lehner

Vienna, 14<sup>th</sup> January, 2021

\_\_\_\_\_  
David Markusfeld

\_\_\_\_\_  
Manuel Wimmer

# Erklärung zur Verfassung der Arbeit

Ing. David Markusfeld, Bsc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 14. Jänner 2021

---

David Markusfeld

# Danksagung

Zuallererst möchte ich mich bei meinen Eltern bedanken, die es mir ermöglicht haben, das zu machen was ich machen möchte und mich während meines gesamten bisherigen Lebens dabei unterstützt haben. Diese Möglichkeit zu haben ist bei weitem nicht selbstverständlich und daher um so wichtiger hervorzuheben.

Für die Unterstützung und Betreuung möchte ich mich bei Herrn Dipl.-Ing. Daniel Lehner und Herrn Privatdoz. Mag.rer.soc.oec. Dr.rer.soc.oec. Manuel Wimmer bedanken. Vielen Dank, für die Ratschläge und Hinweise die maßgebend dazu beigetragen haben, dass diese Arbeit in diesem Umfang existiert.

Weiters möchte ich meiner Frau für ihre umfassende Unterstützung danken. Ohne sie und den von ihr ausgeübten Druck wäre diese Arbeit vermutlich nicht entstanden.

Vielen Dank auch an meine Studienkollegen und Freunde, die durch ihr regelmäßiges Nachfragen den notwendigen Druck aufrecht erhalten haben, um diese Arbeit zu beginnen sowie sie fertigzustellen.

Abschließend bedanke ich mich bei allen Leserinnen und Lesern für Ihr Interesse an dem Thema und den erzielten Ergebnissen.

# Acknowledgements

First, I want to thank my parents for offering me the possibility to do what I like to do and for supporting me in doing that throughout my whole life. Having this possibility is often not given, and therefore it is even more important to highlight that.

For their support and assistance I want to thank Dipl.-Ing. Daniel Lehner and Privatdoz. Mag.rer.soc.oec. Dr.rer.soc.oec. Manuel Wimmer. Thank you very much for your advice and hints that significantly contributed to the existence and the completeness of this work.

Furthermore, I want to thank my wife for her extensive support. Without her and the pressure she carried out this work would not exist.

Thanks to all my friends and colleagues who by regularly asking for the state of my thesis created enough pressure for me to start and finish this work.

Finally, I want to thank every reader of this work for their interest into the topic and the presented results.

# Kurzfassung

Die Verwendung von digitalen Zwillingen gibt Technikern die Möglichkeit das Verhalten eines physischen Systems nachzustellen. Dadurch können verschiedene Systeme an den digitalen Zwilling angehängt werden, um Daten automatisch zu verarbeiten und davon zu lernen. Das ermöglicht nicht nur Echtzeitüberwachung, sondern auch Defekt-Vorhersagen und kann die Laufzeit eines Geräts, durch automatische Anpassung verschiedener Parameter durch den digitalen Zwilling, verlängern. Allerdings bestehen Limitierungen in den existierenden Entwicklungsprozessen, die zu hohen Zeitaufwänden bei der Erstellung und der Wartung von digitalen Zwillingen führen.

1. Modelle von digitalen Zwillingen werden oft händisch entwickelt und deployt, wodurch eine hohe Fehleranfälligkeit und manuelle Aufwände bei jeder Änderung des physischen Systems entstehen.
2. Um die Verbindung zwischen den physischen Geräten und den digitalen Zwillingen herzustellen, müssen Softwareroutinen im Programm des Geräts hinzugefügt werden. Für eine erfolgreiche Kommunikation müssen außerdem Parameter korrekt eingestellt werden, sowie das Programm auf das Gerät übertragen und gestartet werden. Diese Tätigkeiten werden hauptsächlich manuell und von Hand durchgeführt was zu repetitiven Tätigkeiten mit hohem Fehlerpotential führt.
3. Weitere händische Eingriffe sind notwendig, um digitale Zwillinge und die Geräte, die Daten an diese übertragen, kontinuierlich zu überwachen, um ungültige oder korrupte Daten zu erkennen und gegebenenfalls zu alarmieren oder automatische Anpassungen vorzunehmen.

In dieser Arbeit wird untersucht, inwieweit Techniken aus der Modell-getriebenen Softwareentwicklung verwendet werden können, um die bei den eben genannten Herausforderungen auftretenden manuellen Aufwände zu reduzieren. Dazu wird eine Pipeline implementiert, die in einer universell einsetzbaren Modellierungssprache modellierte Systeme in die spezifischen digitalen Zwillingmodelle und weitere Metainformationen übersetzt und der digitalen Zwillingplattform zur Verfügung stellt. Weiters werden Informationen für die mit den digitalen Zwillingen verbundenen Geräte erzeugt, um die Verbindungsherstellung zu automatisieren. Diese Informationen werden, zusammen mit der Geräte-Software,

durch die Pipeline automatisiert, auf das jeweilige Gerät übertragen und gestartet. Das Modell wird erweitert, sodass Monitoring-Funktionen, wie Bedingungen modelliert werden können. Diese werden automatisch von der Pipeline in eine Maschinen-leserliche Sprache übersetzt und von einer Cloud-Applikation, sowie physischen Geräten verwendet, um ausgetauschte Nachrichten zu überprüfen. Eine Evaluierung der Lösung zeigt, dass die Entwicklung von digitalen Zwillingen durch die gezeigte Automatisierung stark unterstützt wird und viele manuelle Aufwände entfallen oder durch einmalige Vorarbeiten ersetzt werden. Die Ausführungszeiten des digitalen Zwillingenmonitoring werden anhand von 2 physischen Geräten und der Cloud-Applikation ermittelt und basierend auf verschiedenen Parametern verglichen. Die Ergebnisse zeigen, dass durch die Pipeline eine hohe Automatisierung der Erstellung, des Deployment und des Monitorings von modellbasierten digitalen Zwillingen erreicht wird.

# Abstract

Using model-based digital twins gives engineers the possibility to imitate the behavior of a physical device. By doing so, various systems can be attached to automatically learn and interpret the from the physical device received data. This not only enables real-time monitoring and failure prediction but can also improve the devices' lifecycle by automatically receiving adjustment parameters from the digital twin. However, limitations especially in the existing development processes lead to a huge amount of time spent to set up and continuously maintain the digital twin.

1. Digital twin specific models are often created and deployed by hand, which is error prone and requires manual efforts every time the physical device changes.
2. To have devices communicate with their digital twins, software routines need to be included into the devices' program. Furthermore, for a successful communication various parameters need to be set up properly and the program needs to be distributed to the physical devices. These tasks are mainly done manually by hand which lead to repetitive tasks with high risk of an error.
3. Further manual work is required to continuously monitor digital twins and the physical devices feeding data to them to detect invalid or corrupt data and alert or automatically adjust to the occurred error.

In this thesis, it is examined to which extent Model-Driven Engineering techniques can be used to reduce the manual efforts necessary in previously mentioned challenges. Therefore, a pipeline is implemented which transforms systems modeled in a general-purpose language into the digital twin platform specific language models and further meta-information to deploy these to the digital twin platform. The generated meta-informations are used for an automatic communication establishment between the digital twin and its connected physical devices. This information is published to the physical devices together with the software running on those. The models' functionality is extended to include monitoring functions like conditions, which are automatically transformed into a machine-readable language and used by the physical devices and a dedicated cloud application to monitor exchanged messages. An evaluation of the created solution shows that the solution supports the development of digital twins by obsoleting or replacing

manual tasks with one-time tasks. The performance of the digital twin monitoring system is evaluated based on two physical systems and a dedicated cloud application. The results show that by using the pipeline a high automation of creation, deployment and monitoring of model-based digital twins is achieved.



# Contents

<b>Kurzfassung</b>	<b>v</b>
<b>Abstract</b>	<b>vii</b>
<b>Contents</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Problem Statement . . . . .	2
1.3 Methodological Approach . . . . .	3
1.4 Structure of the Thesis . . . . .	4
<b>2 Background</b>	<b>5</b>
2.1 Model-driven Engineering . . . . .	5
2.2 Digital Twins . . . . .	9
2.3 DevOps . . . . .	12
<b>3 Automating Development and Operations of Digital Twins: Requirements and Architecture</b>	<b>14</b>
3.1 Requirements for Automating Digital Twins Creation and Deployment	14
3.2 Requirements for Automated Monitoring of Digital Twins . . . . .	16
3.3 System Architecture for Automated Digital Twins Development and Operations . . . . .	17
<b>4 Automation Framework for Digital Twins Development and Operations</b>	<b>20</b>
4.1 Automating the Deployment of an EMF-based Digital Twin . . . . .	20
4.2 Automating the Connection Establishment of Digital Twins and Physical Devices . . . . .	30
4.3 Automating the Monitoring of Digital Twins . . . . .	39
<b>5 Evaluation</b>	<b>49</b>
5.1 Mapping of Requirements to Evaluation Tasks . . . . .	49
5.2 Evaluation of Automation Potential of Digital Twin Deployment . . . . .	50
	ix

5.3	Evaluation of Scalability of Monitoring Execution Time . . . . .	58
5.4	Discussion . . . . .	70
5.5	Limitations and Threads to validity . . . . .	74
<b>6</b>	<b>Related Work</b>	<b>76</b>
<b>7</b>	<b>Conclusion and Future Work</b>	<b>81</b>
7.1	Conclusion . . . . .	81
7.2	Future Work . . . . .	82
	<b>List of Figures</b>	<b>84</b>
	<b>List of Tables</b>	<b>86</b>
	<b>List of Listings</b>	<b>87</b>
	<b>Bibliography</b>	<b>88</b>

# Introduction

## 1.1 Motivation

Based on emerging technologies like Internet of Things (IoT) or Industry 4.0 [Gil16], there is a steady increase in complexity of software and computer systems [RSK20]. Cyber-Physical Systems (CPS) represent such systems by describing highly integrated software with physical devices [RSK20]. Digital twins (DT) are used to act as a virtual representation of a real-life object which mirrors its behavior and its state over the objects' lifetime [Gri15]. By doing so DT's can help to cope with the complexity of these systems, by visualizing, monitoring, controlling and potentially optimizing the underlying CPS [TZLN19]. DTs are used in a vast field of scenarios respectively disciplines like smart manufacturing [BDH<sup>+</sup>20], farming [BW19], automotive [VTBW21] and avionics [Kra16].

Building digital twins is not a trivial process and requires knowledge of the observed objects and also of the platform digital twins are executed and running on (i.e., the Digital Twin platform), as every platform uses its own language to define digital twins [AK22][PLWW22]. Using a platform-neutral modeling language to model digital twins instead can reduce complexity by leveraging Model-Driven Engineering (MDE) techniques [LSV<sup>+</sup>21]. By using these MDE techniques the DT-platform-specific-models can be automatically generated when needed, while still modeling a generic system model.

However, modeling digital twins is not the only task when it comes to engineering digital twins [EBC<sup>+</sup>22]. Physical devices steadily reporting their status need to be correctly connected to their digital twin and new, as well as modified devices, require continuous maintenance to be able to communicate. Furthermore, infrastructure components and intermediate systems need to be created or updated whenever the system changes. In today's world, changes to software artifacts are done incremental and regularly as the "commit early, commit often" strategy is heavily in software engineering to improve quality and reduce risk [DMG07]. The same can be said for digital twins as models should be

developed, treated and version-controlled like source-code to follow modern software development guidelines [BKL<sup>+</sup>12]. This leads to the necessity of automation when it comes to later phases along the software development lifecycle (SDLC) like deployment and maintenance.

Modern software development uses DevOps teams and processes to tackle the previously mentioned problem, reduce silo building and improve the time between a system change and deploying the change into the production environment [ZBC16]. This DevOps approach can be used for CPS and DTs as well by connecting design-time to run-time models and vice versa [CW20]. "Development" processes can use MDE tooling based on design-time models for Continuous Integration and Continuous Delivery to always hold models in a deployable state whether the models are the final product or an intermediate artifact [GC19]. "Operations" processes like deployment or monitoring are intensively used within software development, but still lack solutions for MDE and modeling in general.

This leaves building, deploying and continuously monitoring of DTs as a complex process, which requires knowledge and efforts from different areas of expertise.

## 1.2 Problem Statement

As more and more complex software systems arise, the need for digital copies imitating the real systems state and behavior increases. Digital twins were developed not just to imitate a systems' state and behavior but can also use big data and machine learning systems to further process the systems' data and potentially gather optimizations and predict failures. When it comes to creating digital twins, most digital twin platforms rely on their own description language and infrastructure. Even when a system is already modeled using a model-based approach, the resulting model (business model) is not applicable within the digital twin platform. That means that a separate model representing the real system, but created in the digital twin platforms' own language (digital twins model) needs to be created and maintained throughout the systems' lifetime. Whenever a new device is added, changed or replaced in the system, the corresponding digital twin needs to be adapted as well. The different platforms offer tools to do so, but each platform uses its own tools that are working exclusively for the certain platform. Finally, for a systems' device to be able to send their telemetry to its digital twin, it must be targeted and configured to send messages to the correct target. As soon as devices start to communicate with their corresponding digital twins, they need to be continuously monitored to detect failures and conspicuousness early and alert or adjust to it accordingly.

This is not a one time process as systems change over time which leads to further iterations of not only the digital twins, but all connected and associated components. Creating and deploying the digital twin, connecting devices to it and correctly hooking up monitoring rules are essential and in parts complex processes. These processes are currently mostly executed by hand in a repetitive and error-prone manner. Not only manual work but

also the time it takes for a change to be represented in an environment is important, as changes can be further delayed by companies' security policies on who can access certain resources. As a developer might not be allowed to access the digital twin platform directly and has to wait for the operations department to enable the changes.

This shows, that the engineering of DTs in its current practice requires a lot of effort. MDE is expected to reduce these efforts but is hardly used in this area of expertise. Therefore, it is unknown to which extent MDE can reduce the efforts when it comes building, deploying and monitoring DTs.

The goal of this thesis is to examine to which extent manual efforts can be reduced when creating, deploying and monitoring model-based digital twins. This is done by comparing the number of manual tasks necessary in a manual setup to the number of manual tasks necessary in a MDE-supported setup when (1) creating and deploying the digital twins to the digital twin platform, (2) connecting physical devices to their digital twins and (3) monitoring and evaluating data of physical device and digital twin. More precisely, this requires answering the following research questions:

**RQ1:** *To what extent can MDE techniques be used to reduce the manual effort for deploying digital twins?*

**RQ2:** *To what extent can MDE techniques be used to reduce the manual effort to connect the physical devices to digital twins?*

**RQ3:** *What is the scalability in execution-time of conditions and actions monitoring runtime data (i) on a physical device compared to (ii) in a cloud application?*

### 1.3 Methodological Approach

To provide a solution to the posed challenges, the Design Science Research approach is followed. Its base was laid by Salvatore T. March and Gerald F. Smith in 1995 in "Design and natural science research on information technology" [MS95] and is being used widely throughout modern IT research. Newer publications regarding Design Science Research simplify the approach down to the iteration of two activities: "designing an artifact that improves something for stakeholders and empirically investigating the performance of an artifact in a context" [Wie14]. While the "three-cycle view of Design Science Research" by A. Hevner [Hev07] describes the Design Science process in a generic way, a more specific documentation by R. Wieringa is given in his book "Design Science Methodology" [Wie14]. Based on this the following steps describe the specific methodology used in this work to find an answer to the problems described by the research-questions:

1. *Derive goals and requirements from research questions.* For each of the three research questions posed in Section 1.2 non-functional and functional requirements are derived.

2. *Propose an artifact to satisfy the posed requirements.* Based on the requirements derived in step 1 the *Automation Framework for Digital Twins Development and Operations* is proposed as a solution.
3. *Design use case for the proposed artifacts.* To proof the feasibility of the underlying concept an air quality sensor use-case is created that can be used within the evaluation.
4. *Evaluate the fulfillment of the posed requirements.* To come up with answers to the posed research questions, (1) a case study is executed to evaluate the automation potential of the digital twin deployment (RQ1) and (2) the automation potential of the physical device deployment (RQ2). (3) To evaluate scalability of the monitoring framework running on a physical device compared to in a cloud application (RQ3), a simulation experiment is conducted and evaluated.
5. *Obtain related work from the literature.* It is searched for literature that relates to the created solution. Research is based on previously defined search questions and keywords relating to the work and its research questions.
6. *Compare the obtained results with the found related work.* Solutions gathered from the literature search are compared to the for this thesis implemented solution, using the requirements derived in step 1.

## 1.4 Structure of the Thesis

The structure of this thesis is based on guidelines proposed by B. Minto [Min97]. In Chapter 2, the theoretical background of topics addressed in this work is described. Chapter 3 depicts the proposed solution and lists requirements to be fulfilled by the implementation. The implementation is described in detail in Chapter 4. Chapter 5 evaluates the developed implementation and finds an answer to the posed research questions. In Chapter 6, the created artifacts are compared against other approaches and solutions taken from a detailed literature search. Chapter 7 concludes and summarizes the thesis and gives an outlook on potential future work.

# Background

This chapter is devoted to giving an overview on the theoretical background necessary to understand and follow explanations in this work. First, Section 2.1 focuses on Model-Driven Engineering, models in general and model transformations. Section 2.2 gives more details on digital twins, their platforms and Microsoft Azure as an example digital twin platform. In the last section an introduction into DevOps and DevOps processes is given.

## 2.1 Model-driven Engineering

Model-Driven Engineering is a software development methodology that uses models as the central artifact throughout the process. The use of models brings with it the ability to abstract complexity of the domain of interest[BCW12]. By doing so MDE follows the concept of "everything is a model"[Béz05].

The main aspects of MDE are shown in Figure 2.1 and split into two dimensions, namely the *implementation* (vertical) and the *conceptualization* (horizontal) dimension. The vertical dimension is represented by the *modeling* layer, the *automation* layer and the *realization* layer which features the transition from a model definition into a running system. In the *conceptualization* dimension the process of defining various conceptual models over different levels of abstraction using the *application* layer, the *application domain* layer and the *meta-level* is described [BCW12].

### 2.1.1 Modeling languages and models

To be able to create a model, an abstraction of a real world object, a certain syntax is necessary to allow common understanding and avoid misinterpretations. Modeling languages are introduced to define elements and rules to help solve these challenges. Commonly modeling languages allow textual as well as graphical representations of models by making use of diagrams. Diagrams extract parts of a model relevant for a

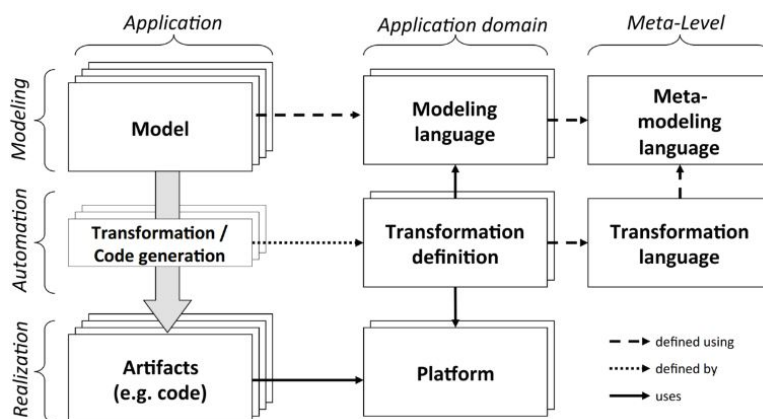


Figure 2.1: Overview of MDE methodology [BCW12]

particular graphical representation. Several diagrams can use the same element of a model but group and display it differently to create a distinct view.

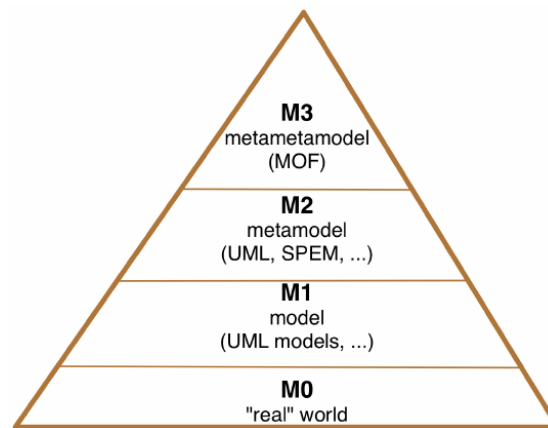
Modeling languages can be classified into two groups: *General-purpose languages* (GPL) and *Domain-specific languages* (DSL). DSLs focus on a certain system or aspect of a system and by doing so create high-level abstractions to reduce efforts and require fewer details to specify a given system. General-purpose modeling languages on the other hand allow modeling of multiple aspects within and between systems and therefore enable more comprehensive models [BCW12].

In general, models are considered static entities used during design phases are often neglected in later life-cycle phases. These models have been used for development in many settings and domains and are commonly called *design-time models* [CW20]. Modern engineering tries to re-use these models by enriching them with runtime information based on the systems state and by doing so turning them into *runtime models* for further usage scenarios [WMWH20].

### 2.1.1.1 Meta-modeling

Meta-modeling is a key functionality of MDE to further reduce the complexity of models by introducing another level of abstraction regarding a specific real world object or system. A model is valid regarding a given metamodel if all model elements and the model itself conforms to it. Since the metamodel is another model, it must be formalized by another language. These so-called meta-languages are dedicated for defining metamodels [CFJ<sup>+</sup>16]. This principle can be further developed as seen in Figure 2.2 to a point where a meta-metamodel defines a distinct way of defining metamodels. The Meta Object Facility (MOF) [OMGa] is a modeling standard specified by the OMG to define metamodels for a particular domain.



Figure 2.2: Model-driven-engineering pyramid [LZN<sup>+</sup>14]

### 2.1.1.2 Eclipse Modeling Framework (EMF)

The *Eclipse Modeling Framework* (EMF) offers such a meta-language, namely *Ecore*, which can be seen as an implementation of the MOF standard. It enables meta-modeling which allows model standards like the *Unified modeling Language* (UML)[OMGc] or *System modeling language* (SysML)[OMGb] to be created based on EMF. EMF is an open-source framework and one of the most used modeling languages. It is based on the programming language Java and is part of the *Eclipse* open-source foundation.

### 2.1.2 Model Transformations

As described in Figure 2.1, transformations are, besides models and modeling languages, another integral part of MDE. They consist of rules which describe how to transform a certain model into another model or text. The defined rules make use of metamodels to create the necessary references between the input and output. When executing a transformation the input model needs to conform to the referenced input metamodel [CFJ<sup>+</sup>16]. When translating input model A into output model B, both A and B have to be compliant to their according metamodels. This type of transformation is called a *model to model* (M2M) transformation. When producing text as an output instead, the transformation is called a *model to text* (M2T) transformations [BCW12].

When using EMF, a M2M transformation uses an EMF-compliant model as an input and output another EMF-compliant model as well. M2T transformations on the other hand take an EMF model as input and output plain text. However, even if the output plain text is structured and follows a certain text-based model, this does not comply to the definition of an EMF M2M transformation as input and output must conform to an Ecore based metamodel.

### 2.1.2.1 ATLAS Transformation Language (ATL)

The *ATLAS Transformation Language* (ATL) is a model-to-model transformation language. It offers the functionality to define rules that express how one (or more) target models can be produced from a set of source models. It uses the inputs' and outputs' meta-model to support transformation development within the Eclipse platform, to validate defined rules and to validate inputs given during runtime.<sup>1</sup>

For usage within the Eclipse platform, the ATL Eclipse platform offers autocompletion and other helpers during development time as well as run-configurations that simplify running the transformation by referencing input and output documents as well as the necessary metamodels. Additionally, ATL can use Eclipse's registry to access already registered metamodels. For the usage in a from Eclipse's registry decoupled environment

- ATL compilation,
- EMF model registration and loading, as well as
- configuring the ATL launcher and
- launching virtual machine

needs to be implemented. To be able to run ATL based on EMF compliant models, the given rules are compiled before being able to be processed by the ATL virtual machine. Figure 2.3 shows an excerpt of how ATL rules are compiled to XML based asm-files before being used by the virtual machine.

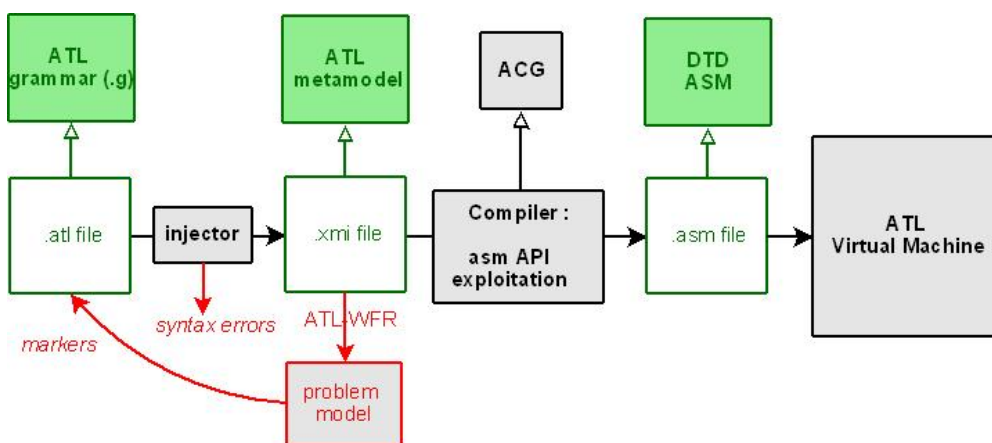


Figure 2.3: ATL compilation process<sup>2</sup>

ATL internally uses an abstraction called "virtual machines" (VM) to support not only EMF but also other modeling languages. Currently, two VMs namely the *Regular VM*

<sup>1</sup><https://wiki.eclipse.org/ATL/Concepts>

<sup>2</sup>[https://wiki.eclipse.org/ATL/Developer\\_Guide](https://wiki.eclipse.org/ATL/Developer_Guide)

and the *EMF-specific VM* exist, with the strong advice to use the later one as it eliminates performance issues when using EMF models in the *Regular VM*. The VM is also shown Figure 2.4 titled as *ILauncher*. To be able to run the VM the according EMF models and metamodels need to be registered and loaded using the *ModelFactory* as well as *Iinjectors* and *IExtractors*.

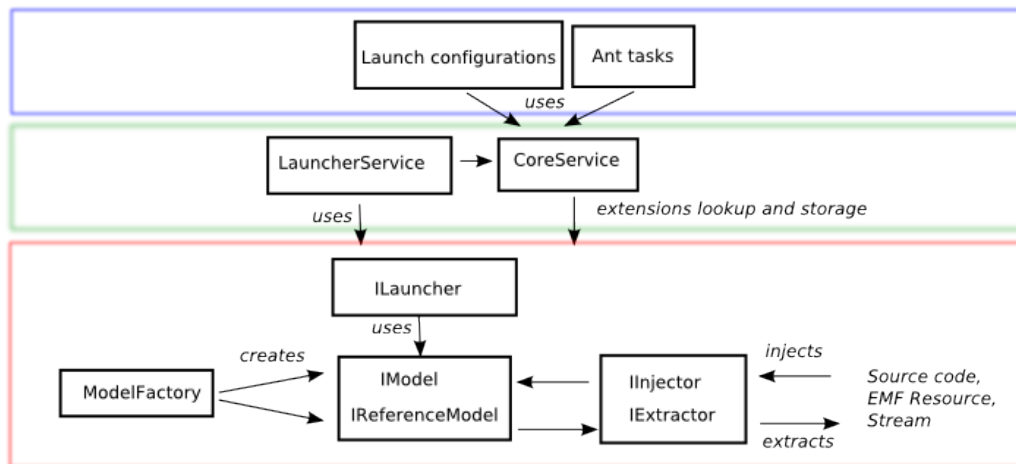


Figure 2.4: ATL Core Architecture<sup>3</sup>

### 2.1.2.2 Xtend

*Xtend* is a statically-typed programming language which translates to Java source code. By using its *Template Expressions* it offers possibilities to be used as a model-to-text (M2T) transformation language. Transformation rules are defined as Java methods offering iterators and conditional operators to create string-templates for a given metamodel. To be able to use a metamodel in *Xtend*, it has to be transformed into Java classes first.<sup>4</sup>

To be able to run the *Xtend* M2T transformation, EMF uses *Xtext* to translate the transformation rules defined in *Xtend* into Java source code. Further setups need to be created to load inputs and write outputs to a certain files and directories.

## 2.2 Digital Twins

The term digital twin was first used by M. Grieves in a University lecture in 2003 [Gri11]. By then the information on a physical product was immature and mostly collected by hand in a paper-based representation [Gri15]. Almost two decades later, the handling

<sup>3</sup>[https://wiki.eclipse.org/ATL/Developer\\_Guide](https://wiki.eclipse.org/ATL/Developer_Guide)

<sup>4</sup>[https://www.eclipse.org/xtend/documentation/203\\_xtend\\_expressions.html#templates](https://www.eclipse.org/xtend/documentation/203_xtend_expressions.html#templates)

of models in production processes changed a lot. Models, if not created prior to the production, are developed in parallel and maintained throughout a products' lifetime.

The modern DTs are no longer only a digital build plan but serve as a digital representation of a physical system, which reflects its state and behavior. According to a study conducted by Dalibor et al. [DJR<sup>+</sup>22] digital twins are mostly used for *behavior optimization*, *monitoring* and *validation*. Together, these three categories make up more than 70% of all fields digital twins are used in. Digital twins are used in various fields such as smart manufacturing [BDH<sup>+</sup>20], farming [BW19], automotive [VTBW21] and avionics [Kra16].

Another very important argument brought up by Dalibor et al. [DJR<sup>+</sup>22] is that according to their study, about 65% of the analyzed publications state that the digital twin is developed in a separate process and not integrated into the system' or product' development life-cycle.

### 2.2.1 Digital Twin Platforms

For a digital twin to actually be able to gather or receive data from a supervised system, not only the specific technical solution but also an environment to run it is necessary. Digital twin platforms offer predefined structures, tooling and support to overcome the different implementations available on the market. Lehner et al. [LPT<sup>+</sup>22] analyzes the digital twin platforms offered by 3 major cloud providers Microsoft Azure, Amazon Web Services and the Eclipse digital twins based on functional and non-functional requirements. Although concepts described in this work are applicable to most Digital Twin Platforms, the practical implementation in this thesis was done using the Microsoft Azure Digital Twins Platform.

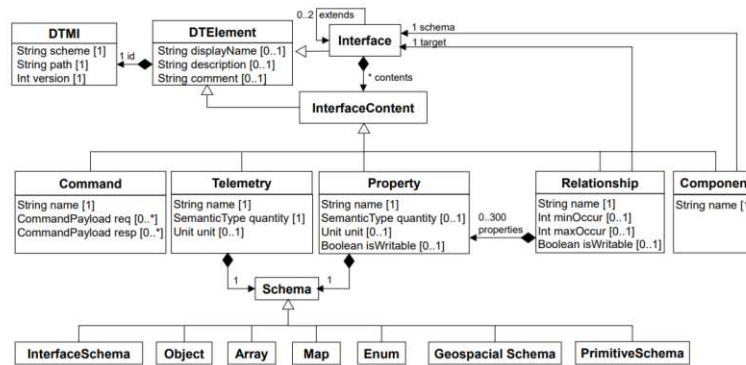
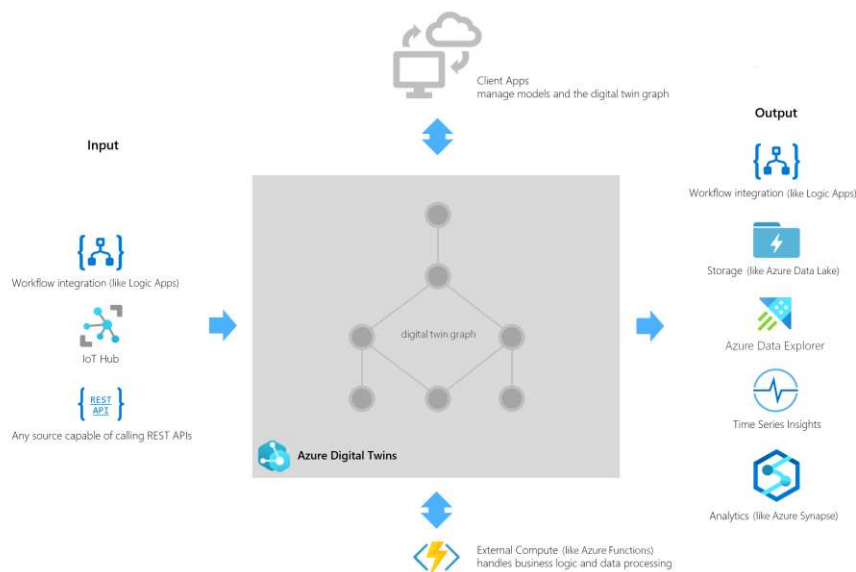
#### 2.2.1.1 Azure Digital Twins Platform

The digital twin platform provided by the cloud provider Microsoft Azure *Azure Digital Twins* is a digital twin platform-as-a-service middleware. It allows modeling digital twins based on the *Digital Twins Definitions Language (DTDL)*<sup>5</sup> as well as the creation and monitoring of digital twins with the browser-based graphical user interface *Azure Digital Twins Explorer*<sup>6</sup>. Figure 2.5 shows the Digital Twins Definitions Language in UML class diagram notation. A programming interface including wrapper-libraries for various programming-languages is available for programmatic access to the digital twins.

Furthermore, the platform offers a rich variety of cloud components to interact with the digital twin as shown in Figure 2.6. The platform not only allows inputs using the programming interface (API) but also via devices connected to the *IoT Hub*. Data egress is possible using various predefined Azure services like big data (Azure Data Lake) and database (Time Series Insights) as well as own programs using *Azure Functions* to handle specific business logic.

<sup>5</sup><https://github.com/Azure/opensdigitaltwins-dtdl>

<sup>6</sup><https://explorer.digitaltwins.azure.net/>

Figure 2.5: Conceptual *DTDL* metamodel in UML class diagram notationFigure 2.6: *Azure Digital Twins* solution overview<sup>7</sup>

**Azure IoT Hub** is a managed service offered by the Microsoft Azure platform. It acts as a central message hub for Internet of Things devices and their connected applications. It supports several messaging patterns including telemetry from the device to the cloud, as well as cloud to device messages to control remote devices. Different services, like machine learning or advanced analytics can be connected to messages processed via the IoT Hub. For an IoT device to be able to communicate via the hub, it needs to be registered to it. By using the created connection credentials, the device is now able to securely communicate with the IoT Hub in the cloud.<sup>8</sup>

<sup>7</sup><https://docs.microsoft.com/en-us/azure/digital-twins/overview>

<sup>8</sup><https://learn.microsoft.com/en-us/azure/iot-hub/iot-concepts-and-iot-hub>

### 2.2.2 Model-Driven Engineering for Digital Twins

Digital twin platforms already offer tooling to design and create digital twins. However, the development, maintenance and evolution, so their change over time, still presents challenges that need to be overcome. Bordeleau et al. [BCE<sup>+</sup>20] analyzes the main challenges and presents where and how MDE can be used to contribute to them. Seven open research challenges are named. Two open challenges, namely the "Modeling Languages for Digital Twins" and "Architectural Framework for Digital Twins" were picked up by Lehner et al. [LSV<sup>+</sup>21]. In this paper a framework is introduced that offers a modeling language and a basic architecture that should allow the reducing of effort when building digital twins. That was done using an in AutomationML modeled system to create and deploy necessary Azure Digital Twin resources from it.

Another challenge, proposed by Bordeleau et al. was picked up by Kirchhof et al. [KMR<sup>+</sup>20]. In their work they propose a solution on how to integrate development of the physical system (CPS - Cyber-physical-system) and the digital twin. MDE techniques are used to create post-development connections (tag) between CPS and DT architecture and vice versa. This information is used by a MDE transformation to support communication and synchronization infrastructure between CPS and DT [KMR<sup>+</sup>20].

## 2.3 DevOps

DevOps, or Development and Operations is a term used in Software Engineering in multiple different ways but quiet similar meanings. According to a study conducted by Jabbari et al. [JbPT16] the term DevOps is defined by the combination of communication, collaboration and team-work between development and operations teams. Additionally, they specify that, software delivery, automated deployment, continuous integration and quality assurance play the most important roles in DevOps. Jabbari et al. define the term DevOps as "a development methodology aimed at bridging the gap between Development and Operations, emphasizing communication and collaboration, continuous integration, quality assurance and delivery with automated deployment utilizing a set of development practices" [JbPT16].

Leite et al. define DevOps similar as "a collaborative and multidisciplinary effort within an organization to automate continuous delivery of new software versions, while guaranteeing their correctness and reliability". This definition puts more significance on the automated and continuous creation and delivery of new software versions. It further categorizes processes into the 4 quarters shown in Figure 2.7 and names frequent terms and processes within them [LRK<sup>+</sup>19].

For this work of special interest are, amongst other things, the "delivery" processes *Continuous Integration*, *Continuous Delivery* and *Continuous Deployment*, as well as the "runtime" process *Continuous Runtime Monitoring* which are described in more detail in the following.

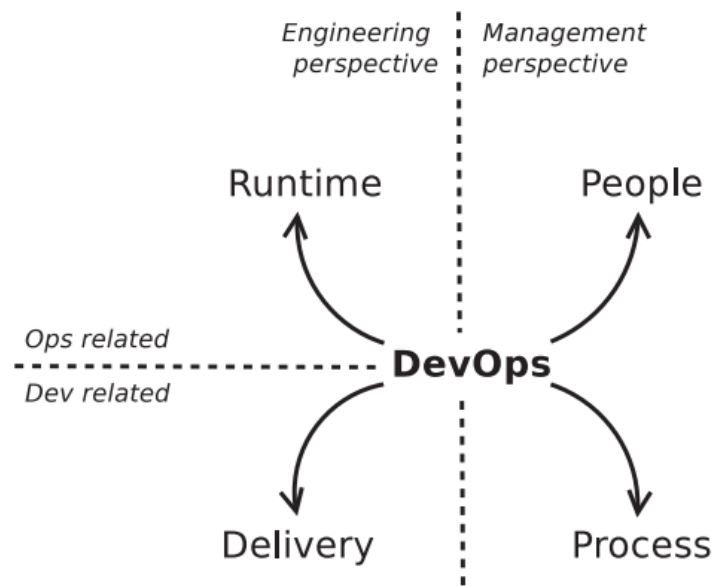


Figure 2.7: "DevOps overall conceptual map" by Leite et al. [LRK<sup>+</sup>19]

**Continuous Integration & Continuous Delivery** Continuous integration (CI) describes a process that is automatically triggered based on the change of a software artifact. It is meant to help a software developer by creating the artifact, run predefined tests on it and give fast or even immediate response without manual actions [DMG07]. By using appropriate tools like source code management and build processes CI can speed up development processes of teams and organizations and produce ready-to-use artifacts for each change done.<sup>9</sup>

Continuous Delivery or Deployment (CD) is a later phase process in the software development life-cycle. It contains further manual and automated tests analyzing a components performance and usability before being able to finally be shipped to a production slot.<sup>10</sup>

**Runtime Monitoring** According to Leite et al. monitoring is a big part of DevOps during the "runtime" stage. Amongst others, performance, availability, scalability, resilience and reliability are the main objectives to be achieved during runtime. Monitoring can be used or is essential to achieve most of them. To do so low-level resource metrics, as well as business metrics can be used, to alert and react upon [LRK<sup>+</sup>19].

<sup>9</sup><https://docs.gitlab.com/ee/ci/introduction/>

<sup>10</sup><https://www.redhat.com/en/topics/devops/what-is-ci-cd>

# Automating Development and Operations of Digital Twins: Requirements and Architecture

To find an answer to the research questions posed in this thesis, a framework to significantly reduce manual efforts when developing digital twins and the corresponding physical devices is proposed. This section is devoted to derive requirements and propose a system architecture for the given challenges. Section 3.1 lists requirements on how to automate the creation and deployment of digital twins and the associated physical devices using Model-Driven Engineering practices in a DevOps process, the requirements on how to extend the creation and deployment of digital twins and the associated physical devices by using design time model annotations to define runtime monitoring are shown in Section 3.2. Finally, in Section 3.3, the basic architecture and approach to solve previously mentioned requirements is presented.

## 3.1 Requirements for Automating Digital Twins Creation and Deployment

To solve the challenges of having to complete various manual steps before a change in the digital twin model is reflected in the digital twin platform, the DT platform itself, the physical devices and infrastructure components between those as well as various other software artifacts need to be developed. A framework that groups and connects these artifacts to build a solution that reduces manual effort significantly is necessary.

Any digital twin components, properties or attributes are described in a proprietary *domain-specific language*, called the digital twin language. Because the format is given by the specific digital twin platform used, there are multiple different languages which



make it hard to switch between platforms and ecosystems. These languages, while being somewhat similar, can currently only be used in their corresponding platform. In today's world of multi- and hybrid cloud solutions this is not favorable. To be able to decouple a digital twin description from a specific platform, a *high-level description* (HL) using the generic digital twin modeling language (gDTL) should be used to describe the digital twin instead.

To be able to create required data for the use in the digital twin service, it needs to be derived from the given high-level information. More specifically, the information necessary to deploy a digital twin are types and instances of the twin. Furthermore, the high-level information needs to yield information regarding the target digital twins environment, so where the information should be deployed to (req 1.1). This information is used to communicate with the digital twin platform, register type information and create necessary digital twin instances in an automated way (req 1.2).

#### **Requirements for RQ1 (automated DT deployment)**

req 1.1 Automatically derive digital twin platform information from HL.

req 1.2 Digital twin can be deployed to digital twin platform automatically.

Before the digital twin platform can work on any data, this data needs to be gathered by sensors connected to physical devices. These devices need to run specific software to not only gather but also send the collected sensor data to a certain digital twin instance. Connection credentials necessary for this communication need to be supplied to each individual physical device (req 2.1). For the digital twin to be able to process received data, information describing the sensor and which value of the sensor was sent, needs to be added. This information needs to be aligned between the digital twin and the physical device to ensure shared understanding (req 2.2). For a high automation level, the physical devices' software as well as necessary configurations should be deployed to the device automatically (req 2.3). To be able to do that on basis of the generic digital twin language (gDTL), it needs to be extended with deployment information like addresses and credentials.

#### **Requirements for RQ2 (automated DT physical device communication establishing)**

req 2.1 Device to digital twin connection credentials are generated, delivered automatically and used by the devices' software.

req 2.2 Device and digital twin use aligned property naming in communication.

req 2.3 Device software and configuration can be deployed to physical device automatically.

Using above described requirements and processes, a pipeline can be built that allows full automation of the creation and deployment of digital twins and physical devices.

By implementing a *Continuous Integration* (CI) pipeline, the transformations can be executed on each change of the input documents and produce the necessary data for deployment. This data can be declared as an output argument of the pipeline so that it can be used by other pipelines. For example a *Continuous Delivery* pipeline can be created so that it automatically uses the CI pipelines' output arguments to deploy the digital twin to a previously configured environment and the physical devices with their according software and configuration.

## 3.2 Requirements for Automated Monitoring of Digital Twins

To monitor digital twins, conditions that inspect messages exchanged between the physical device and the digital twin should be definable. The defined conditions can have actions attached that are executed if a message corresponds to a certain rule. To follow the same approach as before, conditions and actions should be part of the same modeling language the digital twin is modeled in (gDTL) (req 3.1).

To be able to process these conditions, they should be available in a machine-readable and platform-neutral data format so that various systems and platforms can evaluate them and execute the attached actions (req 3.2). This is necessary to be able to answer RQ3 by implementing the conditions and actions evaluation and execution on the physical device (req 3.3) and in the cloud infrastructure (req 3.4). Finally, delivering the machine-readable condition data to the devices and the cloud needs to be done alongside the digital twin and physical device deployment (req 3.5).

### Requirements for RQ3 (automated DT runtime monitoring)

- req 3.1 Conditions and actions can be modeled within the gDTL.
- req 3.2 Conditions and actions are automatically transformed from gDTL into a machine-readable, platform-neutral format.
- req 3.3 Device controller executes conditions and actions before sending updates to digital twin.
- req 3.4 Cloud software component executes conditions and actions from the machine-readable, platform-neutral format.
- req 3.5 Conditions and actions data is published to cloud application and physical device automatically.

Defining monitoring aspects in a design-time stage introduces a way to extend DevOps possibilities further by delivering runtime information back to the developers. The pipeline described in Section 3.1 enables "DevOps for digital twins" by moving responsibilities regarding hosting and deployment towards DevOps teams. By introducing a monitoring framework, the teams possibilities are even further extended.

### 3.3 System Architecture for Automated Digital Twins Development and Operations

In Figure 3.1, the architecture required for fulfilling the previously defined requirements is shown. The *Digital Twins Automation Framework* (DTAF) and its components as well as all targeted systems are depicted. In the following paragraphs, the interaction between the architectures' parts are described.

The proposed solution consists of the three artifacts used for automation, namely the *Model Transformation Engine*, the *Device Deployment Artifact* and the *Digital Twin Deployment Artifact*, a *Device Controller* software in a three layer architecture and cloud infrastructure components *Monitoring Service*, *Device Management Service* and *Digital Twin Service* within the digital twin platform.

The *Model Transformation Engine* is the main artifact and first processing point in regard to automation. The accepted input model and transformation have to correspond to a certain input language. More specific, the transformation provided should provide rules on how to transform the input model into the *gDTL* imposed by the *Model Transformation Engine*. This language shall hold all functionalities necessary to represent a digital twin, the properties necessary to enable automation and conditions and actions (req 3.1). The engine transforms the input model into the *gDTL* according to the given transformation as well as the resulting *gDTL* into *models & instances* corresponding to the *Digital Twin Platform language* (req 1.1) as well as *run-time information* and *deploy-time information* and offer these documents as an output. The *run-time information* contains of information regarding *Device and digital twin aligned property naming* (req 2.2) as well as *Conditions and actions available in a platform-neutral, machine-readable format* (req 3.2) whereas *deploy-time information* holds information on the target *Device Management Service* and *Digital Twin Service* (req 1.1).

The *Digital Twin Deployment Artifact* uses the given *models & instances* in *Digital Twin Platform language* to create the digital twin in the *Digital Twin Service* (req 1.1). Furthermore, it creates the necessary device to digital twin communication credentials at the *Device Management Service* (req 2.1). Finally, the *Monitoring Service* is supplied with runtime-related data from the *Digital Twin Deployment Artifact*.

Using the *Device Deployment Artifact*, software and runtime-configurations like the digital twins' connection credentials and conditions and actions data should be transferred to the remote devices. Furthermore, the software needs to be run on the specific device (req 2.3).

The *Device Controller* software initializes the *Sensor Layer* using *run-time information* to be able to send appropriate meta-information with every message to the digital twin (req 2.2). Before a message is sent, it has to pass through the *Monitoring-Layer* which checks the message content against the condition information contained in the *run-time information* (req 3.4). The *Communication Layer* uses the communication credentials

### 3.3. System Architecture for Automated Digital Twins Development and Operations

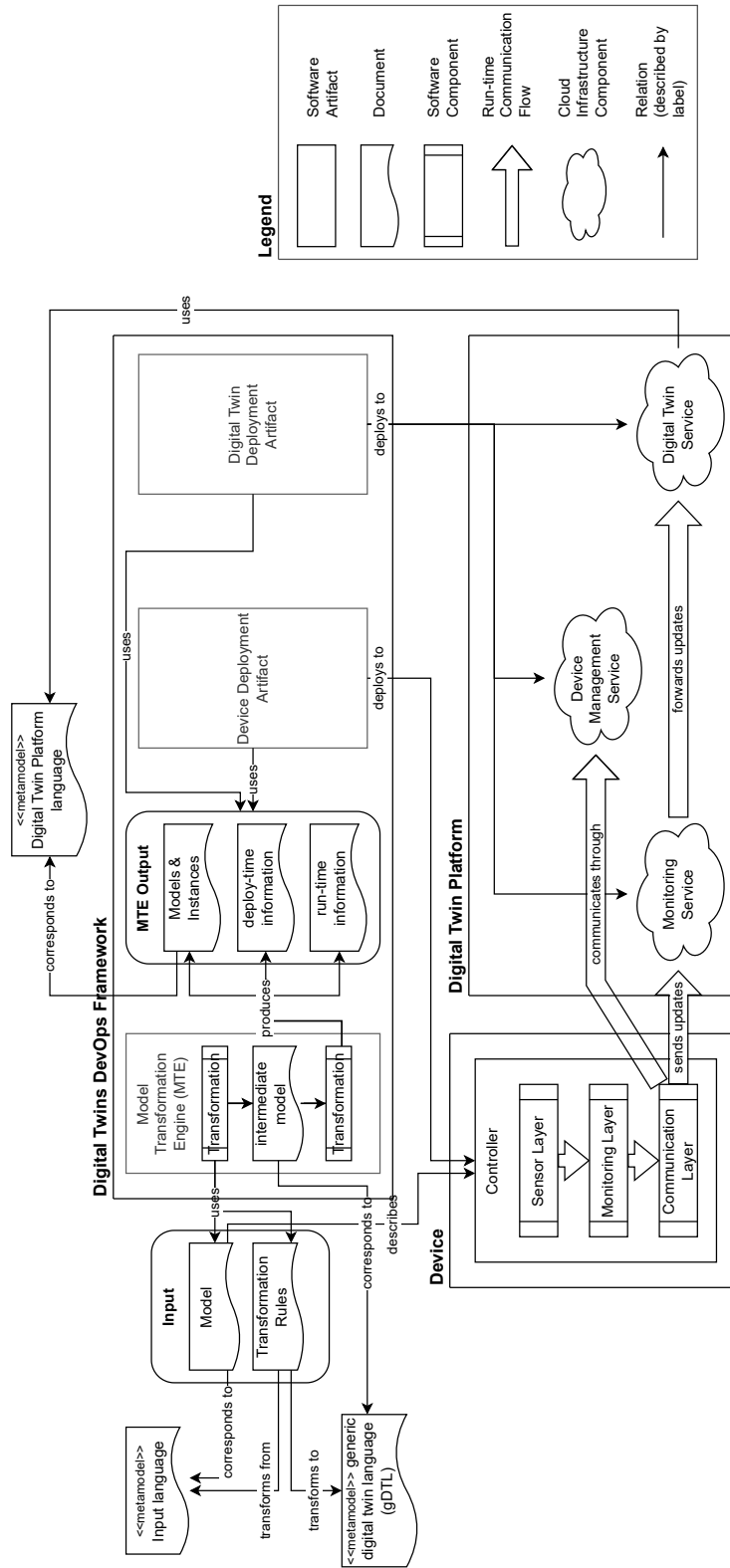


Figure 3.1: System architecture for automated Digital Twin Development and Operations

### 3.3. System Architecture for Automated Digital Twins Development and Operations

---

given in the *run-time information* to establish a connection to the *Device Management Service* to be able to communicate with the *Digital Twin Service* (req 2.1).

A *Monitoring Service* is placed along the "update" data flow, to check any message based on the given conditions and actions' *run-time information* in the cloud. This service can be seen similar to the devices' *Monitoring-Layer*, but as a separate infrastructure component (req 3.3).

# Automation Framework for Digital Twins Development and Operations

In this section, a framework to automate digital twins development and operations based on the system architecture presented in Figure 3.1 is proposed. To enable automation, a Continuous Integration (CI) and a Continuous Deployment (CD) pipeline, as well as how the pipeline can be extended to allow DT monitoring is described in detail. In the first section (4.1) the automated transformation of the input model into the DT model and the setup of the pipeline both running the automated transformation and deploying the resulting model to the DT platform is described. The deployed DT is iteratively extended in Section 4.2, by automating the creation of configuration documents of DT and initializing the device to DT communication. The previous setup is further extended by the implementation of a framework for monitoring of Digital Twins in Section 4.3.

## 4.1 Automating the Deployment of an EMF-based Digital Twin

To successfully deploy a DT in an automated fashion, first the input model needs to be transformed into the DT platform specific language. Section 4.1.1 describes how to enable the use of this transformation in a minimal environment like an automation pipeline, the transformation execution needs to be decoupled from Eclipse IDE. How to execute the decoupled processes in a Microsoft Azure build pipeline is described in Section 4.1.2. The transformed models and instances corresponding to the DT platform language then need to be uploaded into the DT platform to enable the DT. How to

upload DT platform language model to *Microsoft Azure Digital Twins* service using an *Microsoft Azure* deployment pipeline is shown in Section 4.1.3.

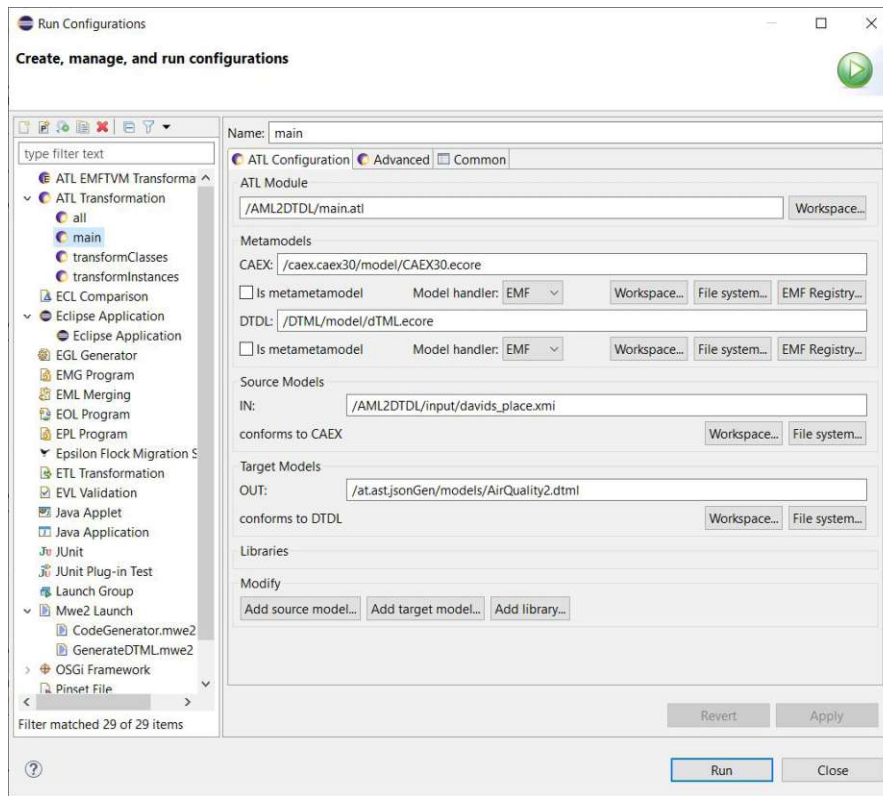
As previously highlighted, it is advisable to model the input-model of a Digital Twin in a generic *Digital Twin Language* (gDTL). This modeling language needs to hold all the information necessary to produce a *Digital Twin Language* based document for a *Digital Twin Platform*. To satisfy the proposed challenges, the implementations proposed by Lehner et al. [LSV<sup>+</sup>21] were used and further adopted to automate the deployment of DTs. The implemented AML4DT framework introduces an EMF-based Azure DTDL abstraction language named *DTML*, which can be seen as a specific implementation of gDTL that will be used within this work. *DTML* can be transformed into JSON-based Azure DTDL documents using the provided AutomationML (AML) to DMTL (AML2DT) model-to-model (M2M) transformation defined in the Atlas Transformation Language (ATL), as well as an ATL model-to-text (M2T) transformation.

According to the *Automation Framework* (DTAF) proposed in this work and shown in Figure 3.1, the AML2DT transformation is used together with *DTML* as the input of the framework used by the *Model Transformation Engine*. This engine also makes use of the existing M2T transformation to produce digital twin documents according to the *Digital Twin Platform Language* which is Azure's *Digital Twins Definition Language* (DTDL). While the general solution described in Section 3.3 is applicable to multiple digital twin platforms, an explicit implementation for the **Microsoft Azure** platform is created in the following chapters.

#### 4.1.1 Decoupling Model Transformation from Eclipse IDE

As the model transformations taken from the *AML4DT* framework [LSV<sup>+</sup>21] already provide most of the logic and functionality needed, the main challenge is to provide further automation. Most *EMF* related tools and languages rely heavily if not exclusively on the *Eclipse* Integrated Development Environment (IDE). To be able to use (1) M2M transformation using *Atlas Transformation Language* (ATL) as well as (2) M2T translation using *Xtext/Xtend* both of these languages need to be made callable outside the *Eclipse* environment as it is not feasible to use *Eclipse* processes as part of a pipeline. It is not an option to run *Eclipse* in a pipeline environment because: (1) Pipelines are usually executed to run on a very slimmed down computer that does not have a GUI environment. (2) *Eclipse* does not offer a simple way to script *ATL* or *Xtext/Xtend* execution without *Eclipse* running. The fact that there is very little to no support of automation is very surprising, given the amount of engineers working with *Eclipse* and its environment. But it shows that the field of Model-Driven Engineering still has huge potential when it comes to further automation.

The M2M transformation language *ATL* defines its rules in *atl* files which are not executable on their own. A run configuration needs to be created where input and output models as well as the *atl* files to use are defined. More complex transformations may be split up into a main *ATL* module with multiple submodules to decrease complexity

Figure 4.1: *ATL* execution configuration in *Eclipse*

and increase readability of the transformation. If so then all modules or files need to be mentioned in the configuration as well. An example *ATL* configuration for *Eclipse* is depicted in Figure 4.1 (note: submodules are configured in the *Advanced* tab).

Using an according run configuration lets *Eclipse* use the *ATL* plugin to execute the transformation. The *ATL* plugin not only requires *Eclipse* for the configuration and execution, but also uses *Eclipse's* *Package Registry*. The *Package Registry* is where *EMF* registers metamodels to be globally usable within *Eclipse*. When loading a project, *Eclipse* searches for all *ecore* files and register them into its registry automatically.

*Eclipse* uses a generic plugin system for every extension that is used within. These plugins contain the necessary libraries (JARs) to fulfill their goal but have an arbitrary interface to *Eclipse* which makes it unusable for automation. To be able to execute model-to-model transformations based on *ATL* a separate interface to control *ATL* configuration needed to be created. For simple and generic usability this interface is designed as a *Command Line Interface* (CLI).

Looking at the *ATL* architecture diagram depicted in Figure 2.4 the newly created CLI needs to use the *ModelFactory* to create and inject *Models* and *ReferenceModels* that represent *EMF* resources like metamodels and their instances. These are then used by



the *ATL Launcher* to execute the transformation.

Furthermore, an *ATL* transformation first need to be compiled into the *ASM* format to be accepted as an input by the *ATL Virtual Machines* (VM). The *ASM* format is an assembly language that *ATL* uses to improve performance and remove references to arbitrary modeling languages. *ATL* supports multiple *Virtual Machines* (VM) which are a generic approach to decouple the generic models from the transformation implementation. Two VMs exist whereas the *EMF-specific VM* is the newer and more performant one<sup>1</sup>. *Eclipse* also compiles *ATL* files to *ASM* files on-the-fly whenever an *ATL* file is changed and saved.

To sum everything up, the new application work needs to do the following to be seen as a general reusable *ATL* executor

- Load and register metamodels (*Reference Model* in *ATL*)
- Load and inject input models with assigned metamodels
- Create or load (if it is an *INOUT* model) and inject output models with assigned metamodel
- Create *ATL* virtual machine and assign *IN*, *OUT* and *INOUT* models
- Compile not yet compiled *ATL* files
- Run *ATL Launcher* in *EMFVM*
- Write results into output models
- Unload models and clean up unmanaged resources.

The full implementation is published and publicly available on *GitHub*<sup>2</sup> together with further documentation and examples.

After finding a solution for executing *ATL* transformations detached from *Eclipse*, the result, an *DTML* conform model must be transformed further into the *Azure DTDL* format. To do so, Lehner et al. [LSV<sup>+</sup>21] implemented a model-to-text transformation (M2T) based on *Xtend*<sup>3</sup>. One could argue that *Azure DTDL* is just another metamodel and therefore a model-to-model (M2M) transformation is the better choice, but *EMF* declares M2M to always output a model that is conforms to a certain EMF metamodel, which is not the case since *Azure DTDL* is a JSON text-based format. Using the given *Xtend* transformation rules the *ATL* tasks' output can be transformed into a valid *Azure DTDL* document. In *Eclipse* this is again done using a proprietary language called *Modeling Workflow Engine* (MWE). Using MWE you can specify source and target

<sup>1</sup>[https://wiki.eclipse.org/ATL/Developer\\_Guide#EMF-specific\\_VM](https://wiki.eclipse.org/ATL/Developer_Guide#EMF-specific_VM)

<sup>2</sup><https://github.com/SirSeven/at.davemar.emf-tool>

<sup>3</sup><https://www.eclipse.org/xtend/>

folders and reference the transformation itself. This is where *Xtend* and *ATL* differ, while *ATL* compiles its rules into proprietary asm files, *Xtend* translates them into *Java* code. From that, it can be said that *Xtend* is a dialect of *Java*, which makes it easy translatable.

The initial plan to solve this challenge was to create a ubiquitous solution just like for *ATL*. But creating a CLI that takes *Xtend* rules and by using them transforms a specified model into any text has a major disadvantage. To do so someone needs to first translate *Xtend* rules to *Java* code and then compile this *Java* code into *JVM* executable bytecode. Using a generic program you are able to then load and execute the compiled *Java* rules using the *Java ClassLoader*. The problem is, this is not only complicated and obviously also dangerous as it can seriously harm your system as you have no control over what is actually executed.

When realizing that the unknown code is executed in a *Platform as a Service* (PaaS) pipeline a foreigners' system, the idea was neglected, and another simpler solution was developed. A simple yet effective solution to this challenge was the assumption was that the transformation from *DTML* to *Azure DTDL* once finished, should be not need to be changed very often, since source as well as target metamodels can be seen as static. So the new plan was to simply use the translated *Java* code and build it into an executable. This executable needs to accept an input and output file path, whereas the input needs to be a valid *DTML* model. By doing so, an executable is created that is not configurable to which rules it executes, the transformation rules can be seen as hard coded into the application. This is where the *Eclipse* platform comes into play again. The *MWE* configuration specifies these parameters and *Eclipse* uses them on-the-fly to execute a compiled *Java* program. While this is a feasible solution, it comes with a lot of overhead and a simple *public static void main()* should do the trick here as well. Listing 4.1 shows an excerpt of the logic to execute a *Java* translated *Xtend* transformation.<sup>4</sup>

Using project management tool *Gradle* the program is compiled into an executable jar and can be used by the pipeline. In case there is a change in the transformation rules, the *Xtend* to *Java* translation has to be done in *Eclipse* and the generated code has to be moved into the *Gradle* solution. In another step this manual process could potentially be eliminated.

#### 4.1.2 Using a build pipeline to automatically execute transformations

After having set up applications to transform an *AML* model into a valid *Azure DTDL* document, these applications are bundled up to assemble the *Model Transformation Engine* and are executed automatically for each incremental change to the model. As the solution is already relying on *Microsoft Azure* for *Platform as a Service* (PaaS) services like *Azure Digital Twin Service* (ADTS), the pipeline needed for automation is also hosted using a *Microsoft Azure* service, called *Azure DevOps*. *Azure DevOps* offers project

<sup>4</sup><https://github.com/SirSeven/at.davemar.dipl.xtext.dtdl>

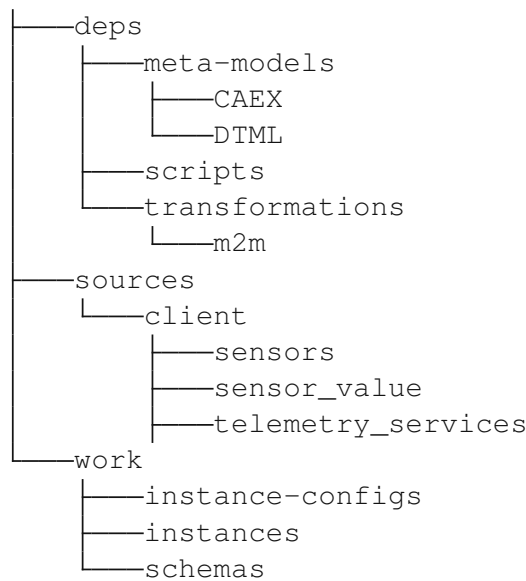
Listing 4.1: Except of *Java Xtend* transformation runner

```

protected void runGenerator(final String string, final String outputPath) {
    final ResourceSet set = this.resourceSetProvider.get();
    Resource.Factory.Registry.INSTANCE
        .getExtensionToFactoryMap()
        .put("*", new XMIRResourceFactoryImpl());
    set.getPackageRegistry().put(DTMLPackage.eNS_URI, DTMLPackage.eINSTANCE);
    final Resource resource = set.getResource(URI.createFileURI(string), true);
    final List<Issue> issues = this.validator
        .validate(resource, CheckMode.ALL, CancelIndicator.NullImpl);
    ...
    ...
    this.generator.generate(resource, this.fileAccess, context);
}

```

and code management services alongside pipelines and artifact servers. For better public availability, the repository is hosted on *GitHub*<sup>5</sup>.



The listing above shows all files that are necessary for the pipeline execution and are therefore checked into the repository. As there is no well known approach for dependency management within Model-Driven Engineering or rather *EMF* solutions, all dependencies need to be checked in to the repository. A better approach is to download them from a package manager when needed during development and build time, just like *Maven* repositories work. Included with the pipeline, is a YAML document (*azure-pipelines.yml*)

<sup>5</sup><https://github.com/SirSeven/at.davemar.dipl.pipeline>

Listing 4.2: Pipeline Model-to-Model transformation command

```

– task: Bash@3
displayName: 'run ATL–transformation (m2m)'
inputs:
  targetType: 'inline'
  script: 'java –jar $(build.SourcesDirectory)/deps/emf–tool–1.0.1–SNAPSHOT–all.jar atl
    ↪ –mm CAEX_Any_Type=$(build.SourcesDirectory)/deps/meta–models/CAEX/
    ↪ GenericAnyType.ecore –mm CAEX=$(build.SourcesDirectory)/deps/meta–
    ↪ models/CAEX/CAEX30.ecore –mm DTDL=$(build.SourcesDirectory)/deps/meta
    ↪ –models/DTML/dTML.ecore –i IN:CAEX=$(build.SourcesDirectory)/sources/
    ↪ davids_place.xml –o:DTD=$(build.ArtifactStagingDirectory)/model.dtd $(
    ↪ build.SourcesDirectory)/deps/transformations/m2m/main.atl $(build.
    ↪ SourcesDirectory)/deps/transformations/m2m/transformClasses.atl $(build.
    ↪ SourcesDirectory)/deps/transformations/m2m/transformInstances.atl'

```

Listing 4.3: Pipeline Model-to-Text transformation command

```

– task: Bash@3
displayName: 'run XTEXT–transformation (m2t)'
inputs:
  targetType: 'inline'
  script: 'java –jar $(build.SourcesDirectory)/deps/at.davemar.dipl.xtext.dtdl–1.0.4–all.jar
    ↪ $(build.ArtifactStagingDirectory)/model.dtd $(build.ArtifactStagingDirectory)/'

```

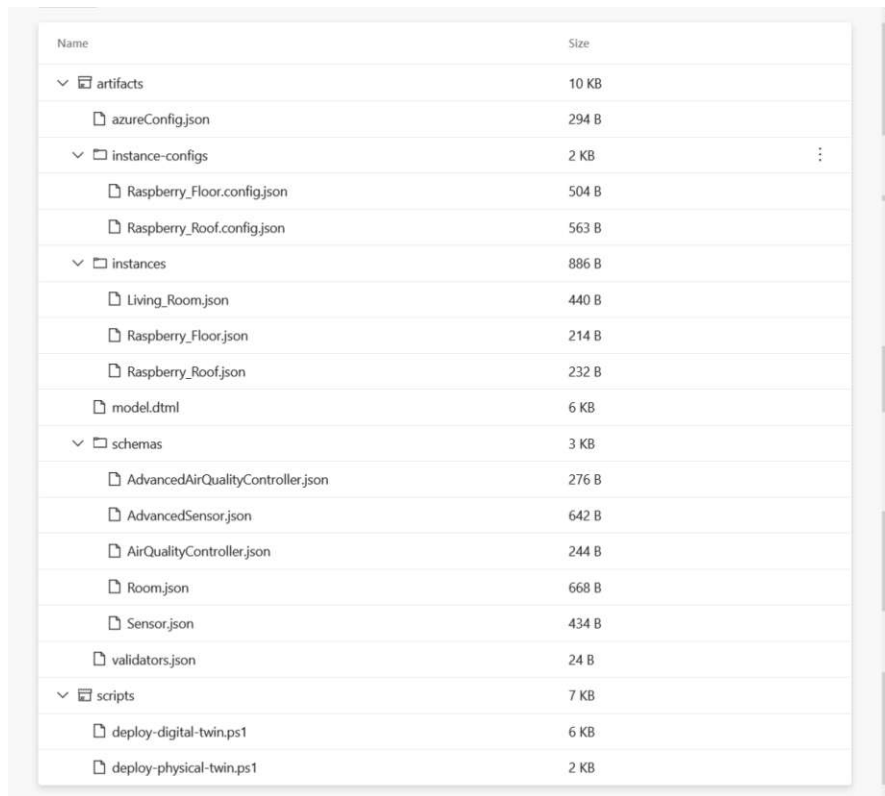
that *Azure DevOps* uses as a pipeline description. By including the build definition with the repository, this allows any build to be reproducible any time in the future.

As everything has already been set up and prepared for a straight forward pipeline execution, the build definition itself is simple. First we need to install *Java* to the pipeline machine (agent). *Java* version 11 was the newest available version by the time the pipeline was created. After *Java* is installed, the *Bash* task is used to run the model transformations using the *java -jar* command (Listings 4.2 and 4.3 depict the detailed commands). Finally, the results are published to the pipelines artifacts using the *PublishBuildArtifacts* command to make them usable in the following deployment process. Figure 4.2 shows the artifacts that are published by the pipeline.

### 4.1.3 Automatically Deploying Digital Twins to Digital Twin Platform

In 4.1.2 all necessary steps to output valid *DTD* files through a build pipeline were taken. This section focuses on creating the *Digital Twin Deployment Artifact* which uses the created *DTD* documents and deploys them to the *Azure Digital Twins* service to create digital twins.

The *Azure Digital Twins* service defines 4 different types of content within each and



Name	Size
artifacts	10 KB
azureConfig.json	294 B
instance-configs	2 KB
Raspberry_Floor.config.json	504 B
Raspberry_Roof.config.json	563 B
instances	886 B
Living_Room.json	440 B
Raspberry_Floor.json	214 B
Raspberry_Roof.json	232 B
model.dtml	6 KB
schemas	3 KB
AdvancedAirQualityController.json	276 B
AdvancedSensor.json	642 B
AirQualityController.json	244 B
Room.json	668 B
Sensor.json	434 B
validators.json	24 B
scripts	7 KB
deploy-digital-twin.ps1	6 KB
deploy-physical-twin.ps1	2 KB

Figure 4.2: Artifacts published by pipeline

every digital twin: (1) Properties, a content type for static data where more read than write operations are performed. (2) Telemetries for highly dynamic values like sensor values. (3) Relationships to other digital twins to be able to build twin graphs of entire environments, (4) and Components which are contained relationships and make reusing of DTs possible. To define a digital twin, a *DTD* conform definition needs to be passed to *ADT*, with which it can verify a twins' integrity and check twin updates for validity. It also enables *ADT* to publish certain events for proper post-processing whenever a twin changes.

There are different ways of passing the definitions to *ADT*. There is a drag- & drop solution to uploading *DTD* files sequentially or a full directory in the *Azure Digital Twins Explorer*<sup>6</sup>, alternatively an HTTP API exists which is also used in the offered *.NET* and *Java* SDKs. The third option is to use the *Azure Powershell* module *azure-iot*. For the use in a pipeline both HTTP API and *Powershell* module are applicable, with the benefit of *Powershell* to already have a simplified interface. As *Azure Powershell* runs platform independent and is available for all major operating systems<sup>7</sup>, it is the

<sup>6</sup><https://docs.microsoft.com/en-us/azure/digital-twins/concepts-azure-digital-twins-explorer>

<sup>7</sup><https://docs.microsoft.com/en-us/powershell/azure/what-is-azure-powershell>

better choice for usage in this work.

As shown in Figure 4.2, the *Xtend* transformation outputs »schemas« and »instances«. Schemas are deployed first using:

Listing 4.4: Creation of digital twin models in *Azure Digital Twin* service

```
az dt model create --from-directory $modelFilesFolder
```

only after that the instances can be created. To do so, we need to iterate each file in the instances folder and use:

Listing 4.5: Creation of digital twin instance in *Azure Digital Twin* service

```
az dt twin create --twin-id $DTId --dtmi $DTModelId --properties
  ↪ $serializedModelProperties
```

Additionally, if instances contain relationships, those need to be created as well. Therefore, we have to iterate over each relationship and run:

Listing 4.6: Creation of digital twin relation in *Azure Digital Twin* service

```
az dt twin relationship create --relationship-id $relationshipId --relationship
  ↪ $relationshipName --source $DTId --target $relationshipTargetId
```

To be able to execute the commands listed above, a certain resource in *Azure* needs to be targeted as there could be multiple *Azure Digital Twin* services running within your account. To identify those two the commands uses the attributes *-n* to determine the given name of the resource in *Azure*, while *-resource-group* points to a certain, as the name says, resource-group, which is a functionality of *Azure* to structure costs and belonging of a certain resource.

```
-n $ADTResource.name --resource-group $ADTResource.resourceGroup
```

To make this information available during deployment multiple options exist. A static configuration of the necessary values in the deployment script or pipeline can be used but comes with the disadvantage that script and or pipeline can not be reused without changes. To prevent this a new property was added to the *DTML* metamodel. The *AzureResource* class was added and holds resource-group and name of an *Azure* resource. It is contained by *DigitalTwinEnvironment* to not interfere with any digital twins.

By adopting the *DTML* metamodel, changes to the *ATL* and *Xtend* transformations of the initial *AML4DT*[LSV<sup>+</sup>21] framework are necessary as well. As implementing meta-information like the *Azure* resource information into *AML* is complicated, the setting is hard-coded in the transformation and the *AML* model stays untouched. Listing 4.7 shows the added transformation rule to create the *AzureResource* class. For the deployment script to work with the information, the *Xtend* transformation is extended to produce another JSON file (see Figure 4.2: *azureConfig.json*), which is then read and used at the deployment (*Deployment-time information* according to Figure 3.1).

Listing 4.7: Additions to *DigitalTwinEnvironment* to include the digital twin *AzureResource*

```

dtdl: DTDL!DigitalTwinEnvironment (
  name ← caex.cAEXFile.fileName.toString(),
  ...
  digitalTwinsResource ← thisModule.newAzureResource('at-davemar-master-thesis',
    ↪ master-thesis', 'azureDigitalTwins'),
  ...
)

```

After adding and using the Azure resource information in the deployment script, the script is added to the pipeline's repository<sup>8</sup> to be version-controlled and available for each deployment. Now finally the pipeline can be created to automatically deploy the created model files to the ADT service. Release pipelines in *Azure DevOps* are defined differently than the build pipelines used in Section 4.1.2. Any release pipeline can have multiple artifacts and stages, whereas artifact sources can be build pipelines or code repositories. A stage is a logical grouping of jobs and tasks to support development, testing, staging and production tenants. For the digital twin release pipeline we only need a single artifact source which is the build pipeline. The build pipelines' artifacts already hold all digital twin documents as well as the scripts necessary for the execution. This could be a potential future optimization to remove the scripts from the models repository and retrieve them from a different source during deploy time. By doing so, this reduces dependencies and can improve maintenance work.

A single stage »Stage 1« with an agent job is introduced. The agent job was configured to run from the *Azure Pipelines* agent pool and run of an *ubuntu-latest* image. This instructs *Azure DevOps* to use an agent from their hosted agents pool which makes having your own build agent needless. Within this agent the previously defined script should be executed. For the script to work an established *Azure CLI* needs to be existent, which makes sense as you don't want to deal with credentials and login here. *Azure Pipelines* offers a specific pre-defined step »Azure CLI« for that, where an *Azure Subscription* can be selected to which it should automatically connect to. Using the CLI step, a script, to be executed on the agent, can be picked, and the appropriate arguments are taken from the selected build pipeline artifacts using the placeholder `__at.davemar.dipl.pipeline`. As the script is using functionality of *Azure Digital Twins*, another step installing the according *Azure CLI* module, is needed. Listing 4.8 shows the by the pipeline executed commands. As release pipelines are not version-controlled within a repository but rather the *Azure Pipeline* platform stores them, an export of the pipeline is attached to the *GitHub* repository<sup>9</sup>.

<sup>8</sup><https://github.com/SirSeven/at.davemar.dipl.pipeline/blob/master/deps/scripts/deploy-digital-twin.ps1>

<sup>9</sup><https://github.com/SirSeven/at.davemar.dipl.pipeline>

Listing 4.8: *Azure* release pipeline commands

```

az extension add --name azure-iot
$(System.DefaultWorkingDirectory)/_at.davemar.dipl.pipeline/scripts/deploy-digital-twin.
↪ ps1 '
$(System.DefaultWorkingDirectory)/_at.davemar.dipl.pipeline/artifacts/azureConfig.json '
$(System.DefaultWorkingDirectory)/_at.davemar.dipl.pipeline/artifacts/schemas '
$(System.DefaultWorkingDirectory)/_at.davemar.dipl.pipeline/artifacts/instances '
$(System.DefaultWorkingDirectory)/_at.davemar.dipl.pipeline/artifacts/instance-configs '
$(System.DefaultWorkingDirectory)/_at.davemar.dipl.pipeline/artifacts/validators.json

```

## 4.2 Automating the Connection Establishment of Digital Twins and Physical Devices

Digital twins are useless without physical devices producing data and sending this data to the DT. How to create such a DT in a model-based manner and how it can be deployed automatically was shown in the previous section. This section uses an already deployed and setup digital twin from Section 4.1. For a successful DT setup, physical devices, digital twins and the connection between them must be created and maintained. This section focuses on automation potential when it comes to connecting a physical device to its digital twin counterpart. In Section 4.2.1 the communication architecture and infrastructure is introduced and areas of interest are listed. Section 4.2.2 introduces a solution on how to establish the connection between physical devices and digital twins. Section 4.2.3 shows necessary tasks to be able to assign updates sent by the physical device to a specific digital twin component within Azure Digital Twins. To automate the previously mentioned steps, Section 4.2.4 focuses on how to automatically deploy the necessary software components and configurations.

### 4.2.1 Digital Twin Communication Architecture

In the previous chapter, the creation of a digital twin definition and deployment to the *Digital Twin Platform* (DTP) is discussed. By doing so, the structure of messages accepted by the digital twin is defined. Also names given to components, properties and telemetries have to be reused by the physical device when communicating, otherwise messages are rejected by the DT. To successfully send a device update to the Digital Twin Service (DTS), not only names and structures have to match, but rather a mapping between any DT telemetry and the value of a physical sensor connected to the device has to be set up.

Before the mapping can be used, the physical device needs to establish a connection to the public endpoint of the DTS to send device updates. Using *Microsoft Azure*, this could be done directly, which means the devices' software needs to include *Azure Digital*



*Twin Service* specific code, or using *Microsofts'* recommended way<sup>10</sup>, by connecting to the *Azure IoT Hub* and sending device updates to the hub instead. This, of course, also requires specific code for communicating with the *IoT Hub* but offers a good layer of abstraction as *IoT Hub* messages can be reused throughout the *Azure* universe. This layer of abstraction on the other hand requires an additional software component to translate *IoT Hub* messages to updates within the *Azure Digital Twins* service. Figure 4.3 shows the described setup using *Azure IoT Hub*, *Azure Digital Twins* service and an *Azure Functions* application used for the translation. According to the system architecture proposed in Figure 3.1 the *Azure IoT Hub* represents the *Device Management Service* that also offers additional message flow systems. The Function application created is represented by the *Monitoring Service* which is described in more detail in Section 4.3.

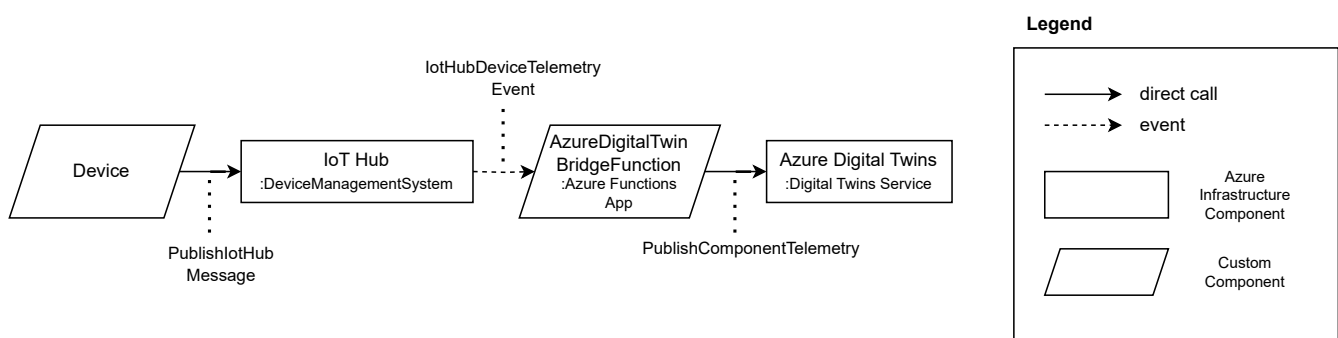


Figure 4.3: Device to Digital Twin communication flow

Based on the *Python* client and *C# Azure Functions* application forked from a group of researchers from the Christian Doppler Laboratory for Model-Integrated Smart Production<sup>11</sup> lead by Daniel Lehner<sup>12</sup>[GJT<sup>+</sup>21], adaptations and improvements should be found, to improve automation in the previously mentioned areas.

#### 4.2.2 Automated Device Management Service Registration and Connection-String Generation

To automatically connect a physical device with its digital counterpart we need to: (1) derive physical devices from the *DTD* or *DTML* description. (2) register each device at the *Device Management Service*, in this case the *Azure IoT Hub*; (3) adapt the client controller program to read and use the connection-string from a configuration file; (4) generate a connection-string for every device and write it into device specific configuration-files.

For the first task, *DTML* is extended by a subclass of the *DigitalTwin* class called *Device* to explicitly mark physical devices. To process this information accordingly, the *Xtend*

<sup>10</sup><https://learn.microsoft.com/en-us/azure/digital-twins/concepts-data-ingress-egress#data-ingress>

<sup>11</sup><https://cdl-mint.se.jku.at/>

<sup>12</sup>[https://github.com/derlehner/IndoorAirQuality\\_DigitalTwin\\_Exemplar](https://github.com/derlehner/IndoorAirQuality_DigitalTwin_Exemplar)

model-to-text transformation was extended to create an empty configuration file for each physical device. This file is subsequently filled with the connection-string information itself. Figure 4.4 shows the extended metamodel.

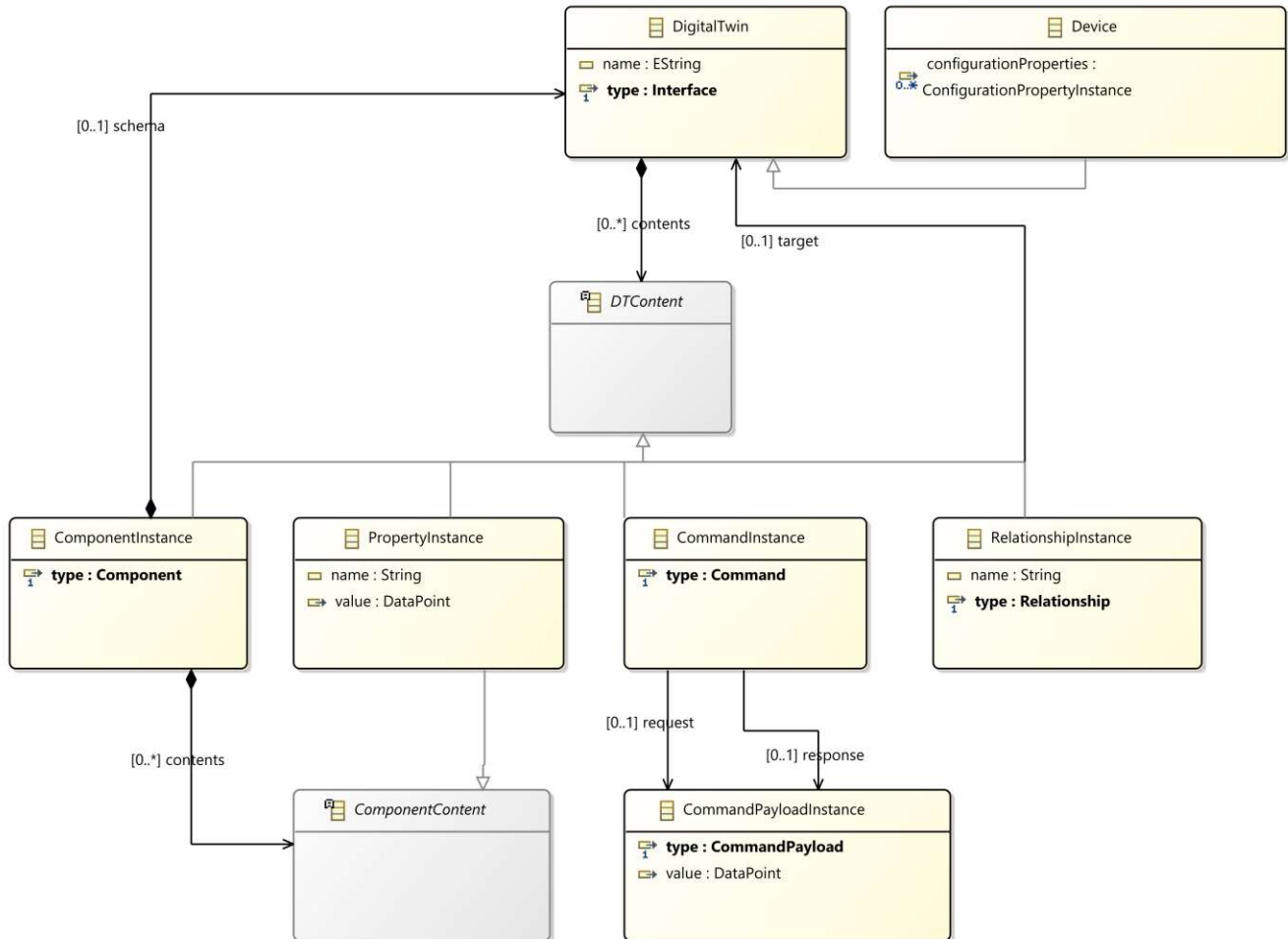


Figure 4.4: Extended conceptual *DTML* metamodel in UML class diagram notation [LSV<sup>+</sup>21]

The next step is to adapt the *Digital Twin Deployment artifact* (i.e., the deployment script) to register each identified device, for which now separate files exist, create a registration in the *Azure IoT Hub*, extract the primary connection-string and write it into the prepared configuration file (*Run-time information* according to the proposed system architecture in Figure 3.1). As the deployment script already has to iterate each DT instance (all files from the "instances" folder see Figure 4.2), if a configuration file for a certain DT exists, use the *Azure IoT Hub* command:

Listing 4.9: Creation of a device in the *Azure IoT Hub* service.

```
az iot hub device-identity create --device-id $DigitalTwinId
```

to create the device. The connection-string is not a result of the previous command but needs to rather be requested explicitly. This is done using the command:

Listing 4.10: Extraction of the connection-string for a certain device in the *Azure IoT Hub* service.

```
az iot hub device-identity connection-string show --device-id $DigitalTwinId
```

Finally, the client's controller program needs to be adapted to use the connection-string from a provided *JSON* file, rather than using a hard-coded one. To do so, the *Python* program was extended by a new main method that takes a file path as an argument. This file is read at the startup and the `config['connectionStrings']['azureIoTHub']` property is used to initialize the *Azure IoT Hub* connection.<sup>13</sup>

### 4.2.3 Mapping components from physical devices to Digital Twins

When deploying a *DTDL* model to *Azure Digital Twins*, every digital twin and its properties, telemetries and further nested components have names attached to them. This is necessary to uniquely identify each one, but also means that an update sent by a physical device, must also use the exact same name for the property it wants to update. Update requests are incremental changes rather than representing the whole device state, as some data changes very often while other data does not. So instead of sending big messages including unchanged data, only the changed properties are reported to the DT to keep the message size low. As described in the previous chapter the device does not communicate with the digital twin service directly but rather to a middleware called the *IoT Hub*. The *IoT Hub* emits messages to a connected *Azure Functions* application that needs to transform the messages' content and publish an update to *Azure Digital Twins*. Figure 4.3 shows the data flow from device to the digital twin.

For a successful device to digital twin communication the following tasks need to be done: (1) create property name mapping from DT model; (2) use it for device controller initialization; (3) define contract between device and *Azure Functions* application (4) and set up *Azure Functions* application to transform incoming messages to *Azure Digital Twins* updates.

First off the structure of the device controller code needs to be adapted to be able to dynamically initialize connected sensors. This change introduces the in system architecture figure (3.1) referenced *Sensor Layer*. It is assumed that a device can have multiple sensors which can yield multiple values. To be able to represent that within the software new interfaces and classes were created for sensors. The *ISensor* interface is the central element of the restructuring and offers methods to get name, data and a

<sup>13</sup>[https://github.com/SirSeven/DigitalTwin\\_Airquality\\_For\\_Covid\\_Risk\\_Assessment](https://github.com/SirSeven/DigitalTwin_Airquality_For_Covid_Risk_Assessment)

timestamp from each sensor. This paired with `SensorValueExtractors` sets up the naming structure corresponding to a digital twin. Figure 4.5 shows the new generic architecture in form of a UML class diagram together with 3 specific sensor implementations for air quality sensors.

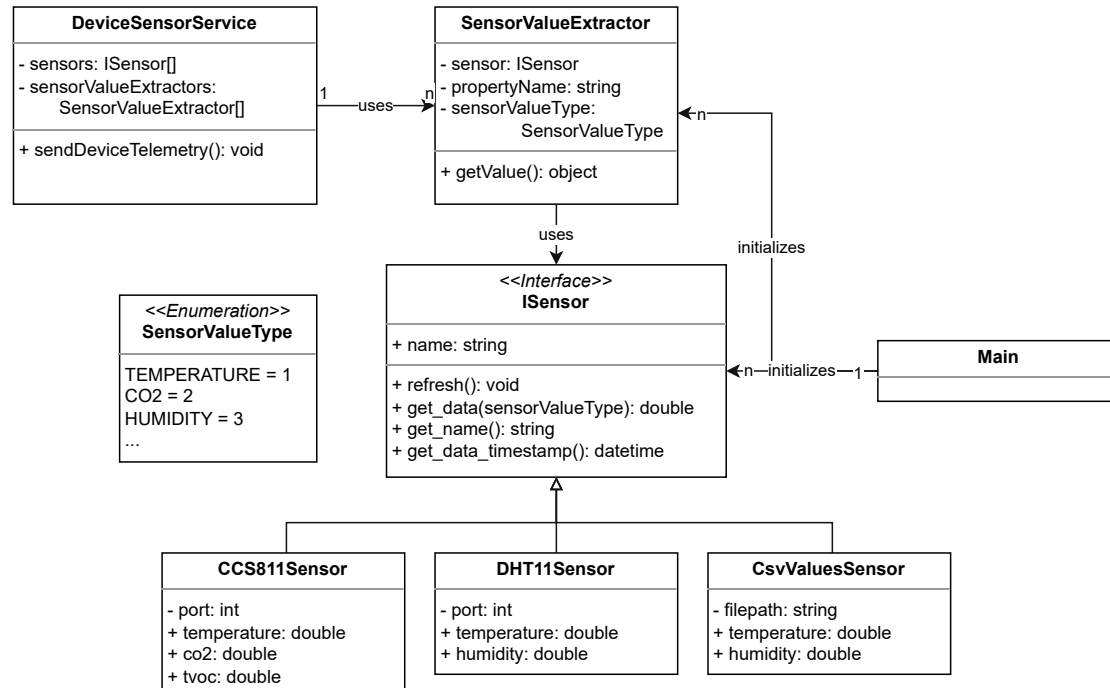


Figure 4.5: Client software architecture diagram in UML class diagram notation representing the device controllers' *Sensor Layer* with 3 example sensor implementations for air quality sensors

To instantiate the new components accordingly the DT model needs to be used by the application. At startup, it loads the model, interprets it and constructs the necessary objects needed for correct communication. When using modern day software engineering practices this DT model should be in *JSON* format. The *Azure DTDL* could be used directly, as it is already using *JSON*, but cannot be easily extended. That means that information necessary for the physical device like "to which port" a certain sensor is connected, needs to be added to a comment field. While this could be a solution, it is also not very pretty as a lot of information in *DTDL*, like *DTID* or reference to the model, is only needed for the digital twin and not for the physical twin. A more favorable approach is to create a separate structure from the *DTML* which only holds the information necessary for the physical device.

Besides the naming, the initialization structure needs to provide certain information necessary for the initialization like the Class and the containing Module that should be instantiated for this sensor. This is of course very dependent on the implementation of

Listing 4.11: Xtend - JSON serialization helper

```

public static String escapeJson(String raw) {
    String escaped = raw;
    escaped = escaped.replace("\\", "\\\\");
    escaped = escaped.replace("\"", "\\\"");
    escaped = escaped.replace("\b", "\\b");
    escaped = escaped.replace("\f", "\\f");
    escaped = escaped.replace("\n", "\\n");
    escaped = escaped.replace("\r", "\\r");
    escaped = escaped.replace("\t", "\\t");
    return escaped;
}

```

the device code and therefore not generally usable. Additionally, various meta-data only interesting for the certain sensor constructor should be held dynamically. The *DHT11* sensor implementation for instance needs information on to which port the physical component is connected to. A created testing-sensor implementation (the *CsvValuesSensor*) reads the values it is emitting from a csv-file and therefore needs to be provided with a certain path to this file.

To store metadata needed by the physical device, the *DTML* metamodel needs to be extended. This is where the generic nature of *DTML* becomes problematic. Sensors or actors are not defined as such in *DTML*, but can be represented as components, relations to another digital twin or properties and telemetries directly attached to a digital twin. While using an assumption could make it possible to find a clear mapping, another idea was followed. This involves adding a generic string property to the Device class that holds the device initialization JSON. This introduces big disadvantages like double maintenance, serialization and de-serialization efforts. Having to manipulate a compressed JSON and keeping it up to date with the rest of the model is a downside and should be tackled in the future. The JSON added to the device class is written into the previously (see Section 4.2.1) prepared empty configuration file. Another smaller issue is the serialization of JSON into a JSON file. For that not only the *Xtend* transformation had to be extended but the script to write the connection-string had to be touched again. A helper class, shown in Listing 4.11, was added to the transformation to replace characters in the textual JSON object

After having set up the transformation and adapted the script, the devices can initialize their internal system appropriately. Sending the data to the *IoT Hub* requires implementing another contract that is used between the physical device, the *IoT Hub* and the *Azure Functions* application. This contract requires holding information of the component path, i.e., the components name plus the property that it wants to update, the properties value and a certain timestamp when the value changed. This contract is represented through the *SensorValueExtractor* and is sent JSON serialized into the *IoT Hub*. Each message can only hold update information on a single property. If a sensor emits multiple

Listing 4.12: C# - Azure Digital Twin bridge function

```
[Function(nameof(AzureDigitalTwinBridgeFunction))]
public async Task Run([EventGridTrigger] EventGridEvent eventGridEvent)
{
    var iotHubDeviceTelemetryEventData = eventGridEvent.Data.ToObjectFromJson<
        ↪ IotHubDeviceTelemetryEventData>();
    string deviceId = iotHubDeviceTelemetryEventData.SystemProperties["iothub-connection
        ↪ -device-id"];

    IoTHubMessage? iotHubMessage = iotHubDeviceTelemetryEventData.ToIoTHubMessage
        ↪ ();
    if (!string.IsNullOrEmpty(deviceId) && iotHubMessage != null)
    {
        IDictionary<string, object> telemetries = new Dictionary<string, object>() {
            {
                iotHubMessage.ComponentPropertyName,
                iotHubMessage.ComponentPropertyValue
            }
        };
        await this.digitalTwinsClient.PublishComponentTelemetryAsync(
            deviceId,
            iotHubMessage.ComponentId,
            null,
            JsonSerializer.Serialize(telemetries),
            iotHubMessage.Timestamp);
    }
}
}
```

properties, multiple messages have to be sent. The device identifier itself is already part of the *IoT Hub* connection, so it does not need to be part of the message.

The *Azure IoT Hub* has various options to egress incoming information from IoT devices. The probably most flexible is to use an *Azure Functions* application. This application should deserialize the incoming message, transform it into a valid *Azure Digital Twins* update message and send it. As all information is already part of the previously defined contract, no further information is necessary. Listing 4.12 shows an excerpt of the mapping logic defined in the *Azure Functions* application.

After adapting the function application, the communication flow is ready, and the device can use the provided structure via a configuration file (adapted *run-time information*) and construct all necessary classes needed for a successful update of the already deployed digital twin.

#### 4.2.4 Automated client artifact deployment to physical devices

This subsection focuses on the automation potential for deploying artifacts to the physical device. Any device that should be automatically deployed is annotated in the *DTML* model, which results in a configuration file being generated holding connection credentials as described in Section 4.2.2 and the data structure needed to instantiate connected sensors described in Section 4.2.3. As the software was prepared to work based on the configuration, it could be seen as a static artifact that is deployed to device only once when it is set up. Therefore, copying the configuration to an already running device is the bare minimum requirement. As the current implementation does not support a file watcher that watches the configuration file and on update trigger a re-initialization of the program, it is also necessary to restart the application on the device. This leads to the conclusion that a file-copy, independently of what ever protocol used, is insufficient, and a remote execution shell needs to be connected and used anyway.

Although previously assumed that the devices' controller software does not change during its' lifetime, when adding another sensor to it, the software needs to be extended as the sensor needs to be implemented. This implies that it might still be necessary to transfer the controller software plus configuration to the device in most cases. It is feasible to instead of deploying the software each and every time, instead an auto-update routine could be added to the code. The issue with that is, that these use polling intervals to check for new software versions and in this time the digital twin might have changed, and the software is still lacking these changes. The same can be said for a cloud-stored device configuration, where no immediate update is triggered, but a polling interval is used.

So deploying a package including the controller software and the configuration seems to be the most appropriate way. The *DTML* model has no limitation on devices or device types, i.e., device A could run software A and device B could run a totally different software B while both software packages use the same configuration format. This might sound unlikely, but internet-of-things devices are offered using different hardware and can often run different programming languages as well. The *DTML* model therefore must offer not only indicate **where** to deploy the software to, but also **what** software. Representing this information in an according *EMF* structure to extend the *DTML* metamodel is, because of the many variants, unfeasible. Instead, a generic `script_to_run` property was added to the Device class which can then be executed when deploying.

The `script_to_run` property can hold what ever script that the operating system it runs on can execute. It can target different software packages and might also hold the necessary deployment information like IP-address usernames and passwords to connect. By doing so all the information necessary is located in the model and an external script has to hold different deployment mechanisms like copying using secure-copy (`scp`) and secure-shell (`ssh`) or others. A huge issue that this solution brings with it, is that credentials are unmasked and unprotected written into the model. This can cause major security breaches in secure environments.

Another big concern is reachability of the devices. For small scale environments like test-

Listing 4.13: `script_to_run` example for testing purposes

```
python sources\copy-deploy.py ignoredIP ignoredUser ignoredPwd sources\client (
  ↪ configurationFilePath)
```

Listing 4.14: Excerpt of the python deploy script

```
datetime_str = datetime.now().strftime("%Y-%m-%dT%H%M%S.%f")
host = f"C:\\Temp\\py-test-{datetime_str}"

print(f"Copying_files_from_{scriptFolderPath}_to_{host}");

if not os.path.exists(host):
    os.makedirs(host)

shutil.rmtree(host)
shutil.copytree(scriptFolderPath, host)
shutil.copy(configFilePath, host+os.sep+"config.json")

call(['python', scriptFolderPath+os.sep+mainExecutable, configFilePath, host+os.sep+"config.
  ↪ json"])
```

labs a IoT device can be made publicly available via a remote-shell, but with increasing security concerns, not actively monitored computers should not be publicly reachable, not even talking about a remote-shell. Firewalls and other network hardware should block incoming connections to these devices, although they might not be the biggest target, they can open the door into a private network. This makes deploying anything to these devices hard and a security concern. *Azure DevOps* provides a possible solution by adding a private agent to the pipelines' execution pool, that can run in the company network and access devices without them being publicly exposed.

Although security concerns persist, a test setup was created that deploys and runs software and configuration to a work-directory. The, in reference to the system-architecture, *Device Deployment Artifact* was realized by extending the pipelines' deployment script to start a background job executing the given `script_to_run` for each physical device identified from the *DTML* model. Listing 4.13 shows the used statement executing a *python* deploy-script (Listing 4.14 shows excerpts of it) with an IP-address, username, password and the location to the software of the client. Through the pipelines script, the location of the configuration file is appended as another argument.

The arguments for IP-address, username and password are ignored here as the deployment is done only on the local machine, but the script can be extended to deploy to a remote target. The *python* deploy-script was developed to copy the provided software folder to a temporary folder and execute it there.



Using the given deployment-script in an *Azure DevOps* pipeline is not possible, as it cannot spawn background-tasks and also stops any execution after a few seconds.

## 4.3 Automating the Monitoring of Digital Twins

This section is dedicated to designing and implementing a solution for active monitoring and acting based on a model-based digital twin. As described in the previous sections, DTs are described using *gDTL*. *gDTL* is later transformed into the digital twin platform (DTP) specific model as well as used for creating necessary device information. The same *gDTL* document should be used to define conditions on properties and telemetries. These conditions are used during runtime to trigger certain actions. While Sections 4.1 and 4.2 focused on setting up physical devices, digital twins and connection them, the following sections define conditions and actions to evaluate the data exchanged between those two parties. In Section 4.3.1 it is described how conditions and actions can be modeled within *gDTL* and how and in which platform-neutral language they are translated. The deployment of the translated conditions to digital twin and physical devices is described in Section 4.3.2. Section 4.3.3 discusses the usage of these conditions along the communication flow and lists possibilities, advantages and disadvantages based on real-world examples.

### 4.3.1 Modeling Conditions

To express conditions against a DT's definition, *gDTL* should be used, as it already contains all necessary DT properties. *gDTL* is based on *EMF* and can therefore use *EMF* tools for modeling. On the other hand, the defined structures in *gDTL* might not be optimal for parsing and executing conditions. Therefore, a certain optimized intermediate language has to be selected and a transformation into the intermediate language needs to be created.

Conditions describe relationships between two or more values, connected using equality operators. Values used in a condition can be static or dynamic, whereas dynamic values in this case describe a certain variable or property of a digital twin. If a condition compares a variable against a static value, it is called unary. If two variables are compared, a binary condition is present. More complex conditions can hold multiple variables and values connected by the logic terms »and«, »or« and »not«. In theory, there are more logical operations, but these can be constructed from above-mentioned terms.

For this exemplary implementation conditions are restricted to exactly two values, whereas only one values is a reference to a digital twin value (unary). This has multiple reasons, (1) it reduces implementation complexity as only one value from a digital twin is compared to a static value. (2) As discussed in Section 4.2.3 each property or telemetry change is sent individually and therefore accessing other values of a certain digital twin, can be problematic as it might no longer be available or outdated. (3) *Azure Digital Twins* do not store the values of »telemetry« properties. Therefore, a secondary system needs to

hold the state of these properties and needs to be queried in case of binary or higher degree conditions. Allowing ternary conditions creates the need of using logic operators »and«, »or« and »not« which further increase complexity. With the latter described solution multiple conditions on one property can be defined and are »and« connected all the time. An »or« solution is not implemented and might be a future addition if necessary.

Before being able to implement a program executing any conditions, solutions for (1) modeling conditions in *gDTL* (2) and serializing and interpreting modeled conditions have to be found. Both modeling and serialization rely on each other heavily, therefore this decision is very important. For this work, different languages were evaluated to find a good candidate to model and execute conditions. The next paragraphs list various solutions and specify advantages and disadvantages for each.

#### 4.3.1.1 Epsilon Validation Language (EVL)

The Epsilon Validation Language (EVL) is a validation language extending *EMF* by using metamodels and allowing the creation of basic and complex conditions in an arbitrary language [KPP08b]. Based on given metamodels, conditions can be created easily even with IDE support when using *Eclipse*. *EVL* additionally allows the definition of (error)messages and guards which make it very flexible when creating conditions. *EVL* uses separate documents and by doing so adds a separate layer on top of the *DTML* metamodel. To evaluate a certain condition a metamodel conform model must be created and loaded using the *EMF* class loader. The *EVL* document can then be read and interpreted using the *EVL* java-libraries. This is somewhat problematic as updates from the device to the digital twin only contain a single property and not the full device state, and therefore a full model cannot be created. Furthermore, the current *DTML* is not designed to be instantiated with runtime data. Another issue, although minor, is that the current *Azure Functions* applications are written in *C#* and *EVL* needs a *Java* runtime to be executed. This means that another separate application needs to be put in place to execute the *EVL* rules. The positive side is, that *EVL* documents can be interpreted and do not need compilation, which means that a similar approach to *ATL* (see Section 4.1.1) can be implemented.

#### 4.3.1.2 Eclipse VIATRA

Another option to define conditions for the given use-case, is *Eclipse VIATRA* (VIATRA) [VP04]. *VIATRA* is another *Java* dialect making use of *EMF* metamodels to describe conditions and transformations. The *Eclipse* IDE can support the creation of basic to highly complex conditions when the according plugin is installed. Conditions and transformations can be created separately and are saved as dedicated files, therefore add the desirable additional layer on top of the *DTML* metamodel. Using the conditions can be tricky as *VIATRA* needs to be compiled to *Java* first and then loaded using a *Java* class loader. Just as with the *Xtext* translation (see Section 4.1.1) this is not a recommended solution as it introduces security threads. Another issue is that, again just like with the

*Xtext* solution, *VIATRA* takes an *EMF* model as input. As already described in the previous section, incremental device update data is insufficient to create a full model which leads to a new challenge when executing the *VIATRA* conditions. *VIATRA* just as *EVL* needs a *Java* runtime to be executed and therefore does not work very well with the already in-use *C# Azure Functions* applications. A separate application needs to be created for running the conditions in the cloud. For running the conditions on a device, this requires every device to be able to run *Java*, which some microcontroller might not be able to.

#### 4.3.1.3 JSON-Schema

With *EVL* and *VIATRA* having potential blockers in regard to the execution, a more execution focussed solution is the usage of *JSON*-schema. *JSON*-schema is primarily used to define the structure of a *JSON* document. In some ways it can be compared to an *EMF* metamodel but for *JSON* files. There are many libraries in various coding languages to verify if a *JSON*-document corresponds to a certain *JSON*-schema and the schema is stored as another *JSON* file so transportation and execution in different environments is not a problem. To actually model the conditions two options exist: (1) create and maintain *JSON*-schema document separately from *DTML* model. Changes to the metamodel are not reflected by the schema as long as it is not changed accordingly. (2) Create arbitrary conditions definition modeling possibilities in *DTML*, and dynamically generate *JSON*-schema from it at build time. The first solution can be compared to what is created with *EVL* and *VIATRA*, but these bring in way better tool support and have their documents and the *EMF* metamodel connected somehow, while the *JSON*-schema is a totally detached. Solution two creates the need of an arbitrary language to create conditions inside *DTML*, but eliminates double maintenance completely by serializing valid schema documents from the model. Representing conditions using *JSON*-schema only allows very basic conditions, as it is not designed to hold complex conditions. Typical operators supported are »maximum« and »minimum« for number data types and »maxLength«, »minLength« and »pattern« for string data. Additionally, each and every property can be made optional. As *JSON*-schema also validates whole documents, which, as described before, is a challenge when working with incremental device updates, the optional flag can be used to suppress errors. Another worked-around could be to break each property into a separate document with its defining schema instead of having only one big schema.

#### 4.3.1.4 JSONPath

*JSONPath* is another candidate to be used to define conditions, although its primary purpose is the search of certain nodes within a *JSON* document. This can be used by defining the search term accordingly to generate a *does a node that evaluates query X = true exist* statement, which states if the condition is hit or not. A *JSONPath* query can be seen as a separate document, but is usually only a few characters long and offers proper equality operators like »largerThan«, »smallerThan«, »equals« and

»notEquals«. Regular expressions are sometimes also supported, but this depends on the library or implementation used. Libraries for executing *JSONPath* exist for all common programming languages, although, not always well maintained. To be able to run the same condition on multiple platforms along the digital twin communication flow, the same *JSON* structure has to be applied to the data, and conditions might need some tweaks as every library implements *JSONPath* slightly different. The creation of *JSONPath* conditions could be done the same ways as *JSON*-schema, but generation is way simpler here as only short strings need to be generated instead of full documents.

#### 4.3.1.5 Designing and transforming Conditions and Actions

After an evaluation of the previously mentioned languages, *JSON-path* was selected. Although *EMF* extension solutions like *EVL* and *VIATRA* offer the best experience when creating conditions by using already provided tooling within the *Eclipse Platform*, the most important part is ease of execution and this is where those solutions are less appropriate given the current scenario. The simplest execution is given when using the *JSON-path* solution. *JSON-path* is especially suiting as it reduces the implementation time as libraries already exist. This implies that conditions will generally be designed in *DTML* and then transformed into *JSON-path* queries for execution. For that an abstract Condition class was introduced and extended by a *UnaryCondition*. An »UnaryCondition« always references a certain operation and a value against which a telemetry or property value should be checked. Figure 4.6 excerpts the newly added classes to model conditions within *DTML*.

Additionally, each condition has a reference to one or more *ConditionActions* and a flag indicating whether after a condition was hit, it should be continued. Each action added to *DTML* needs its own implementation on the certain communication flow stage. For this work a basic *LogAction* was implemented that writes a certain log when the action is executed. For that, the »LogAction« holds a certain »LogLevel« and the »Message« itself. The reference implementation of actions is also shown in Figure 4.6.

For execution simplicity all conditions and actions of a certain *DTML* »Interface« (digital twin or device) are serialized into a *JSON* array, so that if a message from device A is processed, all rules of device A can directly be accessed and executed. The *Xtend* transformation is extended to output all conditions and connected actions into another output file which can then be further processed when deploying. Listing 4.15 excerpts the *xtext* transformation to correctly output the conditions and Listing 4.16 shows an example of the resulting *JSON* structure which is exposed as *run-time information* in the pipeline.

#### 4.3.2 Deploying conditions and actions to Device and Digital Twin

After successful serialization the conditions and actions configuration file needs to be made available to processing points along the communication flow. The communication flow is depicted in Figure 4.3 and shows that the device itself as well as the *Azure Functions*

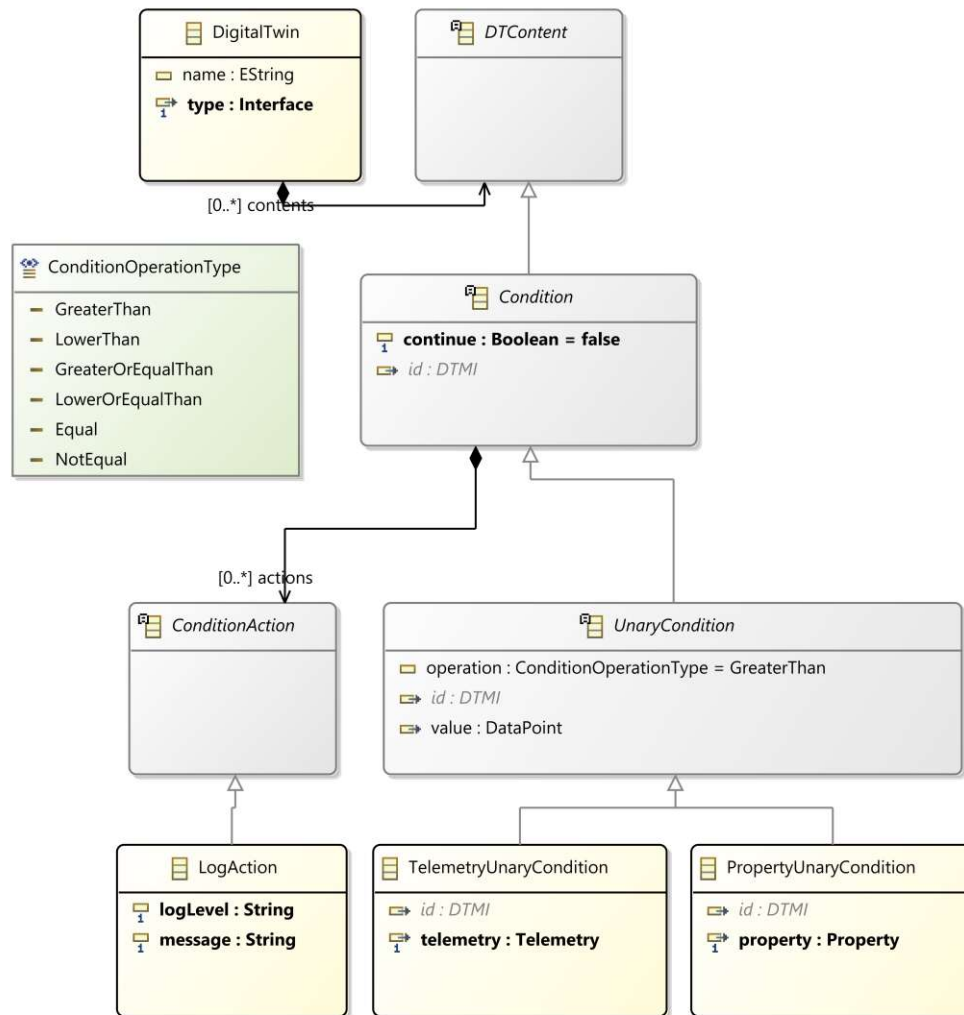


Figure 4.6: DTML condition modeling capabilities in UML class diagram notation

application (*AzureDigitalTwinsBridgeFunction*) could be a potential processing point. Some future work might also look into a "post-storage" execution, where temporal data could be accessed and worked on. This work further focuses on execution of conditions and actions on (1) the *Azure Functions* application that maps *Azure IoT Hub* messages to *Azure Digital Twins* messages, (2) and the IoT device that communicates with the IoT hub. While the implementation of conditions and actions are special to a specific component, the deployment of the configuration might be reusable in other cases as well.

Before any implementation can run conditions or actions on certain incoming messages, the components must be made supplied with the serialized information. In case of the IoT device this is pretty straight forward. As described in Section 4.2.4, one configuration file is already transferred to the device and the application running on it has to be restarted.

Listing 4.15: *Xtext* transformation for conditions and actions

```

<FOR i : 0..<interfaces.size>
  <val interface = interfaces.get(i)>
  <val typeConditions = interface.contents.filter[content|content instanceof Condition]>
  <IF typeConditions.size > 0>
    <IF firstWritten>,<ENDIF>"<interface.displayName>": [
      <FOR j : 0..<typeConditions.size>
        <var condition = typeConditions.get(j) as Condition>
        {
          "jsonPathQuery": <condition.generateJsonPathCondition>,
          "continue": <condition.continue>,
          "actions": [
            <FOR k : 0..<condition.actions.size>
              <condition.actions.get(k).serialize><IF(k < condition.actions.size - 1)
                ↪ >,<ENDIF>
            <ENDFOR>
          ]
        }<IF(j < typeConditions.size - 1)>,<ENDIF>
      <ENDFOR>
    ]<{firstWritten = true; ""}>
  <ENDIF>
<ENDFOR>

```

By adapting the deployment script to accept another argument, the file holding the serialized conditions and actions, the file is added to the copied files and as an argument to the *Python* program that is started on the device. Listing 4.17 shows the adapted script.

Using the same solution to deploy the condition information to the *Azure Functions (Monitoring Service)* application is not feasible, because the internal storage of an *Azure Functions* application, where configuration files are usually placed, is not accessible. Other environments using *Kubernetes* could update the referenced secret or config map and restart their application to read the updated configuration. Using a different configuration provider (an external service) for the functions application and restarting it whenever the configuration changes, is feasible, but the functions' application should be completely independent and should not need to be restarted each time a model changes.

Using the *C#* options pattern, an application can make use of so called *IOptionMonitor* ↪ *<T>*s which can be used to check for changed configurations. This together with a cloud based configuration management system that is updated from the deployment script, can provide an up-to-date configuration to the application. In the *Azure* ecosystem this service is called *Azure App Configuration*. It offers basic key value storage and a ready-made library to use with *C#* applications. Listing 4.18 shows the setup of the *IConfigurationBuilder*.

Listing 4.16: Serialized conditions and actions example

```

{
  "Validators": {
    "Sensor": [
      {
        "jsonPathQuery": "$.[?(@.Property=='co2Value'&&@.Value>=100)]",
        "continue": true,
        "actions": [
          {
            "type": "LogActionImpl",
            "config": {
              "logLevel": "Warning",
              "message": "Critical threshold reached"
            }
          }
        ]
      },
      {
        "jsonPathQuery": "$.[?(@.Property=='type'&&@.Value!='CCS811' )]",
        "continue": false,
        "actions": []
      }
    ]
  }
}

```

Listing 4.17: script\_to\_run example with conditions and actions file

```

python sources\copy-deploy.py ignoredIP ignoredUser ignoredPwd sources\client (
  ↪ configurationFileLocation) (conditionsFileLocation)

```

After registering the *Azure App Configuration* provider, the application can use the `IOptionMonitor<T>` to refresh the configuration and get the current configuration value. For the configuration refreshing to correctly work a sentinel has to be defined. This is needed to reduce reads on the *Azure App Configuration* by reading and comparing only the as sentinel declared key instead of comparing the whole configuration. It has to be said that although a refresher was implemented, the configuration values are still cached for at least 5 minutes (configured in 4.18) to reduce load on the configuration management system.

Finally, after having the *Azure Functions* application set up to work with *Azure App Configuration* the deployment script needs to be adapted to deploy the file holding the serialized condition information into the cloud service and change the sentinel key so that the application informed of the change. As the deployment script was already written in

Listing 4.18: C# application setting up an *Azure App Configuration* configuration provider

```

configurationBuilder
    .SetBasePath(builderContext.HostingEnvironment.ContentRootPath)
    .AddJsonFile("appsettings.json", false, false)
    .AddJsonFile("appsettings.development.json", true, false)
    .AddAzureAppConfiguration(options =>
    {
        TokenCredential credential = builderContext.HostingEnvironment.IsDevelopment()
            ? new DefaultAzureCredential(true)
            : new ManagedIdentityCredential(connectionString);

        options
            .Connect(connectionString)
            .ConfigureRefresh(refreshOptions =>
            {
                refreshOptions
                    .Register("Validators:Settings:Sentinel", true)
                    .SetCacheExpiration(TimeSpan.FromMinutes(5));
            });
    });

```

Listing 4.19: Conditions and actions deployment script for *Azure App Configuration*

```

az appconfig kv import '
  -s file '
  --format json '
  --path $validatorsConfigFile '
  --separator : '
  --content-type application/json '
  -y '
  -name $appConfigurationResource.name

az appconfig kv set '
  --key Validators:Settings:Sentinel '
  --value $(Get-Date -Format "o") '
  -y '
  -n $appConfigurationResource.name

```

*PowerShell*, the *Azure App Configuration PowerShell* commands were used. To target the correct *Azure* resource the already added *DTML* class »*AzureResource*« was used. The necessary commands to upload the configuration from a file and set the sentinel value accordingly can be found in Listing 4.19.



Listing 4.20: *Application Insights* query

```

union traces
| where operation_Name == 'AzureDigitalTwinBridgeFunction'
| where customDimensions['LogLevel'] == 'Warning'
| where message == 'Critical threshold reached'
| project
    timestamp,
    appName,
    operation_Name,
    message,
    LogLevel = customDimensions['LogLevel']

```

### 4.3.3 Evaluating conditions on Device and Microsoft Azure Platform

To execute the previously serialized and supplied *JSON*-path queries on every message, both the devices' *Python* application and the *Azure Functions* application need to be adapted. Before both of these applications send their messages to next stage, the conditions need to be evaluated, and if a condition is hit, its actions must be executed. For this example only one action, the *LogAction*, was implemented.

Whenever the applications needs to check conditions, they: (1) Retrieve all conditions for the given DT component identified by the message. (2) Create a queryable *JSON* message based on the message content. (3) Execute conditions one by one and, if match was found, trigger related actions. To execute *JSON*-path queries the *C# Azure Functions* application uses the *Json.NET*<sup>14</sup>, while the *Python* client uses the *jsonpath\_ng*<sup>15</sup> implementation and introduces the in the system-architecture *Monitoring-Layer* named software component. To execute only as *JSON* specified actions, both applications implement specific action parser. The parsers use the actions' name to identify the according class in the program. Arguments denoted in the conditions and actions *JSON* (see Listing 4.16), are passed to the according class for further processing, but implementations can use these differently. The »LogLevel« information for instance, is unused in the *Python* implementation.

Writing logs is a very simple use-case and not sufficient for most real-life usages, but by adding another *Azure Cloud* feature on top of the logs, a simple but powerful alerting system can be built. Logs and further metadata produced by any *Azure Functions* application is automatically collected and written into the so-called *Application Insights*. This service can be used to create various alerts based on email, text messages, webhooks and many more. For a very basic example the defined log message "Critical Threshold" (shown in Listing 4.16) was used to create an alert rule that sends an email when the log is written. Listing 4.20 shows the query that was evaluated by *Application Insights*.

<sup>14</sup><https://www.newtonsoft.com/json>

<sup>15</sup><https://github.com/h2non/jsonpath-ng>

Alert rules also require configuration of an aggregation time span, the threshold value which the query result count should be checked against and the evaluation frequency. Finally, a certain notification method needs to be chosen and configured.

# Evaluation

In this chapter, the evaluation of the *Digital Twins Automation Framework* (DTAF) is presented in order to answer the research questions posed in Section 1.2. In Section 5.1 the necessary evaluation tasks to validate the requirements, derived from the research questions, (as listed in Sections 3.1 and 3.2) are pointed out. Section 5.2 shows a case study that analyzes the automation potential of a digital twin based on an air quality sensor example to answer research questions 1 and 2. To answer the third research question in Section 5.3, the scalability of the monitoring framework is compared based on its usage on the device or in the cloud. The gathered arguments are discussed to answer the research questions in Section 5.4.

## 5.1 Mapping of Requirements to Evaluation Tasks

In Chapter 3, requirements to be met to allow answering the posed research questions are described. In this section, it is reviewed to which extent the requirements are fulfilled by the implementation described in Chapter 4. The fulfillment of the proposed requirements is validated by connecting each requirement to an evaluation task that is executed to evaluate the particular requirement. This mapping is presented in Table 5.1.

RQ	Goal	Requirement	Evaluation
RQ 1	Automation	Automatically derive DT platform information from HL-information	Case Study
		DT can be deployed to digital twin platform automatically	
RQ 2	Automation	Device to DT credentials are generated and used by devices	Case Study
		Device and DT use aligned property naming in communication	
		Device software and configuration can be deployed to physical device automatically	
RQ 3	Scalability	Conditions and actions can be modeled within the digital twin modeling language	Simulation Experiment
		Conditions and actions are transformed into machine-readable format	
		Cloud software component can execute conditions and actions	
		Device controller component can execute conditions and actions	

Table 5.1: Mapping of requirements to evaluation tasks.

To validate the RQ1 and RQ2 derived requirements, a case study is conducted following the guidelines of Runeson & Höst [RH09]. The execution and results of the corresponding evaluation are described in Section 5.2.3.1 and Section 5.2.3.2.

To analyze the requirements of RQ3, two simulation experiments are conducted. The first simulation experiment targets the created monitoring framework solution running on the IoT device and is described within Section 5.3.2. The second simulation experiment which is evaluating monitoring in a cloud application is described in Section 5.3.3. In Section 5.3, a brief introduction and description of the varied parameters as well as the execution is given.

## 5.2 Evaluation of Automation Potential of Digital Twin Deployment

By using a case study, the applicability of the approach is shown based on a particular scenario. The used scenario is described in Section 5.2.1. The executed evaluation cases and used metrics are described in Section 5.2.2 and finally, based on this scenario, a use case is created, described and evaluated for RQ1 and RQ2 in Section 5.2.3.1.

### 5.2.1 Air quality sensor scenario

A scenario is created based on air quality sensors [GJT<sup>+</sup>21]. Each sensor is connected to a »Controller« which is located in a certain »Room«. Other than »Sensor«s, a controller might also have alarms connected to it. »Sensor«s can gather certain values from the real-world environment and use them for calculations regarding the air quality. If the air quality is insufficient a notification can be given to the room using an alarm. Figure 5.1 visualizes the scenarios baseline and possibilities.

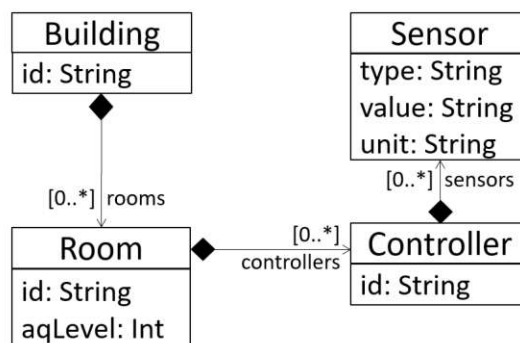


Figure 5.1: Graphical representation of the scenarios baseline

As the scenarios' hardware was not physically available for evaluation, to be able to still validate the approach, the physical controllers and sensors were replaced by locally executed controller software using a test-sensor reading sensor data from a file. The data in the file was collected over a timespan of seven days from an SCD30 sensor in a real world scenario, when almost two persons were working in room during daytime. The file and test-data was taken from the "Indoor AirQuality Digital Twin"<sup>1</sup>.

The **predefined digital twin** is defined as: a building holding 3 rooms (Lobby100, Room101, Room102) with separate controllers (Raspberry1, Raspberry2, Raspberry3) which were all equipped with a CO<sub>2</sub> sensor. Figure 5.2 shows the mentioned setup in the UML object diagram notation.

<sup>1</sup>[https://github.com/derlehner/IndoorAirQuality\\_DigitalTwin\\_Exemplar](https://github.com/derlehner/IndoorAirQuality_DigitalTwin_Exemplar)

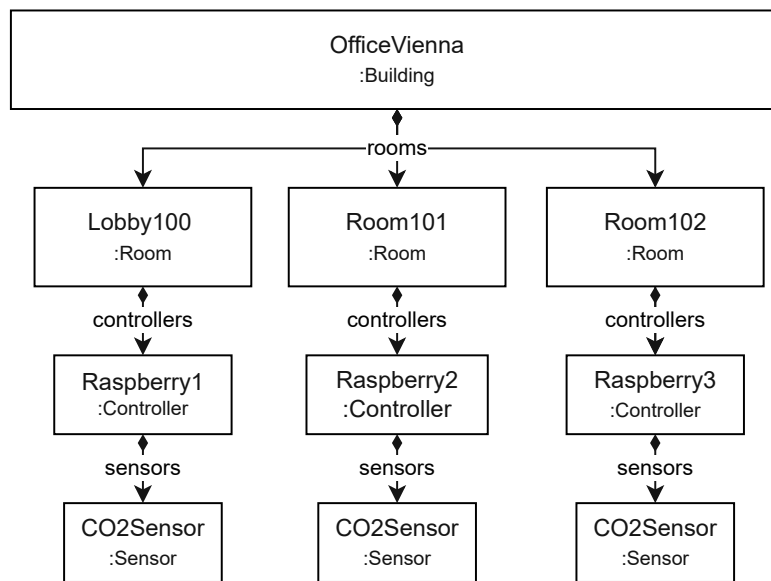


Figure 5.2: **Predefined digital twin** setup in UML object diagram notation

The basic documents and software artifacts were taken from the "Indoor AirQuality Digital Twin"<sup>2</sup> example, and adapted, improved, and extended for this use case.

## 5.2.2 Settings, evaluation cases, and metrics

In this subsection gives an overview on the evaluation settings, the evaluation cases and the metrics which are used to evaluate the automation potential of digital twin deployment.

### 5.2.2.1 AML4DT and pipeline settings

To investigate the required effort for setting up a digital twin environment and evaluate RQ1 and RQ2, the following two setting were deployed and compared.

The **AML4DT setting** is based on the *AML4DT* framework proposed by Lehner et al. [LSV<sup>+</sup>21] where the input AML model needs to be processed by an ATL transformation using Eclipse. The resulting DTML file needs to be further processed by the Xtext/Xtend transformation to produce Azure DTDL type and instance documents. Next the DTDL types need to be uploaded to the *Azure Digital Twin* service, followed by the creation of each digital twin. Finally, the relationships between the digital twins are set up. To set up and connect the physical devices to the Digital Twin, the setting is based on the implementations from *Indoor AirQuality Digital Twin*<sup>3</sup>, where a Python based Raspberry Pi controller implementation and an Azure Functions application is provided. This implementation is prototypical and does not by default support multiple devices

<sup>2</sup>[https://github.com/derlehner/IndoorAirQuality\\_DigitalTwin\\_Exemplar](https://github.com/derlehner/IndoorAirQuality_DigitalTwin_Exemplar)

<sup>3</sup>[https://github.com/derlehner/IndoorAirQuality\\_DigitalTwin\\_Exemplar](https://github.com/derlehner/IndoorAirQuality_DigitalTwin_Exemplar)

or an external sensor configuration. Using these software components, the controller is connected to the Azure IoT Hub using a predefined (hard-coded) connection string. To be able to connect to the IoT-Hub, the device first needs to be registered, and it has to be supplied with the connection string that was created by the IoT Hub. Necessary aligned property-name mappings between device and digital twin need to be registered at the device as well as in the Azure Functions application. After adding the name mappings to the code of the Azure Function application, it needs to be deployed to the cloud infrastructure. Finally, the prepared device controller code needs to be deployed to each an every device and executed there.

Using the pipeline implemented in this work (**pipeline setting**) the *AML4DT* approach and its artifacts are optimized and automated using the in this work implemented pipeline. First, the build-pipeline (Continuous Integration) needs to be created and configured based on the setup file published in this work. The pipeline is linked to a certain Git repository and automatically triggers on a change. Necessary build-artifacts need to be manually added to the pipeline to be able to execute it. Changes to the input documents need to be made to target certain *Azure Digital Twin* service and *Azure IoT Hub* resources. Next, the build pipeline (Continuous Deployment) can be created based upon documents published in this work. After these one-time tasks are done, the input documents need to be adapted to identify physical devices within it, and add deployment credentials and the aligned DT-to-device naming mappings to each of them.

#### 5.2.2.2 Evaluation cases

Based upon these settings, the following evaluation cases are performed to investigate the required effort for initial deployment as well as changes to the system over time. All evaluation cases are evaluated for RQ1 and RQ2 based on different steps.

- *Evaluation Case 1 (EC1): Deploy predefined DT.* The predefined digital twin (described in Figure 5.2) is used and deployed to the Azure Digital Twin service and physical devices.
- *Evaluation Case 2 (EC2): Adding a new raspberry.* This case builds upon EC1 and a new air quality controller (Raspberry4) is added to the system, holding a CO<sub>2</sub> sensor. The new controller is added to an existing room (Room101). Figure 5.3a shows the adapted UML object/class diagram of the digital twin used in EC2.
- *Evaluation Case 3 (EC3): Moving a raspberry.* The third case adapts EC2 by moving an existing controller (Raspberry4) from one room (Room101) to a new room (Restaurant). Figure 5.3b shows the adapted UML object/class diagram of the digital twin used in EC2.

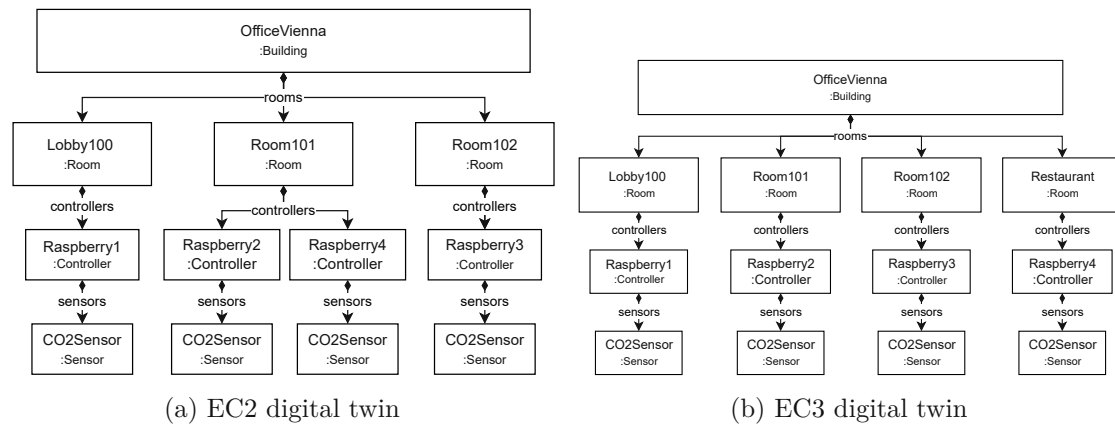


Figure 5.3: Adapted digital twins for Evaluation Case 2 (EC2) and Evaluation Case 3 (EC3) in UML object diagram notation

### 5.2.2.3 Evaluation metrics

To evaluate the manual effort necessary to deploy the described evaluation cases, three metrics are used which are measured in the *manual actions* unit. A *manual action* represents any task that has to be executed by the operator by hand. *Manual actions* might not represent the real count of activities an operator has to perform, but are rather grouped to represent a certain functionality. e.g., the manual action "transform AML model to DTDL" counts as 2 actions, as two transformations, both the ATL and then the Xtext transformation, need to be executed, but ignores the fact that transformations cannot be executed within one click. In general, *manual actions* are counted as if an expert user executes them frequently and knows how to execute them efficiently.

The *infrastructure actions* metric summarizes all actions taken to configure specific infrastructure and services necessary for a certain setting. This can be one-time but also recurring actions that might be done during design or deploy-time. This metric only counts actions that are specific to a defined setting. General infrastructure actions all settings rely on, are not taken into consideration.

Metric *model manipulations* describes the changes necessary to the input model for the pipeline to process it correctly. These are mostly meta-information needed for deployment. Model manipulations might be necessary just a single time in a systems' lifetime, or be coupled to a certain action that is done within the input documents (evaluation). All model manipulation changes are done during design-time, so prior to any build or deployment actions.

The metric *deployment actions* count all actions that are done during the "build-time" or "deploy-time" and don't fit into the two previously described categories. These actions are repetitive actions that need to be executed with every deployment.



### 5.2.3 Results

In the following, the results of the case study is presented. First the results regarding RQ1 are in detail described for both the **Pipeline** and **AML4DT** settings, followed by the results of the RQ2 evaluation. The results are summarized and presented in Table 5.2.

Setting	Pipeline			AML4DT		
	infra	model	deploy	infra	model	deploy
<b>RQ1 (Deploy digital twin)</b>						
EC1: Deploy initial setup	3	1	0	0	0	15
EC2: Adding a new raspberry	0	0	0	0	0	5
EC3: Moving a raspberry	0	0	1	0	0	3
<b>Sum RQ1</b>	<b>3</b>	<b>1</b>	<b>1</b>	<b>0</b>	<b>0</b>	<b>23</b>
<b>RQ2 (Connect &amp; deploy physical devices)</b>						
EC1: Deploy initial setup	0	8	0	6	0	18
EC2: Adding a new raspberry	0	2	0	2	0	6
EC3: Moving a raspberry	0	0	0	0	0	0
<b>Sum RQ2</b>	<b>0</b>	<b>10</b>	<b>0</b>	<b>8</b>	<b>0</b>	<b>24</b>
<b>Sum total</b>	<b>3</b>	<b>11</b>	<b>1</b>	<b>8</b>	<b>0</b>	<b>47</b>

Table 5.2: Count of *manual actions* necessary to complete evaluation cases in *infrastructure actions* (infra), *model manipulations* (model) and *deployment actions* (deploy)

#### 5.2.3.1 Results for RQ1

##### AML4DT setting

In the *AML4DT* setting, no *infrastructure actions* (**0**) are necessary as according to the previously defined metrics (see Section 5.2.2.3) only for this setting specific actions are counted. The necessary *Azure Digital Twin* service instance and other infrastructure setup is created up front and therefore not counted towards *infrastructure actions*. No (**0**) *model manipulations* are necessary either as the model that is going to be used is the predefined digital twin described in Section 5.2.1 which does not need any adaptations. As a preparation for the deployment the AML model needs to be transformed into the necessary DTDL documents using the ATL and Xtext/Xtend transformations (2 actions). After that all types need to be uploaded to the *Azure Digital Twin* service (1 action) just then the necessary instances can be created. Creating each systems' entity needs 1 manual action (9x1 actions). While entities that are contained within another (sensor in air quality controller) are directly referenced to their parent, the 3 air quality controllers need to be referenced to the rooms separately which needs another 1 action each (3x1 actions). All these actions together total **15 deployment actions** for the *AML4DT* setting evaluating *EC1*.

When evaluating *EC2*, no *infrastructure actions* (**0**) (no changes to the infrastructure necessary) or *model manipulations* (**0**) besides the general adaption of EC2 are necessary. For a successful deployment, transformations need to be executed (2 actions). No new types were defined therefore a new upload is not necessary. The new air quality controller as well as its sensor needs to be created (2 actions). Finally, the new air quality controller can be referenced to the containing room (Room101) (1 action), totaling in **5 deployment actions**.

The evaluation of *EC3* results in **3 deployment actions** dividing in the creation of a new room (1 action), removing the relationship between the air quality controller (1 action) and room and the creation of a relationship between the new room and the air quality controller (1 action). Changes to infrastructure or model manipulations are not necessary.

### Pipeline setting

Before being able to use the pipeline for *EC1* the pipeline needs to be set up. For this task **3 infrastructure actions** and **1 model manipulation** are necessary. As described, the repository needs to be complemented by necessary pipeline scripts and applications (1 action) before the build (1 action) and release (1 action) pipeline is created. Finally, the input model is extended to hold a reference to the *Azure Digital Twin* service (1 action). During deployment the pipeline makes any manual actions obsolete resulting in **0 deployment actions**.

*EC2* evaluation leads to no (**0**) more *model manipulation* actions, besides the general model adaption of EC2. No (**0**) *infrastructure actions* need to be executed either, as the deployment script redefines all existing digital twin instances and create the new entities. **0 deployment actions** are needed since the pipeline takes over any steps during the deployment.

When evaluating *EC3*, some *deployment actions* need to be executed before the deployment is done. As the deployment script is not capable of deleting or adapting existing relationships, either the single relationships or all whole digital twin instances need to be removed (**1 action**) prior to its execution. No more (**0**) *model manipulations* are necessary as all adaption was already done in EC1. No (**0**) *infrastructure actions* are necessary as the pipeline takes over all tasks during the deployment.

#### 5.2.3.2 Results for RQ2

##### AML4DT setting

Using the *AML4DT* setting, devices need to be identified first. In the given scenario each "controller" represents a physical device needing deployment. For the controllers to be able to connect to the cloud infrastructure, they need to be registered at the IoT Hub (1 action) and the associated connection string needs to be extracted (1 action) first. These 2 actions count towards the *infrastructure actions* totaling in **6 actions**. The extracted IoT Hub connection string needs to be added to the device code (1 action), as well as the name mapping between the devices' sensor values and DT properties (1 action).

These mappings also need to be made available to the cloud application (1 action). After the setup is done, the cloud application (1 action) and the physical devices need to be deployed (1 action) and the software needs to be executed (1 action) on the devices. For the given use case **6** *deployment actions* per controller have to be performed, resulting in **18** *deployment actions* when evaluation *EC1*. No *model manipulations* are necessary using this setting.

The evaluation of *EC2* results in another **2** *infrastructure actions* including the registration of the new device at the IoT Hub and the connection string extraction, **0** *model manipulations* (no changes to the model necessary besides the general *EC2* change) and **6** *deployment actions* as just as for *EC1* adaptations to the devices and cloud applications code need to be made, and both applications need to be deployed to their according targets.

Executing the use case for the *AML4DT* setting and *EC3* results in **0** *infrastructure actions*, **0** *model manipulations* (no changes to the model necessary besides the general *EC3* change) and **0** *deployment actions*. No changes are necessary as the "room" layer used in the scenario does not have any influence on the software running on the device or in the cloud infrastructure.

### Pipeline setting

Before the created pipeline can be used to automatically create devices at IoT Hub devices and connection strings are extracted some up-front settings need to be made to the input files. To determine a certain Azure IoT Hub instance, a *AzureResource* must be set to the specific instance the devices should be created at (1 action). Furthermore, all physical devices need to be identified and marked (1 action) for automatic processing. Next, for each device the DT property to device sensor values mapping (1 action) needs to be added to the input documents as well as deployment details like connection details (1 action). These tasks are all count towards *model manipulations* and total at **8** for *EC1* - 2 actions up front plus 2 per physical device. No (0) *infrastructure actions* and (0) *deployment actions* are necessary to achieve the goal of *EC1* as the pipeline is already set up and takes over all tasks necessary for the deployment.

As evaluating evaluation case 2 (*EC2*) is building upon *EC1*, only the newly added controller needs to be configured. Just like in *EC1*, the input documents need to be adapted to contain the DT property mapping (1 action) and deployment information (1 action) for the new device. After these **2** *model manipulations* no more (**0**) *infrastructure actions* or (**0**) *deployment actions* are necessary. No *infrastructure actions* and *deployment actions* are needed since the pipeline is already set up and takes over any steps during the deployment.

Just like in the *AML4DT EC3* evaluation, no efforts are necessary in the *Pipeline EC3* evaluation. The already in *EC2* defined property mappings are untouched as long as no names changes, which is assumed. No *infrastructure actions* and *deployment actions* are needed since the pipeline is already set up and takes over any steps during the deployment.

## 5.3 Evaluation of Scalability of Monitoring Execution Time

This section is devoted to providing information about the scalability of the created artifacts in regard to modeled conditions and associated actions. The scalability is evaluated based on two experiments which measure execution times of monitoring rules on data exchanged between physical device and Digital Twin. The first experiment will measure the execution times of the monitoring rules on the physical device, the second experiment measures the execution times of the monitoring rules in a cloud application. Separate experiments are necessary as both tested monitoring rule engines differ in their setup and need to be executed in different ways. In Section 5.3.1 the experiments' scenario and parameters are described. The two experiments are conducted using the identified parameters and their effect on the performance of the conditions and actions framework used on a physical device (Section 5.3.2) and in a cloud application (Section 5.3.3).

### 5.3.1 Scenario parameters and setup

This section discusses parameters influencing the execution time or message throughput (how many messages can be processed per second) of the conditions and actions framework implemented in Section 4.3. As execution times of simple monitoring rules can reach sub-millisecond can be hardly measured, it is better to measure in *message throughput* rather the pure execution time. Message throughput and execution time are negatively correlated and can therefore be calculated from each other value. To be able to identify parameters influencing the performance, the physical devices software and the cloud application running the monitoring framework needs to be analyzed.

Two different experiments running the same conditions, the same data and therefore the same parameters should be used to evaluate the message throughput. For these experiments two different applications are built, one application executing monitoring rules on the physical device, one application for the cloud based monitoring rule execution. To ensure the comparability between both application each data record sent from a device to the digital twin or rather the cloud application always holds a single data entry indicating the device and sensor name, the sensor value name and the sensor value itself. This implies that a condition may only target a single sensor and sensor value at a time as already described in Section 4.3. This means that for a condition to be evaluated, it must target the sensor of currently processed message (e.g., device "Rasp1"s sensor "DHT11" yielded a new "temperature" value - a condition targeting "Rasp1"s "CCS811" sensor "temperature" is not evaluated). All conditions that target the according sensor are evaluated regardless of the sensor value targeted (e.g., device "Rasp1"s sensor "DHT11" yielded a new "temperature" value - a condition targeting "Rasp1"s "DHT11" sensors "humidity" value is evaluated). All associated actions are executed as soon as the condition evaluates positively and therefore to test the performance influence of actions, at least one (1) condition must be met.

This leads to 3 parameter variations that influence the throughput of the monitoring framework:

- The number of conditions not applicable based on the processed message. Varying this number gives an indication on how well conditions are filtered based on the targeted sensor.
- The number of conditions applicable based on the processed message. This number gives an indication on how well the conditions' execution time scales.
- The number of actions associated with a condition. As actions are only interpreted and executed if a certain condition evaluates to true the only meaningful variation is varying actions on a single positively evaluated condition.

To be able to scale conditions and actions for a given test-set, one condition and one action are created and manually duplicated (copied and pasted) to create the defined amount necessary for the test-set. The created test-sets are used in both experiments to show the general usability of the selected format. The designed condition, visible as an example in Listing 5.1, queries the given message structure for a *temp* property with a value higher than 1. The example data used only provides values higher than that which in this case always triggers the associated actions. The used action makes use of the *DelayActionImpl* which imitates a generic IO operation. A full example of a test-set is given in the following listing. It shows the *0-1-10* test-set containing zero non-applicable conditions, one applicable condition and ten actions.

### 5.3.2 Experiment I - Scalability of monitoring framework on an IoT device

The impact of the identified parameters on the throughput on the physical device is evaluated by performing a simulation experiment. The latter described computers run the in Section 4.2 described device controller implementation to gather sensor data, process them using the monitoring framework and finally send them to the digital twin. Subsequently, described adaptations have been made to the application to be able to examine its raw performance.

To simplify the setting, the experiment gathers a single sensors' value only. Based on the device controller implementation described in Section 4.3, it handles multiple sensors one after another. The same is true for multiple values of a sensor. For this experiment the values from the sensors are gathered continuously, so as soon as a message is done processing another sensor value is gathered and processed. Therefore, adding further sensors or sensor values does not influence the performance of the framework. Therefore, a single device *Rasp1* is configured to continuously gather its sensor *TestSensors'* property *temp* and send it to its digital twin. As no physical hardware was available at the time of testing and also to eliminate external influence, the application is configured to use the implemented *CsvValuesSensor* test-sensor as the *TestSensor's* implementation and

Listing 5.1: 0-1-10 test-set used in experiment I and II

```

{
  "validators":{
    "TestSensor": [
      {"jsonPathQuery": "$[?(@.Property=='temp'&&@.Value>=1)]", "continue": true, "
      ↪ actions": [
        {"type": "DelayActionImpl", "config": {"delayInMilliseconds": 10}},
        {"type": "DelayActionImpl", "config": {"delayInMilliseconds": 10}},
        {"type": "DelayActionImpl", "config": {"delayInMilliseconds": 10}},
        {"type": "DelayActionImpl", "config": {"delayInMilliseconds": 10}},
        {"type": "DelayActionImpl", "config": {"delayInMilliseconds": 10}},
        {"type": "DelayActionImpl", "config": {"delayInMilliseconds": 10}},
        {"type": "DelayActionImpl", "config": {"delayInMilliseconds": 10}},
        {"type": "DelayActionImpl", "config": {"delayInMilliseconds": 10}},
        {"type": "DelayActionImpl", "config": {"delayInMilliseconds": 10}},
        {"type": "DelayActionImpl", "config": {"delayInMilliseconds": 10}},
        {"type": "DelayActionImpl", "config": {"delayInMilliseconds": 10}}
      ]
    ]
  }
}

```

use the in Section 5.2.1 described `sensorData.csv` instead. Based on this the following input parameters are varied to measure the impact on the systems' throughput: (1) For the number of conditions: (1) the number of conditions not targeting "TestSensor" (not applicable), (2) the number of conditions targeting "TestSensor" (applicable). (2) The number of actions associated with every "TestSensor" targeting condition.

The device controller implementation used to evaluate the functionality of message transfer uses the *AzureIoTHubService* to transmit messages to the Azure IoT Hub. The resulting IO (input-output) communication process can influence the measured performance significantly and therefore the service is replaced with the service *DelayedTelemetryService* using a static delay of 10 milliseconds instead. This means that the throughput per second should be close to 100 messages per second when not evaluating any conditions. This value is compared to the real world value using 0 conditions, ramping up to 100 conditions in steps of 10. The same set of data is used when varying the condition associated actions. Table 5.3 shows the used parameters in detail.

To test the general performance impact of executing actions, a single action has to be executed each and every time. Using a real action, communicating to external systems, results in unpredictable performance. To eliminate external factors a simple "DelayAction" is introduced which delays the execution by 10 milliseconds mocking an external communication. In all the following experiments this action is used to measure the performance impact.

To measure the amount of processed messages, the application is extended by a simple counter that is increased for each message processed. 10 minutes after being launched,

Name of setting	# conditions non-applicable	# conditions applicable	# actions in applicable conditions
vary non-applicable conditions	10-100	0	0
vary applicable conditions	0	10-100	0
vary actions	0	1	10-100

Table 5.3: Parameters used to evaluate the monitoring frameworks performance on the physical device

the program is manually terminated and outputs the counter as well as the exact time it was running. These values are used to determine the throughput in messages per second. Each setting is repeated for 5 times to gather reliable data. Additionally, the experiment is executed on two different hardware setups to gather information on the vertical scalability of the application. As no *IoT* hardware is available two different but more powerful system setups are used. Hardware setup 1 (*Desktop*) runs an Intel® Core™ i7-8700 CPU and 32 GB DDR4 RAM memory. Setup 2 (*Notebook*) runs an Intel® Core™ i7-3517U CPU and 10 GB DDR3 RAM memory.

### 5.3.2.1 Results analysis

To analyze the results of the simulation runs, a specific procedure is used for each setting: (1) The box plot notation is used to visualize the message throughput of each value over 5 simulation runs. All settings box plots are displayed in a single graph. The results of each device the experiments were done are presented in separate plots. This visualization is used to (1) check plausibility of the results in general and (2) get an impression on the influence of the parameter on the throughput. (2) Two functions are constructed and approximated for each setting and device to describe the relationship between the size of the varied parameter and the measured execution time. The first function uses linear approximation by using two coefficients, one representing the slope and the other representing the intercept. The second function approximates the real data using a third coefficient representing the squared slope. (3) The functions are analyzed to find a formulation of the impact of the particular varied parameter on the throughput of the conditions and actions framework. The two functions are compared based on the root mean squared error (RSME) value. The function that produces better approximation based on the given data, is used to represent the prediction. If the RSME values equal, the linear function is chosen as it provides reduced complexity.

When varying the number of non-applicable conditions (simulating setting *vary non-applicable conditions*), so conditions that are not getting evaluated as the target property differs to the message that is being processed, the following results can be obtained. As shown in the Figures 5.4 the simulation yields very little to no effect of this parameter

Simulation setting	Notebook		Desktop	
	RMSE linear	RMSE squared	RMSE linear	RMSE squared
vary non-applicable conditions	<b>0.14</b>	0.14	0.05	<b>0.03</b>
vary applicable conditions	0.4	<b>0.29</b>	1.89	<b>0.89</b>
vary actions	1.32	<b>0.7</b>	0.88	<b>0.47</b>

Table 5.4: List of simulation settings with the associated RMSE values

on the total throughput. In case of the *Notebook* hardware this assumption is backed by a linear regression line with a slope of

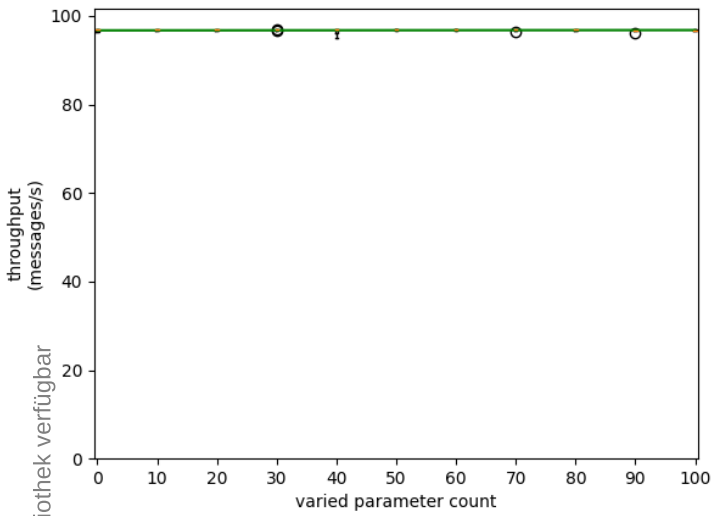
$$5.6 * 10^{-4} * x$$

where  $x$  is the number of conditions. The linear regression as well as the quadratic approximation function resulted in a RMSE of 0.14, this leads to selecting the linear regression in the case of the *Notebook* simulation. While the first simulation resulted in values close to the mathematical possible 100 messages per second, running the experiment on the *Desktop* hardware which runs significantly newer and better hardware results in very poor values. While the values behave somewhat similar to the previous run in relation to varying the number of conditions, the overall results are lower by one third and are also more spread. As this result is surprising, the experiment was repeated after restarting the machine but the same results yielded. Because of the spread values, the RSME of the linear approximation is 0.05 and therefore higher than an approximation with 3 coefficients resulting in 0.03. The 3-coefficient regression line is used which uses

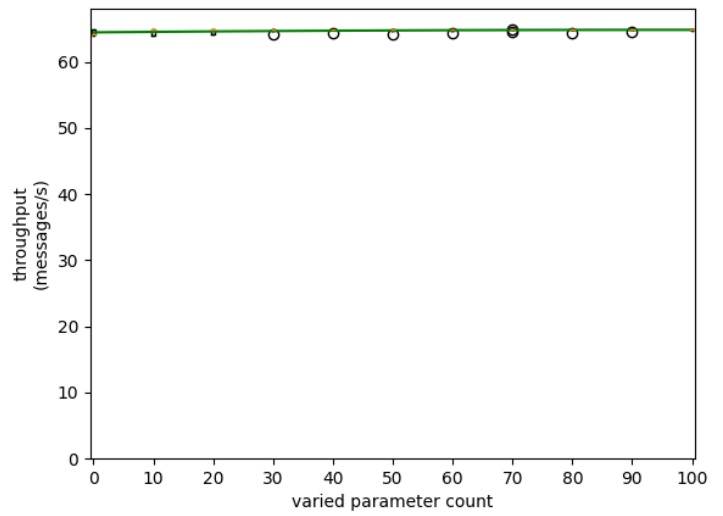
$$-4,1 * 10^{-5} * x^2 + 7,8 * 10^{-3} * x \quad (5.1)$$

as slope. As in the *Notebook* experiment the influence of the tested parameter is very low and can be mostly neglected in relation to the execution time or throughput. More information on possible issues for the lower message rates on the more powerful hardware can be found in the *Discussion* Section 5.4.





(a) Hardware: Notebook



(b) Hardware: Desktop

Figure 5.4: Experiment I: visualization of simulation runs for *vary number of non-applicable conditions*

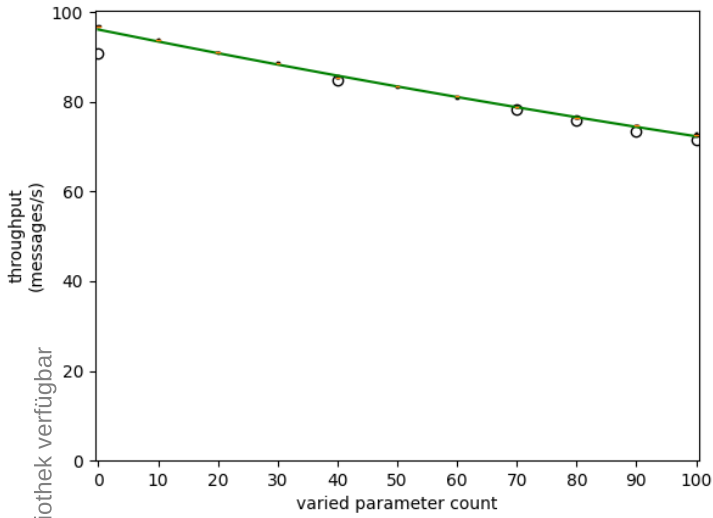
An analysis of the simulation setting *vary applicable conditions* produces the following results. The plots shown in Figure 5.5 show the graphical results for both hardware settings. Both plots show a significant influence of the varied parameter on the execution time. This is backed by the regression lines. For the different hardware settings Table 5.3 shows the different RMSE values. Both settings use a quadratic estimation function with slope

$$-1.9 * 10^{-3} * x^2 + 8.8 * 10^{-2} * x \quad (5.2)$$

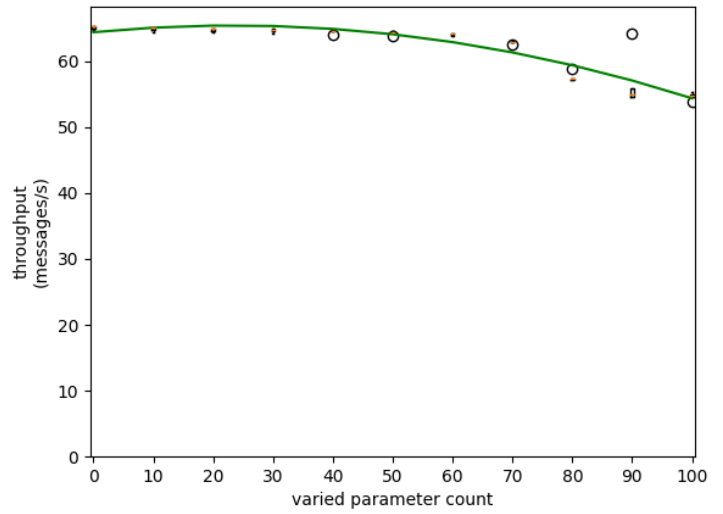
for the Desktop setting and

$$3.1 * 10^{-4} * x^2 - 2.7 * 10^{-1} * x \quad (5.3)$$

for the Notebook setting as the RMSE is lower compared to the linear value. While the linear approximation is very close using the *Notebook* hardware, the *Desktop* hardware again shows very spread results that definitely show a more than linear impact. As the *Desktop* hardware already showed weird results in the first setting, the analysis focuses more on the *Notebook* results. It can be said that there is a significant impact on the performance, but the influence is still linear.



(a) Hardware: Notebook



(b) Hardware: Desktop

Figure 5.5: Experiment I: visualization of simulation runs for *vary number of applicable conditions*

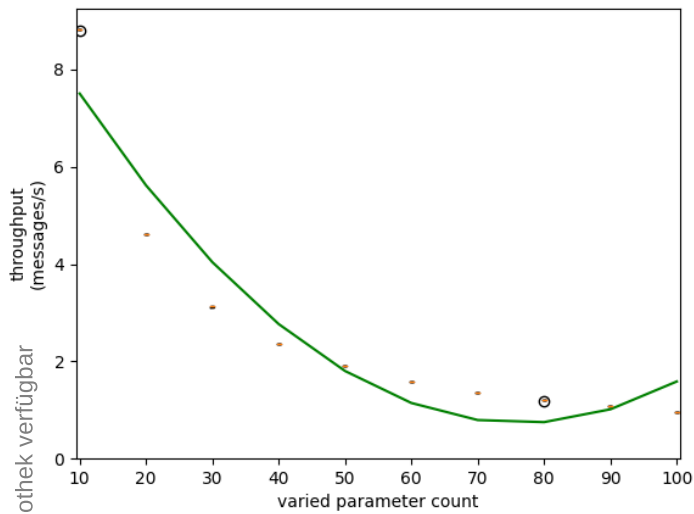
The simulation of setting *vary actions* yields the following results. Varying the number of actions that are executed significantly impacts the overall performance. Both hardware settings show comparable results with quadratic estimation functions. The impact itself very much depends on which action is triggered. Evaluating the impact besides the used *DelayAction* based on the *Notebook* hardware, shows that the average execution time with no conditions and no actions was  $10.34ms$ . Subtracting the 10 milliseconds delay that are caused by the replacement of *AzureIoTHubService* this results in a delay-adjusted execution time of  $0.34ms$  per message. Doing the same calculations for each set of the *vary actions* setting, results increase by an average of  $0.31ms$  per executed action when neglecting all introduced delays. Table 5.5 shows an excerpt of the detailed calculations for 1 condition and  $n$  actions. The plots for both devices with the according regression lines are shown in Figure 5.6 and show the exponential impact of actions on the overall message throughput. The slopes for Desktop respectively Notebook setting are as follows:

$$1 * 10^{-3} * x^2 - 1.6 * 10^{-1} * x \quad (5.4)$$

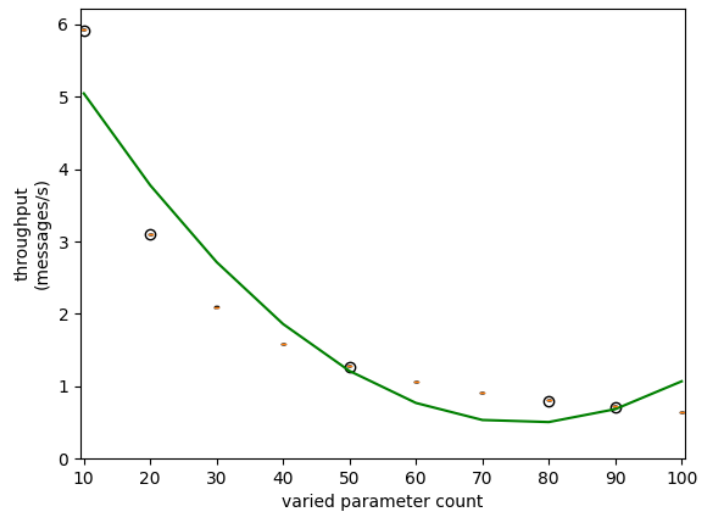
for the Desktop setting and

$$1.5 * 10^{-3} * x^2 - 2.3 * 10^{-1} * x \quad (5.5)$$

for the Notebook setting.



(a) Hardware: Notebook



(b) Hardware: Desktop

Figure 5.6: Experiment I: visualization of simulation runs for *vary number of actions*

# actions	Throughput (msg/s)	Execution time (ms)	delay-adjusted execution time (ms)	Influence of action (ms/action)
0	95.45	10.48	0.48	n.a.
10	8.82	113.40	3.40	0.34
50	1.90	526.32	16.32	0.33
100	0.96	1041.67	31.67	0.32

Table 5.5: Calculation details for delay-adjusted execution-time influence of actions based on *Laptop* hardware results

### 5.3.3 Experiment II - Scalability of monitoring framework in a server-less cloud application

The impact of the identified parameters on the message-throughput based on the *Monitoring Service* application in the cloud is evaluated by performing a second simulation experiment. The application is deployed to Azure Functions using the "consumption" plan. To produce measurable data a single physical device is adapted and used to send 10 updates continuously. The client *Rasp1* is configured to gather its sensor *TestSensors*' property *temp* and send it to its digital twin. As no physical hardware was available at the time of testing and to eliminate external influence, the client is configured to use the *CsvValuesSensor* test-sensor as the *TestSensor*'s implementation and use data from *sensorData.csv*.

Name of setting	# conditions non-applicable	# conditions applicable	# actions in applicable conditions
vary non-applicable conditions	10-100	0	0
vary applicable conditions	0	10-100	0
vary actions	0	1	10-100

Table 5.6: Parameters used to evaluate the monitoring frameworks performance in the server-less cloud application

The throughput is calculated by using the cloud applications' execution times of all 10 messages as presented by the *Azure ApplicationInsights* service. These execution times are compared to the total time taken to process all 10 updates by analyzing the first and last timestamp of the series. This is done to determine the parallelism of the application. Based on the messages produced by the *Rasp1* client the following input parameters are varied to measure the impact on the systems' throughput: (1) For the number of conditions: (1) the number of conditions not targeting "TestSensor" (not applicable), (2) the number of conditions targeting "TestSensor" (applicable). (2) The number of actions associated with every "TestSensor" targeting condition.

Due to the nature of the hosting of the application a different strategy then before has to be followed. Cloud native applications come with advantages but also with disadvantages such as that the applications get torn down after not being in use for a while. This leads to the need of a "warm up". More specifically the function needs to be called a few times before the test starts.

To be able to focus on the performance of the condition evaluation and actions execution, the component to transform and forward the received messages to the digital twin service is replaced by a 10 milliseconds delay. This means that the throughput per second should be close to 100 messages per second when not evaluating any conditions. This value is compared to the real world value using 0 conditions, ramping up to 100 conditions in steps of 10. The same set of data is used when varying the condition associated actions. Table 5.6 shows the used parameters in detail.

To test the general performance impact of executing actions, a similar solution to the one used in experiment I is used which replaces the real action by a "DelayAction" which delays the execution by 10 milliseconds.

### 5.3.3.1 Results analysis

To analyze the results of the simulation runs a similar process to before is used. Once more the box plot notation is used to visualize the message throughput of each setting over 10 messages. All box plots are presented in a single graph. The throughput is

Simulation setting	RMSE linear	RMSE squared
vary non-applicable conditions	6.80	<b>5.49</b>
vary applicable conditions	6.19	<b>6.17</b>
vary actions	1.03	<b>0.5</b>

Table 5.7: List of simulation settings with the associated RMSE values for experiment II

calculated for each result using the following formula  $message\ throughput\ per\ second = 1000 / execution\ time\ in\ milliseconds$  on the gathered data records. Again functions to describe the relationship between the size of the varied parameter and the measured results are constructed. This relationship is represented based on the execution time of each run and not the total execution time of  $n$  runs and therefore ignores parallelism. Linear approximation is compared to squared approximation using the root mean squared error (RMSE). The function that provides the better result is added to the box plot diagram. If linear and quadratic approximation yield the same RMSE the linear function is chosen.

After the analysis of the influence of the varied parameter itself, an additional analysis regarding parallelism is conducted. To support argumentation a horizontal bar chart is used to visualize starting times and execution times of each of the 10 executions of the setting with the biggest parameter number.

Table 5.7 shows the yielded RMSE values for the given settings. The bold written value indicates the littler error and therefore better approximation. This approximation is used in the according diagram presented beneath.

When varying the number of non-applicable conditions (simulation setting *vary non-applicable conditions*), more precisely conditions that are not evaluated as the targeted property differs to the one contained in the processed message, the following can be obtained. As shown in the Figure 5.7 the simulation yields, although results are very spread, that there is no visible effect of this parameter on the total throughput. This is backed by the squared approximation results yielding a slope of  $0.004545 * x^2 - 0.4837 * x$  which shows a decrease in performance from parameter size of 0 until 50, but increases afterwards again. With an RMSE of 5.49 the squared regression offers a better approximation than the linear and is therefore chosen as representative.

Comparing the results to those from experiment I, it can be seen that there are more and worse outliers even though extensive preparation was carried out. The pure throughput is lower than on the physical devices which is not surprising given the not very representative hardware used in the first experiment. More detailed discussion on the comparison between local and cloud execution is conducted in Section 5.4.

An analysis of the simulation setting *vary applicable conditions* produces the following results. The plot shown in Figure 5.8 show the graphical results. Again results are widely

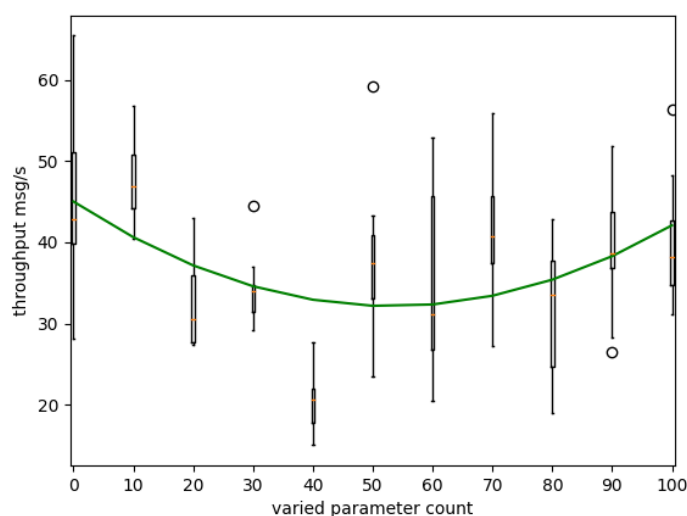


Figure 5.7: Experiment II: visualization of message throughput of *vary non-applicable conditions* on the cloud application

spread with outliers, but closer than in the execution of the first setting. Comparing the results to the first setting a very similar average execution time of all results can be obtained (29.18 msg/s in setting *vary non-applicable conditions* to 29.74 msg/s in the second setting). The results show no influence of the parameter on the runtime. This is backed by the regression line. The slope of the used regression line is  $0.0006231x^2 - 0.06802 * x$  which is almost linear. Nevertheless, the setting uses a quadratic estimation function as the RMSE is lower. In comparison to experiment I, it is very interesting to see that there is no influence, although a comparable logic is in place. The lookup and execution of JSONPath-based conditions seems to be better implemented and works with less/not measurable influence based on the tested numbers.

The simulation of setting *vary actions* yields the following results. Varying the number of actions that are executed impacts the overall performance. The executed setting shows comparable results with quadratic estimation functions. The approximated function yields the smallest RMSE of the test series (0.5) which indicates a good assumption. The plot with the according regression line is shown in Figure 5.9 and shows the exponential impact of actions on the overall message throughput. This is not surprising as every action adds another 10 milliseconds delay when executed.

Finally, the parallelization of the execution is analyzed by comparing the execution start and end times of each of the settings' biggest run. More precisely this is the run having the highest parameter count of the variation. So for the *vary non-applicable conditions* setting this is 100-0-0 execution run regarding parameter sizes listed in Table 5.6. In the *vary applicable conditions* setting the 0-100-0 and for the last setting *vary actions* the

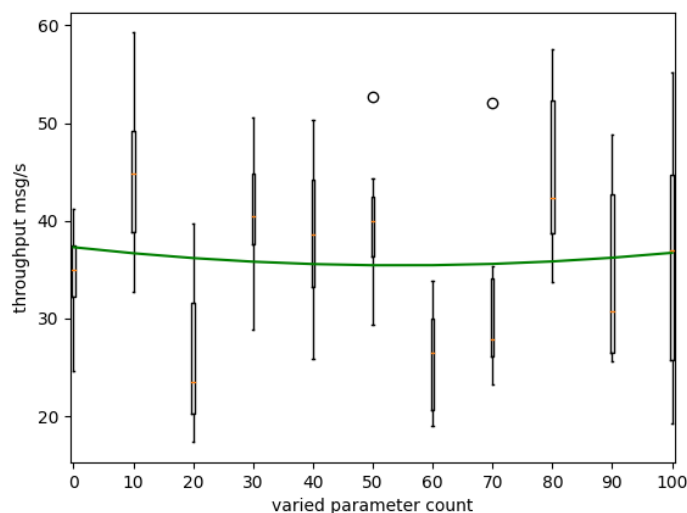


Figure 5.8: Experiment II: visualization of message throughput of *vary number of applicable conditions* on the cloud application

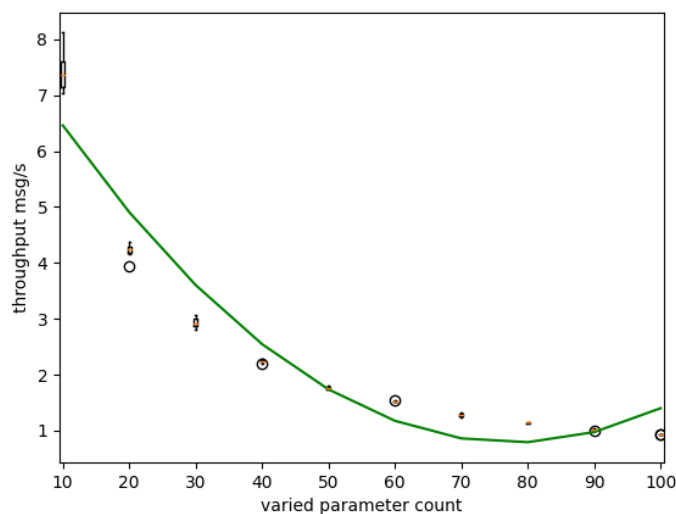
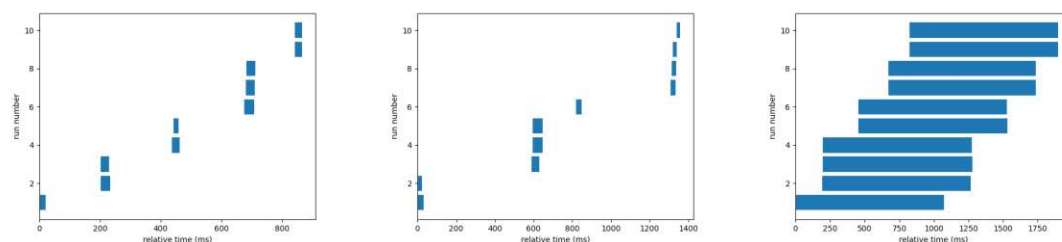


Figure 5.9: Experiment II: visualization of message throughput of *vary number of actions* setting on the cloud application

0-1-100 run is shown in Figure 5.10 and analyzed in the following.

Based on the data presented in the first diagram, it can be seen that the Azure platform creates multiple instances of the application and uses those to parallel processes messages



(a) 100-0-0 in setting *vary non-applicable conditions* (b) 0-100-0 in setting *vary applicable conditions* (c) 0-1-100 in setting *vary actions*

Figure 5.10: Experiment II: visualization of relative execution start time and execution time

in every setting. The time span it takes to schedule a new instance seems to be around 200 milliseconds in optimal situations. Based on the 10 messages sent by the client no more than 4 instances were scheduled in parallel. The first diagram also shows the inefficiency of the horizontal scaling of the platform and the function implementation it-self. An application continuously processing messages could process the messages faster. The *0-100-0* setting presented in the second diagram shows that starting additional instances can even take up to 600 milliseconds and up to 4 parallel instances can be scheduled. Only the last setting (*0-1-100*) shows the advantage of parallelization as the execution times grow up towards 1 second for each run and all messages get processed in parallel at some point in time. Instead of an execution time of  $10 * 1second$  the total execution time is only about  $1.8seconds$ . More details are discussed in the next chapter when comparing the results against experiment I.

## 5.4 Discussion

In the first chapter, three research questions were stated. The goal of this subsections is to give an answer to each of those based on data gathered from the software artifacts created in this thesis.

**RQ1: *To what extent can MDE techniques be used to reduce the manual effort for deploying Digital Twins?*** The use case to determine the automation potential was conducted in Section 5.2.3.1 and yields the following results. All necessary steps can be, with very little preceding efforts, automated using a combination of scripts and applications. In detail these are:

- Transforming existing models into digital twin platform specific documents.
- Uploading necessary type definitions to the digital twin platform.
- Creating all digital twins instances in the digital twin platform.



To enable the automation of above-mentioned tasks, some setup needs to be made preemptive or within the first run.

(1) Identify and specify a certain *Azure Digital Twin* service instance in the AML model and transformation. (2) Add necessary scripts and applications to the repository containing the AML model. (3) Set up a build pipeline to automatically execute the transformations. (4) Set up a release pipeline to automatically deploy data to the selected *Azure Digital Twin* service instance.

Using the described framework and pipeline most of the digital twin deployment can be automated. From **23** actions necessary to be processed in the *AML4DT* setting, only **5** need to be executed using the *pipeline* setting (-78.26%) when considering all 3 evaluation cases. The remaining tasks can be split into two groups. The first one are one-time setup tasks (3 actions) that are needed to take advantage of the pipeline features. These tasks have been eased up as much as possible by providing generic and predefined pipeline definitions but still require manual effort. The second group are repetitive actions that the framework or pipeline currently does not support. Currently, the evolution of models and the according co-evolution of the digital twin are not supported when using the pipeline and therefore make manual actions (2 actions) like the deletion of a digital twin necessary. The given numbers might not be accurate for all possible scenarios but with an evolving scenario the pipeline can be further improved to support more and more cases.

**RQ2: To what extent can MDE techniques be used to reduce the manual effort to connect the physical devices to Digital Twins?** To determine the automation potential of RQ2 a case study executed in Section 5.2.3.2. The results show that most steps can be automated with minor one-time efforts. Extending the implementation from RQ1 the following steps can be fully automated:

- Identify physical devices based on predefined markers and create IoT hub registrations for them.
- Extract connection credentials and serve in machine-readable format.
- Extract mapping for physical device to digital twin components and properties to ensure proper naming.
- Serve connection credentials and name-mapping to device on a configuration injection basis.

To enable the automation for the listed tasks, again prerequisites need to be completed. The necessary tasks are based upon already done tasks for RQ1, if the setup for RQ1 was not yet performed, tasks from both lists need to be executed. Some of the following tasks might only need a one time set up, others need maintenance when certain features change. The frequency of these changes is denoted specifically in the following enumeration.

(1) Identify and specify a certain Azure IoT Hub instance in the AML model or transformation. (2) Mark physical devices within the AML model and transformation. (3) Add physical device to digital twin name-mapping information to AML model and transformation.

Besides the listed tasks, the deployment (i.e., transfer of application and configuration) to the device can technically be done fully automated, but concerns regarding security arise. As IoT devices are rarely monitored and even less updated, opening access through secure shell (SSH) is a very big security problem. Because of that IoT devices are often hidden behind network infrastructure and therefore not accessible from a server situated in the cloud. Although there are solutions ready to make IoT devices available while still maintaining the security, settings like that should still be used carefully and need deeper investigation. Besides accessibility and availability of the remote devices, another concern is the insecure storage of device credentials within the pipelines input data version-controlled within a Git repository. Even with solutions presented in this thesis, it is stated multiple times that automatic deployment to remote locations should not be used without further improvements to the security.

Using the framework and pipeline created in this thesis the number of manual actions necessary to connect and deploy physical devices could be reduced from **32** when using the *AML4DT* setting, to only **10** (-68.75%) considering all 3 evaluation cases. The remaining 10 tasks distribute between configuring deployment information and setting up a valid digital twin to device sensor name mapping. Both the physical devices' and the cloud applications code could be improved making no more code changes necessary when the system changes which reduces the necessary actions drastically.

**RQ3: What is the scalability in execution-time of conditions and actions monitoring runtime data (i) on a physical device compared to (ii) in a cloud application** To compare the scalability in execution-time of the created monitoring framework executed (1) on a physical device and (2) in a cloud hosted server-less application a simulation experiment with 2 parameters varied in 3 ways is executed. In general, it must be said that due to the widespread results of experiment II, which reasons are later discussed, the data produced has to be used with caution and is only of limited value.

The results show no influence of the parameter size when varying the number of not applicable conditions (setting *vary non-applicable conditions*) on the execution time, on either physical device or the cloud application. This implies that the conditions' lookup per sensor, implemented into both applications works good and achieves its goal.

It is obvious that the *vary actions* setting, always produces an impact on the execution time no matter if run in the cloud or locally. The impact besides the IO simulation 10 milliseconds delay is as little as  $0.31ms$  per action based on the data shown in Table 5.5. This implies that the majority of execution time influence is produced by the action itself and shows how important it is to optimize the performance of every action.

The setting when applicable conditions are varied (*vary applicable conditions* setting)

yields different results on the physical devices and the cloud application. While executing JSON-path queries on the physical device impacts the overall performance, the cloud application on the other hand showed no impact given the number of the conditions used. As discussed in the previous section, this indicates that the general execution of JSON-path queries is more performant using the C# implementation than the Python equivalent. As both implementations always execute all conditions no matter if they evaluate positively or not, the libraries used to execute the created conditions make the biggest difference in the created implementations. Another reason for the cloud application to show no measurable influence, might be due to the outliers and general deviation of the results.

Before evaluating outliers and deviation reasons, another aspect that was evaluated is the affect of the used hardware to evaluate conditions. It could be seen that older hardware (Laptop hardware) performed significantly better than newer and more powerful hardware. This is probably a problem of power-optimization, where the modern processor reduced CPU clock speed to reduce power consumption which resulted in less message throughput. Comparing this results to the unknown hardware used in the cloud setup, the *Desktop* hardware still produced 180.81% the throughput but was still far from the throughput the Laptop hardware was able to achieve (269.84% more msg/s). Another important difference is, that the cloud setup always queues each and every message that is sent by the client and can scale better from there. Furthermore, the cloud application was run in the "consumption" tier<sup>4</sup> which is free and therefore does not offer dedicated hardware which could improve performance significantly.

Using the "consumption" tier might be one reason for the poor performance of the cloud application compared to the performance achieved on the physical device. When running applications using Azure Functions "consumption" tier, hardware and generally resources are shared amongst multiple and even foreign applications. This can lead to very different results within seconds as runs might be scheduled to different physical machines and or the load of a machine changed as new functions were started. This is a big disadvantage if stable performance is necessary and can only be fixed by upgrading to a paid plan. Another factor that underlines this argument is the way Azure Functions applications are designed and ran. There is no continuously running process waiting for messages, but most parts of the application are torn down and started up again with every message which associated with the unsteady performance when using shared resources, can increase outliers even more.

However, it can be shown that parallelization can improve the overall performance by a lot as shown by Figure 5.10. As previously discussed, the horizontal scaling used by Azure Functions has a very good influence on the total time needed to process all message, when execution times increase. For small execution times it is generally not advisable as the scheduling algorithm takes 100-300 milliseconds to schedule. Azure Functions offers the functionality of batch-processing which creates a solution improving

<sup>4</sup><https://azure.microsoft.com/en-us/pricing/details/functions/>

the overall execution time of the *vary non-applicable conditions* and *vary applicable conditions* setting but worsen performance in the *vary actions* setting as long as no local threading is implemented.

Which option to prefer depends on the requirements. In general the cloud application should be able to be tuned to perform at least as good as the *Desktop* hardware. As hardware resources are usually very limited on typical IoT devices collecting sensor data and reporting it to their digital twins, it can be said that running conditions in the cloud application is more desirable. It also offers a good separation of concerns compared to the devices' implementation. Nevertheless, if performance is most important and latency is considered as well, the execution on the physical IoT device might be better.

## 5.5 Limitations and Threads to validity

The validity of research deals with the question of how the conclusions might be wrong. In non-technical terms validity is concerned with "How any result might be wrong". Validity is not something that can be proven or guaranteed, but is rather a goal that is tried to be achieved. For quantitative research in software engineering Wohlin et al. [WHH03] define four types of validity threads which are discussed for the context of this work in the following.

**Conclusion Validity:** *"Does the treatment/change we introduced have a statistically significant effect on the outcome we measure?"* The applied solution shows significant influence in the observed results. The in Section 5.2.3 displayed numbers show an overall improved result that can be attributed to the introduced changes. As these resulting numbers are compared to those of the reference implementation which also acts as the base implementation in this thesis it can be said that the changes introduced in this thesis have a direct influence on the results.

**Internal Validity:** *"Did the treatment/change we introduced cause the effect on the outcome? Can other factors also have had an effect?"* It can't be ruled out that other factors might have had impact on the gathered results. For RQ1 (creation, deployment model-driven DT) and RQ2 (connecting DT and device) the external influence can be neglected based on the defined "manual task" definition, as it compared the same process with two different approaches. Regarding RQ3 (monitoring DT data), when measuring execution time external factors can influence the result significantly. Doing multiple runs within each experiment and statistical methods like average and mean were used to try to eliminate or reduce external influence. A good example for external influence was already discussed in the last section as potentially faster hardware delivered lower execution-times.

**Construct Validity:** *"Does the treatment correspond to the actual cause we are interested in? Does the outcome correspond to the effect we are interested in?"* In all cases the achieved outcome corresponds to the effect we are interested in. For RQ1 (creation, deployment model-driven DT) and RQ2 (connecting DT and device) the created framework

(DTAF) (treatment) shows the expected effect by reducing manual efforts when creating and deploying digital twins and the connected physical devices. While the proposed solution shows an improvement in comparison to the original solution, it might not be the only solution to the posed question. Other solutions might even further improve the situation. The solution implemented to monitor data exchanged between physical device and DT (RQ3) while fulfilling the need to monitor DT data, might not be generally applicable and represents just a single solution of many.

**External validity:** *"Is the cause and effect relationship we have shown valid in other situations? Can we generalize our results? Do the results apply in other contexts?"* The research conducted in this thesis shows once again that using Continuous Integration and Delivery (CI/CD) and automation in general helps to reduce manual efforts. Automation is generally defined as the use of technology and machinery to perform tasks with reduced human intervention. CI/CD on the other hand enables the usage of automation during specific phases of the Software Lifecycle. The proposed solution uses CI/CD and therefore automation to allow different input models and still produce the observed improvements. Some parts of the solution like the ATL execution CLI (Section 4.1.1) for instance can also be generally reused even in non-digital Twin scenarios. Other parts of the solution are very specific and may not be able to be reused.

## Related Work

In this thesis, multiple artifacts are developed. The combination of these artifacts define a workflow to automatically create the necessary digital twins and physical device data, but also to deploy them to a certain target environment and enable automatic validation of data sent to the digital twin. All of this is done with the goal of high automation and reducing manual effort. This chapter compares the results from the previous chapters to related approaches found in the literature.

To be able to compare related results an adequate literature research is conducted using online literature databases. Returned results are evaluated ordered by applicability, and then, the most suitable literature is summarized and used for comparison.

Digital twins are not created by only defining the digital twins describing document (digital twin model) but need to be created within or as a platform which provides interfaces to interact with the digital twin. To interact with a certain digital twin or digital twin platform, usually certain configurations have to be done on physical devices. These configurations consist of certain endpoints, authentication parameters and identifiers which connect physical devices to a digital twin instance. All these steps have to be taken into account when searching for comparable literature. The conducted literature research can be grouped into the following areas automated digital twin model creation, twin deployment, twin connection establishing and digital twin data monitoring. A short summary of each identified research together with a comparison to the implemented Digital Twin Automation Framework (DTAF) of this thesis is given in the next paragraphs.

Garcia et al. [GC19] highlights the integration of MDE artifacts in Continuous Software Engineering (CSE) as a beneficial situation for all software practitioners. In their work, they are covering two levels of MDE integration into modern continuous deployment (CD) pipelines. First the integration of modeling artifacts as standalone executable components and secondly an MDE artifact as the target of the CD where model-to-model and model-

---

to-text transformations are used to generate the final artifact. An example using ATL and MOFScript was given to transform a certain input model to an HTML<sup>1</sup> page. Further research was done regarding the evolution of metamodels and co-evolution and their corresponding models. The CD pipeline was extended to automatically detect changes and calculate whether the changes done would have breaking or non-breaking character. Breaking changes could then be either automatically solved by co-evolving the associated models and transformations, or trigger manual interaction to co-evolve manually. **Comparison to DTAF:** Both Garcia et al. in their work and this work point out the "weakness" of MDE and its high coupling to its tools. EMF and extensions to EMF are closely coupled to not only the Eclipse project, but also the Eclipse IDE runtime, which makes it hard to use in automated environments like CD pipelines. The solution used by Garcia et al. uses a similar approach utilizing Jenkins<sup>2</sup> and Java commands in batch scripts to execute transformations. The solution of this work described in Section 4.1.1 is more advanced regarding the external libraries necessary using Maven dependencies, and in general, more up to date. The ATL transformation engine created in this work is independent of the Eclipse runtime and can be reused with more ease.

With *MDCI* García Díaz et al. [GDPCL12] proposes a *process in which the continuous integration practice can be applied to model-driven software developments in an effective way*. The creation of *MDCI* was mainly reasoned by the usage of continuous integration (CI) tools due to their benefits of cost and risk reduction and the usage of MDE in general to reduce risk and improve a product's quality. In their work, a solution using multiple version controlled repositories namely Domain repository, Application repository and Models repository combined with a CI tool and an incremental model generator was proposed to evaluate (1) the number of generated artifacts, (2) the size of such artifacts, (3) the generation time of the artifacts, and (4) the deployment time to a production environment. **Comparison to DTAF:** While the *MDCI* implementation is not described in detail, the evaluation and its results are of interest and can be compared. The evaluation compares traditional MDE without version control and CI against MDE with continuous integration. In all conducted case studies using MDE without the usage of a CI pipeline created more artifacts, with a larger size and both the generation time and the deployment time was significantly higher. This can be compared to the manual tasks reduction derived in Section 5.2 where around 78% of the necessary actions could be removed due to automation.

In Kirchhof et al. [KMR<sup>+</sup>20], a cyber-physical-system (CPS) with the corresponding DT is created using *MontiArc*<sup>3</sup> and *MontiGem*<sup>4</sup>. To create and connect the created CPS and DT the following 4 steps are done: (1) develop the CPS architecture, (2) develop digital twins information system (DTIS), (3) connect CPS properties with DTIS attributes (tagging), (4) use tags to generate communication and synchronization infrastructure. Using specific information (tags) in the CPS and DTIS models

---

<sup>1</sup><https://html.spec.whatwg.org/>

<sup>2</sup><https://www.jenkins.io>

<sup>3</sup><https://github.com/MontiCore/montiarc>

<sup>4</sup><https://www.se-rwth.de/projects/#MontiGem>

---

common parts can be identified. These common parts identify necessary information that is needed in both systems to enable communication and keep systems in sync. Using model-to-model transformations and previously mentioned tools *MontiArc* and *MontiGem* provide a very efficient solution to prototype and create DTs. **Comparison to DTAF:** The solution proposed by Kirchof et al. shows different tooling to solve a similar problem. The usage of two dedicated models and a corresponding tagging to match information together can be seen superior to the solution implemented in this work. As described in Section 4.2.3 the disadvantages or shortcomings of the proposed solution due to name mapping and the resulting complexity can be overcome by such a solution. On the other hand is the complexity of generating dedicated interfaces and endpoints for all *DTIS* (which can be compared to deployed digital twin models) high, in comparison to generating a model based on a certain language (Azure Digital Twins Definition) and following a certain contract when exchanging data between digital twin and physical device such as proposed in Section 4.2.3.

Michael and Wortmann [MW21] extend the work of Kirchof et al. by envisioning how to use a model-driven low-code approach for configuring digital twins in a Low-Code Development Platform (LCDP). *MontiArc* and *MontiGem* are technologies to not only create the digital twins but also a complete digital twin cockpit (DTC) offering operations to analyze and adapt values of a digital twin. To do so, it relies on model-to-model and model-to-text transformations to generate various artifacts as well as source code to run a digital twin. Instead of using one of the various available digital twin platforms, this solution can run platform independent. **Comparison to DTAF:** A digital twin cockpit can be visualized using the *Azure Digital Twin Explorer* when using the *Azure Digital Twins Platform*. But it is not possible to integrate any monitoring or alerts into it. The DTC on the other hand could potentially be extended to not only show value of digital twins but also create rules and alerts. These rules and alerts could integrate more nicely into the platform than the proposed solution of this work.

Ejersbo et al. [ELFE23] researched another aspect of digital twin deployment by analyzing and proposing a solution how to replace a digital twin during runtime without requiring to shut down the operation of the DT or the underlying CPS. A "model swap mechanisms" is proposed to integrate new models during runtime without service interruption, by using an orchestrator to swap between multiple DT instances. Tests conducted using a *Desktop Robotti*<sup>5</sup> robot prototype show the exchange of the DT feeding data into the robot during runtime. The mechanism makes use of synchronization, step and swap conditions to align the different DT instances and enable a smooth transition. Dobaj et al. [DRK<sup>+</sup>22] takes this idea a step further and evaluates different software deployment strategies for digital twins enabling zero downtime but also real-traffic testing and user targeting. To achieve this, Dobaj et al. not only use CI/CD pipelines but also modularization to address technical evolution, interoperability and the ability to independently continuously develop operations and management functions. To enable A/B testing scenarios with

---

<sup>5</sup>[https://gitlab.au.dk/desktop\\_robotti](https://gitlab.au.dk/desktop_robotti)



---

digital twins, a complex data-flow model was designed which uses a management digital twin that gives control input to the physical system which will then forward business data to the according digital twin to process the information in a time-synchronous operation. This is done to enable comparability of both A and B-path of this solution. **Comparison to DTAF:** Ejersbo et al. use advanced CI/CD pipelines to deploy digital twins like proposed in this work, but take this solution not only one but many steps further by integrating digital twin evolution, deployments without interruption, A/B testing and many more features. Many modern systems rely on A/B testing to verify and validate new developments or functionality.

Gautham et al. [GJRE21] propose a DevOps process using model-based design and engineering for a tighter integration of design verification activities and runtime monitoring properties. More precisely validation conditions were derived from design time models and used during runtime to monitor the system while running. The work focuses heavily on how to (re-)build requirements based upon an executable model and by doing so creating and/or improving conditions for runtime verification by applying a DevOps workflow. **Comparison to DTAF:** The methodology proposed by Gautham et al. [GJRE21] differs a lot to the implementations done in this work. Specifically how conditions are derived from the design time model and are improved and extended using the DevOps workflow is a great addition. Using this process as a post model creation iterative approach could enable another layer of work-sharing and have teams specifically focus on runtime verification. It has to be said that the verification solution implemented in this work is superficial and not real-world relevant. Building constraints in *Object Constraint Language* (OCL) or using other MDE verification techniques would greatly improve the proposed solution.

A complete digital twin deployment stack is proposed by Hugues et al. [HHHY20]. An *SysML* input model is used as an abstraction of a system to monitor humidity and temperature in multiple points of a building. The system shall collect the data and store it in a central place. It should further detect and report errors in the reported data and monitor the systems' health status and report issues. The *SysML* model is refined into a *AADL*<sup>6</sup> model. To enable verification and validation the given requirements are used to create *ALISA*<sup>7</sup> verification rules which integrate to the *AADL* model. In a CI/CD pipeline these artifacts are transformed using model transformations to build a deployable artifact including the digital twin source code, the monitoring functions and rules created via *ALISA* rules. Using the *GitLab*<sup>8</sup> DevOps CI/CD pipeline the generated source code is compiled, containerized and deployed to the *Microsoft Azure* platform. **Comparison to DTAF:** The goal and used methodology are pretty much comparable to the approach expelled in this work. Details like the used modeling languages and technologies vary but the main part, the pipeline, is comparable. Again a different solution for validation and verification is used which shows that there are other and

---

<sup>6</sup><http://www.openaadl.org/>

<sup>7</sup><https://github.com/osate/osate2/tree/master/alisa>

<sup>8</sup><https://about.gitlab.com/>

---

better integrated solutions available. However, Hugues et al. [HHHY20] work covers only the digital twin creation and deployment, not the deployment and configuration of the associated physical device.

# Conclusion and Future Work

## 7.1 Conclusion

The steady increase in complexity of software and computer systems requires new solutions to simulate, monitor and optimize these systems. Digital twins step in to fulfill this need, however their creation and maintenance requires manual tasks to be performed with each change to the real world system.

In this thesis, the aim was to analyze to which extent a model-driven approach can be used to reduce manual efforts for creation, deployment and configuration of digital twins. Additionally, the execution time of monitoring rules executed on physical devices was compared to those of a cloud application. Therefore, multiple artifacts were developed and combined in a *DevOps* pipeline to enable high automation. The first artifact transforms a given model into an *Azure Digital Twins Definition* model and creates the digital twin in the *Azure Digital Twins Platform*. The second artifact generate necessary configurations and creates *Azure IoT Hub* connection credentials for all associated physical devices. Additionally, the created configurations can be automatically deployed to the device together with the necessary controller software to send real-time data to the *Azure Digital Twins* service. Finally, a monitoring framework was created that is based up upon conditions and actions. To be able to define these conditions and actions, the input model was adapted, and the transformation was extended to automatically build monitoring conditions in *JsonPath*. These *JsonPath* conditions and associated actions are evaluated in the device controller software and *Azure Digital Twins* platform to compare the performance scalability.

An evaluation of the created artifacts yielded the following results. The pipeline combining creation of the digital twins and configuration of the physical devices reduced the manual efforts by 78% and 68% considering all executed evaluation cases. To profit from these improvements only minor changes to the input model were necessary.

The performance experiments to determine monitoring execution performance were conducted on two different physical devices and in the *Microsoft Azure* cloud infrastructure. Different implementations of *JsonPath* evaluators were used due to different programming languages used. Three different settings were applied to investigate which parameter variation results in which performance impact. The results show that varying non-applicable conditions or executed actions does not significantly affect the overall performance. However, the third scenario (varying the count of applicable conditions) showed performance impact on the physical devices while the cloud application showed no impact. As different libraries and programming languages were used, this can be seen as the source of this difference in performance.

Compared to other approaches investigated, the created pipeline and underlying artifacts offer a "full" solution, while most researches investigate a particular area. In various areas there are other solutions available which might be superior to the created solution when it comes to usability or complexity.

## 7.2 Future Work

In this section, directions for future work are proposed.

**Model evolution- and co-evolution:** The proposed pipeline is only capable of creating digital twins. Due to the nature of cyber-physical systems and their models these evolve over time which makes supporting model evolution and co-evolution in the pipeline necessary. Various research [KSW19][LGWC21] was already conducted in this direction and should be realized in the pipeline.

**Model package management:** Package management has important role in software engineering offering the version-controlled access to internal and external libraries. When working within Model-Driven Engineering models can usually be referenced locally in a registry (*EMF*) or externally via file URLs. This puts the developer into charge when it comes to accessing these models and handling multiple versions. To reduce coupling and enable access to certain versions of metamodels, a *Maven Repository*<sup>1</sup> or *NuGet*<sup>2</sup> like solution should be implemented and introduced. By doing so, the repository of a certain model would be reduced down to the "real" unit of work. Reusability could also be drastically improved.

**Model project build tool:** In software engineering build tools play a big role especially when it comes to automation and reproducible builds in continuous integration (CI) and continuous delivery (CD) pipelines. While project management and build tools like *Maven* and *Gradle* can be used in *EMF* projects to manage dependencies, there are currently no plugins to support *EMF* language features. Introducing predefined build tasks for *Maven* or *Gradle* could ease up or even make scripting in CI/CD pipeline obsolete. Weghofer[Weg17] showed how operations can be simplified by using his *Moola*

<sup>1</sup><https://maven.apache.org/repository/index.html>

<sup>2</sup><https://www.nuget.org/>

framework. Another approach was proposed by Kolovos et al. [KPP08a] where the *Apache Ant* build tool was extended by *epsilon* tasks to load, transform or merge models.

**Digital twin communication layer:** This thesis proposed a digital twin that is integrated into the physical device. This solution is simple and fast but is typically an unwanted situation in real-world scenarios. As real-world systems usually consist of lots of different devices that are not only developed in-house, integrating a digital twin communication layer into third-party software is hardly ever possible. Therefore, different alternative solutions should be proposed, developed and evaluated. Existing communication within cyber-physical systems could be proxied or duplicated so that a middleware could be used to extract and send the necessary information to the digital twins' data. This solution should be able to understand and communicate using different state-of-the-art protocols like HTTP, MQTT, ModBus and others.

**Multi-layer modeling:** Current Model-Driven Engineering approaches make use of a single model and metamodel to express all characteristics of a certain real-world entity. In this thesis, an existing model describing a real-world system was extended by (1) digital twin communication credentials, (2) digital twin and digital twin property naming and (3) monitoring rules and associated actions. This information does not necessarily belong to the same level or layer of the model and should be moved to a different level of abstraction. More precisely a different model that is built upon the existing model that describes the real-world system. By doing so engineers from different areas can work on their part of the whole independently of others and enable parallelization, and by that, more efficient work. García Díaz et al. [GDPCL12] propose a multi repository setup where a similar approach is chosen which could be adopted and enhanced to improve usability and separate different concerns within MDE.

# List of Figures

2.1	Overview of MDE methodology [BCW12]	6
2.2	Model-driven-engineering pyramid [LZN <sup>+</sup> 14]	7
2.3	ATL compilation process <sup>3</sup>	8
2.4	ATL Core Architecture <sup>4</sup>	9
2.5	Conceptual <i>DTDL</i> metamodel in UML class diagram notation	11
2.6	<i>Azure Digital Twins</i> solution overview <sup>5</sup>	11
2.7	"DevOps overall conceptual map" by Leite et al. [LRK <sup>+</sup> 19]	13
3.1	System architecture for automated Digital Twin Development and Operations	18
4.1	<i>ATL</i> execution configuration in <i>Eclipse</i>	22
4.2	Artifacts published by pipeline	27
4.3	Device to Digital Twin communication flow	31
4.4	Extended conceptual <i>DTML</i> metamodel in UML class diagram notation [LSV <sup>+</sup> 21]	32
4.5	Client software architecture diagram in UML class diagram notation representing the device controllers' <i>Sensor Layer</i> with 3 example sensor implementations for air quality sensors	34
4.6	<i>DTML</i> condition modeling capabilities in UML class diagram notation	43
5.1	Graphical representation of the scenarios baseline	51
5.2	<b>Predefined digital twin</b> setup in UML object diagram notation	52
5.3	Adapted digital twins for Evaluation Case 2 (EC2) and Evaluation Case 3 (EC3) in UML object diagram notation	54
5.4	Experiment I: visualization of simulation runs for <i>vary number of non-applicable conditions</i>	63
5.5	Experiment I: visualization of simulation runs for <i>vary number of applicable conditions</i>	64
5.6	Experiment I: visualization of simulation runs for <i>vary number of actions</i>	65
5.7	Experiment II: visualization of message throughput of <i>vary non-applicable conditions</i> on the cloud application	68
5.8	Experiment II: visualization of message throughput of <i>vary number of applicable conditions</i> on the cloud application	69
		84

5.9	Experiment II: visualization of message throughput of <i>vary number of actions</i> setting on the cloud application . . . . .	69
5.10	Experiment II: visualization of relative execution start time and execution time . . . . .	70

## List of Tables

5.1	Mapping of requirements to evaluation tasks. . . . .	50
5.2	Count of <i>manual actions</i> necessary to complete evaluation cases in <i>infrastructure actions</i> (infra), <i>model manipulations</i> (model) and <i>deployment actions</i> (deploy) . . . . .	55
5.3	Parameters used to evaluate the monitoring frameworks performance on the physical device . . . . .	61
5.4	List of simulation settings with the associated RMSE values . . . . .	62
5.5	Calculation details for delay-adjusted execution-time influence of actions based on <i>Laptop</i> hardware results . . . . .	65
5.6	Parameters used to evaluate the monitoring frameworks performance in the server-less cloud application . . . . .	66
5.7	List of simulation settings with the associated RMSE values for experiment II	67



# List of Listings

4.1	Except of <i>Java Xtend</i> transformation runner . . . . .	25
4.2	Pipeline Model-to-Model transformation command . . . . .	26
4.3	Pipeline Model-to-Text transformation command . . . . .	26
4.4	Creation of digital twin models in <i>Azure Digital Twin</i> service . . . . .	28
4.5	Creation of digital twin instance in <i>Azure Digital Twin</i> service . . . . .	28
4.6	Creation of digital twin relation in <i>Azure Digital Twin</i> service . . . . .	28
4.7	Additions to <i>DigitalTwinEnvironment</i> to include the digital twin <i>AzureResource</i> . . . . .	29
4.8	<i>Azure</i> release pipeline commands . . . . .	30
4.9	Creation of a device in the <i>Azure IoT Hub</i> service. . . . .	32
4.10	Extraction of the connection-string for a certain device in the <i>Azure IoT Hub</i> service. . . . .	33
4.11	Xtend - JSON serialization helper . . . . .	35
4.12	C# - Azure Digital Twin bridge function . . . . .	36
4.13	script_to_run example for testing purposes . . . . .	38
4.14	Excerpt of the python deploy script . . . . .	38
4.15	<i>Xtext</i> transformation for conditions and actions . . . . .	44
4.16	Serialized conditions and actions example . . . . .	45
4.17	script_to_run example with conditions and actions file . . . . .	45
4.18	C# application setting up an <i>Azure App Configuration</i> configuration provider . . . . .	46
4.19	Conditions and actions deployment script for <i>Azure App Configuration</i> . . . . .	46
4.20	<i>Application Insights</i> query . . . . .	47
5.1	<i>0-1-10 test-set used in experiment I and II</i> . . . . .	60

# Bibliography

- [AK22] Colin Atkinson and Thomas Kühne. Taming the Complexity of Digital Twins. *IEEE Software*, 39(2):27–32, March 2022.
- [BCE<sup>+</sup>20] Francis Bordeleau, Benoit Combemale, Romina Eramo, Mark van den Brand, and Manuel Wimmer. Towards Model-Driven Digital Twin Engineering: Current Opportunities and Future Challenges. In Önder Babur, Joachim Denil, and Birgit Vogel-Heuser, editors, *Systems Modelling and Management*, Communications in Computer and Information Science, pages 43–54, Cham, 2020. Springer International Publishing.
- [BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*, volume 1. September 2012.
- [BDH<sup>+</sup>20] Pascal Bibow, Manuela Dalibor, Christian Hopmann, Ben Mainz, Bernhard Rumpe, David Schmalzing, Mauritius Schmitz, and Andreas Wortmann. Model-Driven Development of a Digital Twin for Injection Molding. In Schahram Dustdar, Eric Yu, Camille Salinesi, Dominique Rieu, and Vik Pant, editors, *Advanced Information Systems Engineering*, Lecture Notes in Computer Science, pages 85–100, Cham, 2020. Springer International Publishing.
- [Béz05] Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, May 2005.
- [BKL<sup>+</sup>12] Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. An Introduction to Model Versioning. In Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio, editors, *Formal Methods for Model-Driven Engineering: 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*, Lecture Notes in Computer Science, pages 336–398. Springer, Berlin, Heidelberg, 2012.
- [BW19] Florian Biesinger and Michael Weyrich. The Facets of Digital Twins in Production and the Automotive Industry. In *2019 23rd International Conference on Mechatronics Technology (ICMT)*, pages 1–6, October 2019.

- [CFJ<sup>+</sup>16] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, Jim Steel, and Didier Vojtisek. *Engineering Modeling Languages : Turning Domain Knowledge into Tools*. November 2016.
- [CW20] Benoit Combemale and Manuel Wimmer. Towards a Model-Based DevOps for Cyber-Physical Systems. In Jean-Michel Bruel, Manuel Mazzara, and Bertrand Meyer, editors, *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, Lecture Notes in Computer Science, pages 84–94, Cham, 2020. Springer International Publishing.
- [DJR<sup>+</sup>22] Manuela Dalibor, Nico Jansen, Bernhard Rumpe, David Schmalzing, Louis Wachtmeister, Manuel Wimmer, and Andreas Wortmann. A Cross-Domain Systematic Mapping Study on Software Engineering for Digital Twins. *Journal of Systems and Software*, 193:111361, November 2022.
- [DMG07] Paul M. Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Pearson Education, June 2007.
- [DRK<sup>+</sup>22] Jürgen Dobaj, Andreas Riel, Thomas Krug, Matthias Seidl, Georg Macher, and Markus Egretzberger. Towards digital twin-enabled DevOps for CPS providing architecture-based service adaptation & verification at runtime. In *Proceedings of the 17th Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '22*, pages 132–143, New York, NY, USA, August 2022. Association for Computing Machinery.
- [EBC<sup>+</sup>22] Romina Eramo, Francis Bordeleau, Benoit Combemale, Mark van den Brand, Manuel Wimmer, and Andreas Wortmann. Conceptualizing Digital Twins. *IEEE Software*, 39(2):39–46, March 2022.
- [ELFE23] Henrik Ejersbo, Kenneth Lausdahl, Mirgita Frasher, and Lukas Esterle. Dynamic Runtime Integration of New Models in Digital Twins. In *2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 44–55, May 2023.
- [GC19] Jokin Garcia and Jordi Cabot. Stepwise Adoption of Continuous Delivery in Model-Driven Engineering. In Jean-Michel Bruel, Manuel Mazzara, and Bertrand Meyer, editors, *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, Lecture Notes in Computer Science, pages 19–32, Cham, 2019. Springer International Publishing.
- [GDPCL12] Vicente García Díaz, B. Pelayo García-Bustelo, and Juan Cueva Lovelle. MDCI: Model-driven continuous integration. *Journal of Ambient Intelligence and Smart Environments*, 4:479–481, September 2012.

- [Gil16] Alasdair Gilchrist. *Industry 4.0*. Apress, Berkeley, CA, 2016.
- [GJRE21] Smitha Gautham, Athira Varma Jayakumar, Abhi Rajagopala, and Carl Elks. Realization of a Model-Based DevOps Process for Industrial Safety Critical Cyber Physical Systems. In *2021 4th IEEE International Conference on Industrial Cyber-Physical Systems (ICPS)*, pages 597–604, May 2021.
- [GJT<sup>+</sup>21] Hari Shankar Govindasamy, Ramya Jayaraman, Burcu Taspinar, Daniel Lehner, and Manuel Wimmer. Air Quality Management: An Exemplar for Model-Driven Digital Twin Engineering. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 229–232, October 2021.
- [Gri11] Michael Grieves. *Virtually Perfect: Driving Innovative and Lean Products through Product Lifecycle Management*. Space Coast Press, Cocoa Beach, FL, 2011.
- [Gri15] Michael Grieves. Digital Twin: Manufacturing Excellence through Virtual Factory Replication. March 2015.
- [Hev07] Alan Hevner. A Three Cycle View of Design Science Research. *Scandinavian Journal of Information Systems*, 19, January 2007.
- [HHHY20] Jerome Hugues, Anton Hristosov, John J. Hudak, and Joe Yankel. TwinOps - DevOps meets model-based engineering and digital twins for the engineering of CPS. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, MODELS '20*, pages 1–5, New York, NY, USA, October 2020. Association for Computing Machinery.
- [JbPT16] Ramtin Jabbari, Nauman bin Ali, Kai Petersen, and Binish Tanveer. What is DevOps? A systematic mapping study on definitions and practices. In *Proceedings of the Scientific Workshop Proceedings of XP2016, XP '16 Workshops*, New York, NY, USA, 2016. Association for Computing Machinery.
- [KMR<sup>+</sup>20] Jörg Christian Kirchhof, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Model-driven digital twin construction: Synthesizing the integration of cyber-physical systems with their information systems. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS '20*, pages 90–101, New York, NY, USA, 2020. Association for Computing Machinery.
- [KPP08a] Dimitrios Kolovos, Richard Paige, and Fiona Polack. A framework for composing modular and interoperable model management tasks. In *Model-Driven Tool and Process Integration Workshop*, (2008/6):79–90, January 2008.

- [KPP08b] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. The epsilon transformation language. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Theory and Practice of Model Transformations*, pages 46–60, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- [Kra16] Edward Kraft. The Air Force Digital Thread/Digital Twin - Life Cycle Integration and Use of Computational and Experimental Knowledge. In *54th AIAA Aerospace Sciences Meeting*, San Diego, California, USA, January 2016.
- [KSW19] Wael Kessentini, Houari Sahraoui, and Manuel Wimmer. Automated meta-model/model co-evolution: A search-based approach. *Information and Software Technology*, 106:49–67, February 2019.
- [LGWC21] Daniel Lehner, Antonio Garmendia, Manuel Wimmer, and Christian Doppler Forschungsgesellschaft. Towards flexible evolution of Digital Twins with fluent APIs. *Proceedings of the 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2021.
- [LPT<sup>+</sup>22] Daniel Lehner, Jérôme Pfeiffer, Erik-Felix Tinsel, Matthias Milan Strljic, Sabine Sint, Michael Vierhauser, Andreas Wortmann, and Manuel Wimmer. Digital Twin Platforms: Requirements, Capabilities, and Future Prospects. *IEEE Software*, 39(2):53–61, March 2022.
- [LRK<sup>+</sup>19] Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojevic, and Paulo Meirelles. A Survey of DevOps Concepts and Challenges. *ACM Computing Surveys*, 52(6):127:1–127:35, November 2019.
- [LSV<sup>+</sup>21] Daniel Lehner, Sabine Sint, Michael Vierhauser, Wolfgang Narzt, and Manuel Wimmer. AML4DT: A Model-Driven Framework for Developing and Maintaining Digital Twins with AutomationML. In *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, September 2021.
- [LZN<sup>+</sup>14] Levi Lúcio, Qin Zhang, Phu Nguyen, Moussa Amrani, Jacques Klein, Hans Vangheluwe, and Yves Le Traon. Advances in Model-Driven Security. In *Advances in Computers*, volume 93, pages 103–152. February 2014.
- [Min97] Barbara Minto. *The Pyramid Principle: Logic in Writing and Thinking*. April 1997.
- [MS95] Salvatore T. March and Gerald F. Smith. Design and natural science research on information technology. *Decision Support Systems*, 15(4):251–266, December 1995.
- [MW21] Judith Michael and Andreas Wortmann. Towards Development Platforms for Digital Twins: A Model-Driven Low-Code Approach. In Alexandre

Dolgui, Alain Bernard, David Lemoine, Gregor von Cieminski, and David Romero, editors, *Advances in Production Management Systems. Artificial Intelligence for Sustainable and Resilient Production Systems*, IFIP Advances in Information and Communication Technology, pages 333–341, Cham, 2021. Springer International Publishing.

- [OMGa] OMG. MetaObject Facility (MOF). <http://www.omg.org/mof/>.
- [OMGb] OMG. System Modeling Language (SysML). <https://www.omg.org/spec/SysML>.
- [OMGc] OMG. Unified Modeling Language (UML). <http://www.omg.org/spec/UML/>.
- [PLWW22] Jérôme Pfeiffe, Daniel Lehner, Andreas Wortmann, and Manuel Wimmer. Modeling Capabilities of Digital Twin Platforms - Old Wine in New Bottles? *The Journal of Object Technology*, 21(3):3:1, 2022.
- [RH09] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, April 2009.
- [RSK20] Adil Rasheed, Omer San, and Trond Kvamsdal. Digital Twin: Values, Challenges and Enablers From a Modeling Perspective. *IEEE Access*, 8:21980–22012, 2020.
- [TZLN19] Fei Tao, He Zhang, Ang Liu, and A. Y. C. Nee. Digital Twin in Industry: State-of-the-Art. *IEEE Transactions on Industrial Informatics*, 15(4):2405–2415, April 2019.
- [VP04] Dániel Varró and András Pataricza. Generic and meta-transformations for model transformation engineering. In Thomas Baar, Alfred Strohmeier, Ana Moreira, and Stephen J. Mellor, editors, «UML» 2004 — *The Unified Modeling Language. Modeling Languages and Applications*, pages 290–304, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
- [VTBW21] Cor Verdouw, Bedir Tekinerdogan, Adrie Beulens, and Sjaak Wolfert. Digital twins in smart farming. *Agricultural Systems*, page 19, 2021.
- [Weg17] Stefan Weghofer. Moola - a Groovy-based model operation orchestration language. Master’s thesis, TU Wien, 2017.
- [WHH03] Claes Wohlin, Martin Höst, and Kennet Henningsson. Empirical Research Methods in Software Engineering. In Reidar Conradi and Alf Inge Wang, editors, *Empirical Methods and Studies in Software Engineering: Experiences from ESERNET*, Lecture Notes in Computer Science, pages 7–23. Springer, Berlin, Heidelberg, 2003.

- [Wie14] Roel Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. January 2014.
- [WMWH20] Sabine Wolny, Alexandra Mazak, Manuel Wimmer, and Christian Huemer. *Model-Driven Runtime State Identification*. Gesellschaft für Informatik e.V., 2020.
- [ZBC16] Liming Zhu, Len Bass, and George Champlin-Scharff. DevOps and Its Practices. *IEEE Software*, 33(3):32–34, May 2016.